CODESIMILARITY: AN APPROACH FOR CLUSTERING INTRODUCTORY PROGRAMMING ASSIGNMENTS

BY

JONATHAN OSEI-OWUSU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Adviser:

Professor Tao Xie

ABSTRACT

Enrollment in introductory programming (CS1) courses continues to surge and hundreds of CS1 students can produce thousands of submissions for a single problem, all requiring timely feedback and accurate grading. While not exclusive to CS1 courses, instructors of such courses are challenged to provide feedback at scale (e.g., to hundreds of students). Because these students have a diverse range of skills and backgrounds, it is essential to differentiate common strategies and shortcomings of student submissions to a given problem. There is a strong need for clustering submissions by the similarity of their strategies for enabling instructors to provide customized feedback to students. To fill this need, in this thesis, we present the CodeSimilarity approach, which first automatically generates test data for correct student submissions and then uses semantic program features (i.e., path conditions) to cluster correct student submissions by their strategies. We define the strategy employed by a student submission as the way that the problem space is partitioned into sub-spaces and how the problem is uniquely addressed within each sub-space. In particular, CodeSimilarity leverages automated test generation based on symbolic execution to determine the path conditions for a given submission; comparing each submission's path conditions allows to establish behavioral equivalence relationships with respect to the strategies employed by these submissions. We evaluate CodeSimilarity on four datasets to assess the effectiveness of our approach. The evaluation results show that by using semantic program features (i.e., path conditions), CodeSimilarity can effectively cluster submissions that employ the same strategy.

To my A-Team, family, and beloved friends.

ACKNOWLEDGMENTS

For those who have seen me during the highs and lows, you know that these past two years have been nothing short of a wild journey.

I specifically thank my advisor, Professor Tao Xie, for being committed to challenging the way that I think and work so that I may learn to become a better researcher. I am also very grateful to have met and be mentored by my Angello; thanks for putting up with my antics. Thank you so much, Liia, for being a friend and confidant through thick and thin. Thank you, Steven, Zirui, Linyi, Wenyu, Ping, Zhengkai, Wing, Ben, Yue, and Xueqing, Kaiyuan for being such fun, patient, and relatable group members. Thank you so much, Faria, for being the most encouraging person I may have ever met. Thank you, Jacob and August, for sacrificing sleep to review my first conference paper. Thank you, Derek, for being an amazing roommate and for always being willing to review my writing. I also thank my A-Team and my friends from the Alpha Omega Campus Ministry for always being in my corner and encouraging me through the hills and valleys. I sincerely thank The National GEM Consortium and Siebel Scholars Foundation for funding my education and allowing me to have a wonderful graduate school experience.

I am sure that I missed some names and mentions, so please forgive me. Know that those of you reading this thesis have been an astounding help—I would not have been able to make it through the Master's program had it not been for your help. To God be the glory!

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Enrollment in introductory programming (CS1) courses continues to surge at remarkable rates [1]. A large number of students produce an even larger number of submissions to programming problems, each of which must be graded and assessed to provide feedback. This problem is further exacerbated when CS1 is taught using many small problems. Using many small problems has become a successful and increasingly widespread approach [2]. To provide customized feedback in the form of partial credit at scale (e.g., to hundreds of CS1 students), instructors must inspect individual submissions to identify faults committed by students. For instructors, however, it is nearly impossible to inspect thousands of submissions in a timely manner. For students, on the other hand, timely feedback is critical for their learning experience [3]. While this dilemma is not exclusive to CS1 courses, there is a strong need to provide tool support for instructors and provide scalable, customized feedback to students to maintain the quality of education in larger classrooms.

Many researchers realize this need and have developed approaches [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] to analyze programming assignments at scale so that instructors can provide timely, customized feedback to students. To facilitate giving feedback at scale, many of these approaches group into clusters programming submissions that are semantically or syntactically similar. More specifically, these approaches may randomly select and analyze one correct representative from each cluster, and then propagate the generated feedback to other submissions of the same cluster.

Clustering approaches such as OverCode [6] and CLARA [7] place much emphasis on syntactic features and require these program features to match exactly. This requirement results in an excessive number of clusters, where semantically similar programs with small syntactical differences are placed in different clusters [4]. Clustering approaches such as TipsC [9] rely on expensive computation, e.g., computation of the Levenshtein distance between Abstract Syntax Trees (ASTs). This expensive computation greatly hinders the scalability of such approaches. Clustering approaches such as our previous work called Grasa [17] take a more abstract view of the student submissions by grouping those submissions that produce the same outputs over the input space. The grouped submissions are considered to be *approximately behaviorally equivalent*. While Grasa is able to form clusters based on this approximate behavioral similarity, it is unable to distinguish between the different strategies taken in programming assignments (e.g., a bubble sort implementation versus an insertion sort implementation). Consequently, two submissions employing two different strategies may be classified as approximately behaviorally equivalent since both submissions yield the same

output over a given input space. The issue with this strategy-blind clustering is particularly problematic during hint generation—without a cluster of student submissions employing the same strategy, consistently providing relevant hints to a new submission is not possible.

To address the issue of strategy-blind clustering and the provision of feedback at scale, we propose CodeSimilarity, a new approach for clustering student coding submissions into equivalence classes based on their program strategy to provide partial credit to students. Our approach relies on semantic features (i.e., *path conditions*) to determine a submission's program strategy and subsequently form equivalence classes. A path condition [18, 19] denoted as ρ consists of conjuncted predicates $(\phi 1, \phi 2, ..., \phi_{|\rho|})$ collected from the executed branch conditions in a program's execution path.

The *strategy* [4] employed in a given program is defined by how the problem space is partitioned into sub-spaces and how the problem is uniquely addressed within individual sub-spaces. More intuitively, let us consider two elementary sorting algorithms: insertion sort and bubble sort. Although both algorithms share the same time complexity, they leverage different strategies to sort their inputs. What makes the two strategies different is the sequence that the partitions or sub-spaces (i.e., input array elements) are handled and changed. When programmers design their programs, they not only need to design suitable control structures to partition the input space, but they must also decide the operations to use and define how they interact with inputs and memory [4]. Existing approaches [6, 7] group programs with the same variable sequences into the same cluster, i.e., those programs that have the same sequence of values during the programs' execution on the same input data.

Apart from enabling instructors to write feedback about a single submission and propagating the feedback to all other submissions belonging to the same cluster, clustering student programs helps instructors better understand the distribution of common strategies being employed in student submissions [11]. Instructors may use this insight to tailor the class lectures and assignments to meet the needs (i.e., learning styles) of the students in the course. For example, if an instructor is to assign a problem involving the Fibonacci sequence, and the majority of students choose to employ the iterative solution instead of the simpler recursive one, then we may infer that the students are less comfortable with recursion and could benefit from further instruction on the matter.

Two main program features can be used to cluster programs: syntax and semantics. Syntax-based clustering approaches [6, 7] place much emphasis on syntactic program features and require the program features to match exactly. This emphasis results in an excessive number of clusters, where semantically equivalent programs with small syntactical differences may not be clustered together. Syntax-based clustering is error-prone and susceptible to

minor variations between submissions, thereby causing misclustering.

In contrast, our proposed CodeSimilarity approach uses semantic features (i.e., path conditions) to cluster correct[1] student submissions according to our definition of program strategy. To perform the clustering, CodeSimilarity uses two techniques: test generation and path-condition-based clustering.

In the technique of test generation, we use a structural test generator such as Pex [20] to produce test data and corresponding path conditions for each student submission. Note that such structural test generators are not guaranteed to produce the same set of test data when applied on two submissions, respectively. Thus, we need to normalize the generated test data for the submissions so that we can fairly compare the path conditions pertaining to each submission over the input space. To normalize the generated test data for the submissions, we take the union of all the test data generated by our structural test generator for each submission, respectively, and place these unioned test data into a set to capture only the unique test data. Then, on each submission, we configure and invoke Pex again to collect the path conditions corresponding to the unique test data (without generating additional new test data).

Then in the technique of path-condition-based clustering, we leverage a theorem prover to determine the submissions that have equivalent path conditions with respect to each unique test data. The submissions with equivalent path conditions are considered to have the same strategy and are clustered together.

We evaluate CodeSimilarity on four datasets: CodeHunt [21], Pex4Fun APCS (Advanced Placement Computer Science) [22], University Assignments (a collection of student submissions from a CS1 course taught at a major university), and Algorithms (a collection of publicly available sorting algorithms implemented in C#). The evaluation results confirm that our approach effectively clusters student submissions by strategy.

In summary, this thesis makes the following main contributions:

- We propose the CodeSimilarity approach for clustering student submissions by semantic program features (i.e., path conditions) that capture strategy.
- We create a taxonomy that classifies the strategies used in each student submission, enabling us to quantify the false positives produced by our approach.
- We evaluate and demonstrate the effectiveness of our approach in clustering student submissions by strategy on four datasets.

[1]We focus on the correct submissions because, in general, there are fewer ways for students to get a problem correct than incorrect. Therefore, focusing on this subset of submissions would be easier for our approach to handle during our initial experimentation. We discuss this idea further in Chapter 6.

The rest of the thesis is organized as follows. Chapter 2 highlights potential use cases of our approach and a motivating example to illustrate the problem that we address in this thesis. Chapter 3 describes our CodeSimilarity approach in detail. Chapter 4 discusses our evaluation of CodeSimilarity on four datasets. Chapter 5 discusses related work. Chapter 6 concludes this thesis.

CHAPTER 2: USE CASES AND EXAMPLE

2.1 USE CASES AND APPLICATIONS

2.1.1 Specialized feedback based on strategy

Since our CodeSimilarity approach can be easily extended to incorrect student submissions, one potential use case for our approach is providing personalized feedback to address the myriad of mistakes that may exist in incorrect student submissions. For example, assume that an instructor has prompted students to use the quick sort algorithm to sort an integer array in ascending order and return the largest element. Figure 2.1 shows one student's (Student A) code submission. We see that this student implemented the bubble sort algorithm instead. Our approach may provide the following personalized feedback: "Your code passes all of the test cases but does not use the quick sort strategy. Consider using a divide-and-conquer recursive technique instead." If a different student (Student B) submits the code in Figure 2.2, the following feedback may be given: "Your program passes all of the test cases, but does not implement any sorting algorithm." If a third student (Student C) submits a buggy implementation of the correct answer, as shown in Figure 2.3, the following feedback may be given: "You are almost there! Your algorithm yields runtime errors (IndexOutOfRangeException) but employs the same strategy as the correct solution. Check the bounds of your loops as well as your recursive calls for off-by-one mistakes."

2.1.2 Partial credit/penalization

Another use case of the CodeSimilarity approach is awarding partial credit to submissions based on how closely their employed strategies adhere to the assignment prompt. Continuing with the previous example, we see that Student C's submission (Figure 2.3) may be awarded the greatest amount of partial credit because, although the correct answer is not reached, the program strategy does match the expected answer's and, except for the runtime errors, nearly arrives at the correct solution. Student A's submission (Figure 2.1) may be awarded fewer partial credit points than Student A's because it passes the test cases and implements a sorting algorithm, albeit the incorrect one. Student B's submission (Figure 2.2) may be awarded the least amount of partial credit because, despite that the correct answer is produced, no sorting algorithm is implemented (Student C's submission, instead, utilizes C#'s built-in `Max()` function).

```
// Student submission A
using System;
public class Program {
  static void bubbleSort(int[] arr) {
    for (int j = 0; j <= arr.Length - 2; j++)
      for (int i = 0; i <= arr.Length - 2; i++)
        if (arr[i] > arr[i + 1]) {
          temp= arr[i + 1];
          arr[i + 1] = arr[i];
          arr[i] = temp;
        }
  }
  static int Puzzle(int[] arr) {
    bubbleSort(arr);
    return arr[arr.Length - 1];
  }
}
```

Figure 2.1: Student A implements the incorrect sorting algorithm (bubble sort instead of quick sort).

```
// Student submission B
using System;
public class Program {
  static int Puzzle(int[] arr) {
    return arr.Max();
  }
}
```

Figure 2.2: Student B uses C#'s built-in `Array.Max()` method, without performing any sorting.

2.1.3 Tracking of how student submissions move through different clusters

In this use case, we could see how students' code changes as they progress from incorrect to eventually correct submissions. In the Code Hunt [21] platform, a program can pass all of the generated test data and is not awarded full credit (it is possible to get fewer points for having more lines of code than needed). Therefore, student submissions may be tracked to see how the program strategies change as they work toward getting full credit (i.e., how their submissions move throughout the different clusters). This type of analysis may help an instructor understand how their students are approaching different problems progressing toward the correct answer.

2.2 ILLUSTRATIVE EXAMPLE

In this section, we aim to illustrate why syntax-based clustering approaches fail to cluster programs with similar strategies. In our motivating example, we focus on the second use case: partial credit/penalization. Consider a scenario where the instructor asks students to

```
1  // Student submission C
2  using System;
3  public class Program {
4    static public int Partition(int[] arr, int left, int right) {
5      int pivot;
6      pivot = arr[left];
7      while (true) {
8        while (arr[left] <= pivot)  // IndexOutOfRangeException
9            left++;
10       while (arr[right] >= pivot) // IndexOutOfRangeException
11           right--;
12       if (left < right) {
13           int temp = arr[right];
14           arr[right] = arr[left];
15           arr[left] = temp;
16       } else
17           return right;
18     }
19   }
20
21   static public void quickSort(int[] arr, int left, int right) {
22     int pivot;
23     if (left < right) {
24       pivot = Partition(arr, left, right);
25       if (pivot > 1)
26         quickSort(arr, left, pivot);  // Stack overflow
27       if (pivot < right)              // Stack overflow
28         quickSort(arr, pivot, right); // Stack overflow
29     }
30   }
31   // Overall error: IndexOutOfRangeException
32   static int Puzzle(int[] arr) {
33     quickSort(arr, 0, arr.Length - 1);
34     return arr[arr.Length-1];
35   }
36 }
```

Figure 2.3: Student C's incorrect implementation of the quick sort algorithm.

write a C# method that returns the difference between the largest and smallest element in an input integer array.

Figure 2.4 shows a correct submission from the first student who uses C#'s built-in `Max` and `Min` methods to return the expected result. On the other hand, in Figure 2.5, the second student manually implements her own `Max` and `Min` methods to return the expected result. Lastly, the third student manually implements the bubble sort algorithm (shown in Figure 2.6) to sort the array before returning the expected result.

Limitations of existing approaches. Naive extraction of code transformations through simple text differencing or abstract syntax tree differencing does not work well [11]. When considering all three submissions, we see that the third student employs a strategy that is quite different from that of the first two students. Let us then focus particularly on the programs from the first two students. While both programs are conceptually similar, they differ in variable names and code structure—the first program uses C#'s built-in `Max` and

```
using System;
using System.Linq;
public class Program {
  public static int Puzzle(int[] a) {
    return a.Max() - a.Min();
  }
}
```

Figure 2.4: First student submission showing the use of built-in C# `Array.Max()` and `Array.Min()` methods.

```
using System;
public class Program {
  public static int Puzzle(int[] a) {
    int min = a[0], max = a[0];
    int i = 0, k = 0;
    while((i + 1) != a.Length) {
      if(a[i + 1] > max) max = a[i + 1];
      i++;
    }
    while((k + 1) != a.Length) {
      if(a[k + 1] < min) min = a[k + 1];
      k++;
    }
    return max - min;
  }
}
```

Figure 2.5: Second student submission showing the manually implemented `Array.Max()` and `Array.Min()` methods.

`Min` methods, whereas the second program uses two while-loops to identify the largest and smallest integer elements in the array, respectively. Some syntax-based approaches [9, 10, 12] rely on Abstract Syntax Tree (AST) comparison and do not account for operations such as API calls. As a result, such approaches would not consider these two programs to have the same strategy. Thus, it is important to go up multiple layers of abstraction to cluster based on the semantics of programs, instead of syntax.

Other existing approaches such as CLARA [7] and OverCode [6] place all three student submissions into different clusters because these approaches consider two programs to be matching only when their program features (e.g., control flow, variable sequence, and total bijective relation between variables) are the same. Neither approach clusters based on the *degree* to which submissions are similar. Rather, their clustering is binary (i.e., matching or not matching). Further, CLARA requires that the control flows (i.e., looping structure) of the two programs match for them to be placed in the same cluster.

While the first two student submissions (in Figures 2.4 and 2.5) do not share the same looping structures, both programs share the same strategy. Therefore, the clusters produced by CLARA may not group programs employing the same strategies, but instead by syntactic

```
using System;
public class Program {
  public static int Puzzle(int[] a) {
    if (a.Length > 2) {
      for (int i = 0; i < a.Length; i++)
        for (int j = 0; j < a.Length - 1; j++)
          if (a[j] > a[j + 1]) {
            int temp = a[j];
            a[j] = a[j + 1];
            a[j + 1] = temp;
            }
      int max = a[a.Length-1];
      int min = a[0];
      return max-min;
    }
    else if (a.Length == 2)
      return a[0] > a[1] ? a[0] - a[1] : a[1] - a[0];
    return 0;
  }
}
```

Figure 2.6: Third student submission using bubble sort to first sort array before returning the result.

features. OverCode, on the other hand, first cleans programs by renaming *common variables* (i.e., those with identical sequences across two or more program traces). Then, each line of a given cleaned program is placed into a set, and those that match the set exactly are clustered together. As is the case with CLARA, minor syntactic differences between two programs could cause misclustering [4].

CHAPTER 3: APPROACH

In this section, we introduce our approach to cluster student submissions based on the path conditions generated by using a structural test generator. We first present an overview of our approach. Next, we expose lower-level details of how we implement our approach.

3.1 APPROACH OVERVIEW

Our approach takes as inputs a dataset of correct student submissions and returns as output the equivalence classes containing submissions that employ the same strategy. Our approach consists of two techniques: test generation and path-condition-based clustering. Figure 3.1 shows an overview of our approach, where "PC" denotes path condition and "subs." denotes submissions.

With the correct student submissions as inputs, our test generator returns a mapping of the test data produced by our structural test generator to the corresponding path conditions per student submission as output. Our path-condition equivalence checker then uses this mapping as input to compare each submission pairwise and clusters those submissions with *semantically equivalent* path conditions (we elaborate upon how we determine this semantic equivalence later in the chapter).

3.1.1 Test Generation

Our test generation technique intends to generate test data for exposing different strategies of student submissions that share the same interface (e.g., the same method signature). The key idea of our test generation technique is first to apply a structural test generator such as Pex [20] on each submission to generate test data. The structural test generator aims to achieve high branch coverage of each submission and the methods directly or indirectly invoked by the submission.

If the generated test data for a submission causes the structural test generator to fail in its path exploration (e.g., the default exploration time limit or the maximum number of conditions have been exceeded, thereby generating no path condition), then the generated test data is automatically discarded (i.e., classified as invalid).

After applying the structural test generator on each submission, we say that two correct submissions are *semantically equivalent* for each test data (in the input space) if the following two conditions are satisfied: (1) they share the same set of generated test data and (2) their

Figure 3.1: Overview of our approach.

path conditions corresponding to each test data are *equivalent* (in Section 3.2, we explain how to determine such equivalence).

Structural test generators (including Pex) are not guaranteed to generate the same set of test data when applied on two submissions, respectively. In such cases, satisfying Condition 1 would not be possible. Thus, we need to perform a normalization such that all submissions share the same set of generated test data. To conduct this normalization, we first take the union of all test data generated by our structural test generator for all submissions, respectively, and place the unioned test data into a set to capture only the unique test data (i.e., the set resulting from the union). Then, we configure and invoke Pex once more on each submission, but this time we limit Pex to use only the concrete set of test data (i.e., the set that we union) to produce the path condition for each test data. Now every submission in the given dataset has a mapping of the same set of generated test data to their corresponding path conditions, respectively. Having this mapping enables us to fairly compare the path conditions of two submissions so that we can reason about whether or not these path conditions are semantically equivalent (we detail our implementation of this comparison in Section 3.3).

3.2 PATH-CONDITION-BASED CLUSTERING

The key idea behind the path-condition-based clustering technique is to group the submissions in a given dataset with equivalent path conditions for each generated test data. To conduct this grouping, we first translate the path condition strings produced by Pex to a syntax that is interpretable by theorem provers such as Z3 [23].

We then build a model that checks the equivalence of two parsed path condition strings. A simplified representation of the model can be found in Figure 3.2. For example, these two path conditions `PC1` and `PC2` are logically equivalent for integer x: `PC1: x > 0 && x < 100 && x == 5`, `PC2: x > 0 && x < 100 && x <=5 && x >= 5`.

Using this model to check the logical equivalence of the path conditions results in highly dispersed clusters, in that there are many clusters with few submissions inside. While having

```
1  def simplifiedModel(pc1, pc2):
2      solver = Solver()
3      solver.add(pc1 != pc2)
4      return solver.check().r == -1
5
```

Figure 3.2: Simplified Z3 model to compare parsed path conditions.

highly dispersed clusters is not a bad thing, this result suggests that the path-condition-based clustering technique is quite strict in its definition of strategy. In other words, two submissions employing the same strategy with different design decisions (e.g., using a for-loop versus a do-while loop when calculating Fibonacci sequence iteratively) do not get clustered together when they actually should (providing that both submissions partition the input space and address the problem within each sub-space in the same way).

To alleviate this strictness, we provide a relaxed model, which relaxes the clustering constraints by incorporating *logical implication* into our model. In other words, the underlying predicates do not need to be logically equivalent for two path conditions to be considered equivalent; instead, two path conditions can be considered equivalent if the predicates in one path condition imply the predicates in the other path condition. Informally, logical implication states that "a true premise cannot imply a false conclusion," and is expressed as [24]:

If condition A holds

Then condition B holds

As an example, predicate `x > 5` implies predicate `x >= 5`.

For a given model, we run the model pairwise on every two submissions to determine the submissions that are to be clustered together.

3.3 APPROACH IMPLEMENTATION

To perform our test generation technique, we invoke Pex on each submission, take the union of all the test data generated by Pex, and place them into a set as a form of normalization.

We then re-invoke Pex on every submission in a given dataset, but this time we limit Pex to use only the concrete set of test data (i.e., the set that we union). Some submissions are so complex (i.e., having an indeterminate number of branch conditions) that a certain test data causes Pex to fail to generate the path condition. Our approach automatically discards such test data as invalids. Figure 3.3 features an implementation of the bogo sort algorithm. Bogo sort works by first checking whether the input array is already sorted;

```
using System;
class program {
  // Method for checking array ordering
  static bool IsSorted(int[] a) {
    for (int i = 0; i < a.Length - 1; i++)
      if (a[i] > a[i + 1])
        return false;
    return true;
  }

  // Shuffle array elements
  static int[] RandomPermutation(int[] a) {
    Random random = new Random();
    var n = a.Length;
    while (n > 1) {
      n--;
      var i = random.Next(n + 1);
      var temp = a[i];
      a[i] = a[n];
      a[n] = temp;
    }
    return a;
  }

  // Random sort
  static int[] BogoSort(int[] a) {
    while (!IsSorted(a))
      a = RandomPermutation(a);
    return a;
  }
}
```

Figure 3.3: Implementation of the bogo sort algorithm.

if the elements are not already sorted, then it randomly permutes the elements and once again checks whether the array is sorted. These operations are repeated until the array is sorted [25]. We see here that the number of branch conditions in a single execution path is indeterminate and may exceed Pex's maximum threshold for branch conditions (its default value is 500, but even after we change it to 5000, Pex fails to produce path conditions for the paths when the input is an unsorted array of length three or greater), especially as the input size grows.

After re-invoking Pex on each submission, we parse and translate the resultant path condition strings to SMT-LIB2 syntax so that they are interpretable by the Z3 theorem prover. This translation enables us to build a model to compare each submission's parsed path conditions strings.

CHAPTER 4: EVALUATION

We evaluate the effectiveness of our path-condition-based clustering approach by answering the following two research questions:

- RQ1: How effective is our approach in clustering submissions by strategy while minimizing the number of the produced false positives?
- RQ2: How syntactically different are the submissions clustered by our approach?

The answer to RQ1 shows the effectiveness of our approach in using program semantics to cluster submissions by strategy. Answering RQ2 allows us to investigate the syntactical uniformity of the submissions clustered by our approach.

4.1 EVALUATION SETUP

4.1.1 Datasets

To evaluate our approach, we leverage four datasets: CodeHunt [21], Pex4Fun APCS (Advanced Placement Computer Science) [22], University Assignments (a collection of student submissions from a CS1 course taught at a major university), and Algorithms (a collection of publicly available sorting algorithms implemented in C#).

CodeHunt. The CodeHunt dataset contains around 13,000 student submissions from a 48-hour worldwide coding contest. The contest has four sectors, and each sector contains six puzzles. The contest participants (students) were allowed unlimited attempts to solve the programming problems within 48 hours. A screenshot of the platform is shown in Figure 4.1.

The Code Hunt [26] platform provides feedback based on test generation through the Pex engine to guide players to modify their code. The way that students write code in Code Hunt is quite different from what is seen in traditional software development because there are no known requirements (either informally or formally documented or existing in developers' minds); the game aspect in Code Hunt is essentially re-engineering from sample expected behaviors observed from generated test cases [21].

We choose to include this dataset in our evaluation because the difficulty of its puzzles is representative of homework problems typically assigned in introductory-level programming courses and also because it is used in related work [4, 8, 13]. The puzzles increase in difficulty as the levels and sectors increase; consequently, the number of submissions decreases as the difficulty increases. Of the 24 total puzzles in the CodeHunt dataset, we choose the 12

Figure 4.1: The Code Hunt platform.

puzzles whose solutions include at least one branch condition. Brief descriptions of each selected puzzle are shown in Table 4.1.

We select these puzzles because path conditions track the unique paths that the execution takes to reach an endpoint. Consequently, if there are no branch conditions, then there would exist only one unique path. For those puzzles with no branches, we would see no variance among the path conditions, meaning that our approach would not distinguish the student submissions from each other.

The student submissions are devoid of all information regarding student identity, including anonymous identifiers. The students who participated in the contest were permitted to use either Java or C#, so because we use Pex[1] [20] as our test generator (which is specific to .NET languages such as C#), we take only the C# student submissions. From these submissions, we take only those that are labeled as correct. We focus on the correct submissions because, in general, there are fewer ways for students to get a problem correct than incorrect. Therefore, focusing on this subset of submissions would be easier for our approach to handle during our initial experimentation. Table 4.2 shows the total number of C# and Java submissions for each puzzle, as well as the number of correct submissions per puzzle. There are 4585 C# submissions for the puzzles examined in the CodeHunt dataset, and 362 correct submissions across all puzzles. Subsequently, the ratio of correct to incorrect implementations is 362:4223.

Pex4Fun APCS. We also include student submissions from the Pex4Fun [27] platform (Code Hunt's predecessor). Pex4Fun APCS is an online lab that features 156 coding duels designed for advanced placement computer science students. We include this dataset in our evaluation because it is used in related work [15]. These coding duels are similar in difficulty to homework assignments in introductory-level programming classes. In each coding duel,

[1]We use Pex because of its effectiveness in previous related work [15, 17]. However, alternative structural test generators could also be used.

Table 4.1: Description of selected Code Hunt puzzles.

Puzzle	**Description**
Sector2-Level1	Compute the average of a list of numbers, rounded to closest integer
Sector2-Level2	Count the depth of nesting parentheses in a string
Sector2-Level5	Find the maximum difference between two elements in an array
Sector2-Level6	Generate the string of binary digits for input integer n
Sector3-Level1	Create a filter that retains only values greater than or equal to a given threshold (i.e., a crude noise filter)
Sector3-Level2	Compute the sum of the n-th and (n-1)th Fibonacci numbers
Sector3-Level3	Find the k-th largest element in an array
Sector3-Level6	Compute the set difference of input integer arrays a and b (i.e., find the elements that are unique to a)
Sector4-Level2	Compute n choose m, i.e., $n!/(m! * (n-m)!)$
Sector4-Level3	Given two integer arrays a and f, apply $b[i] = f[a[i]]$ and return b
Sector4-Level4	Return the (first) value with the greatest number of 1's in its binary representation
Sector4-Level6	Advance each character in a string by the Fibonacci number evaluated at that character's integer ASCII value

Table 4.2: Statistics of correct/total C# and Java submissions in the CodeHunt dataset.

Puzzle	**Correct C#**	**Total C#**	**Correct Java**	**Total Java**
Sector2-Level1	42	1495	15	1374
Sector2-Level2	48	767	79	739
Sector2-Level5	44	287	55	247
Sector2-Level6	55	249	48	334
Sector3-Level1	15	102	22	156
Sector3-Level2	48	259	51	207
Sector3-Level3	13	255	24	255
Sector3-Level6	32	178	18	229
Sector4-Level2	10	128	7	126
Sector4-Level3	4	72	5	125
Sector4-Level4	8	732	12	427
Sector4-Level6	10	61	15	163

```
using System;
public class Program {
  public static string Puzzle(int i, int j) {
    if(j == int.MinValue)
      return “false”;
    if(j == int.MaxValue)
      return “false”;

    int delta = i - j;
    if(int.MaxValue+i < j)
      return “false”;
    bool isOneAway = Math.Abs(delta) == 1;
    return isOneAway? “true” : “false”;
  }
}
```

Figure 4.2: Example student submission for guessing-and-checking.

the student has to implement a program against a specification. At each attempt, Pex4Fun analyzes the student program and the specification program to produce an input-output table showing the cases where the behavior of the two programs matches and diverges, respectively. Based on this feedback, the student can iteratively refine her solution until it has the same observable behavior as the specification program [15, 28]. While the overarching Pex4Fun platform is compatible with .NET languages (C#, Visual Basic, and F#), the Pex4Fun APCS initiative supports only C#.

Of the 156 coding duels, we select only those whose submissions and solutions include at least one branch condition; we identify 61 such puzzles. Of these 61 puzzles, we select those with at least two students who answered them correctly (so that we can compare different students' submissions), leaving us with 45 puzzles included in our evaluation. The Pex4Fun APCS platform features some coding duels with clear problem statements and other coding duels with unclear or no problem statements, where the students need to guess-and-check until they reach the behavior of the underlying solution. Given that our approach aims to distinguish strategies of student submissions in the classroom setting, we remove those puzzles where the student submissions perform blatant guessing-and-checking (compare a student's submission performing guessing-and-checking in Figure 4.2 to the underlying solution in Figure 4.3), as this behavior is generally not representative of coding submissions in the classroom. After removing such puzzles, we are finally left with 40 puzzles to include in the evaluation.

Then, from these 40 puzzles, we evaluate only the correct student submissions. Table 4.3 shows the number of correct submissions, along with the total number of submissions for each puzzle in the evaluation. Across all puzzles, there are 184 correct submissions and 1083 total submissions. The ratio of correct to incorrect submissions is 184:899.

Table 4.3: Statistics of correct/total submissions in the Pex4Fun APCS dataset.

Puzzle	**# Correct Subs.**	**# Total Subs.**
Puzzle 33	13	38
Puzzle 34	10	115
Puzzle 35	7	40
Puzzle 36	4	41
Puzzle 38	7	88
Puzzle 40	6	34
Puzzle 42	6	47
Puzzle 43	5	13
Puzzle 45	4	17
Puzzle 47	2	16
Puzzle 48	2	16
Puzzle 49	5	37
Puzzle 50	6	22
Puzzle 53	6	14
Puzzle 55	7	22
Puzzle 57	6	16
Puzzle 58	6	18
Puzzle 59	6	13
Puzzle 60	6	9
Puzzle 61	5	22
Puzzle 62	4	74
Puzzle 63	5	25
Puzzle 64	5	29
Puzzle 65	4	15
Puzzle 73	3	17
Puzzle 74	5	18
Puzzle 75	3	24
Puzzle 83	2	4
Puzzle 105	3	8
Puzzle 107	2	25
Puzzle 110	2	25
Puzzle 111	2	14
Puzzle 112	3	15
Puzzle 132	6	15
Puzzle 135	4	37
Puzzle 140	3	28
Puzzle 141	2	11
Puzzle 143	2	4
Puzzle 144	3	51
Puzzle 152	2	6

```
using System;
public class Program {
  public static string Puzzle(int i, int j) {
    long i2 = i, j2 = j;
    return Math.Abs(i2 - j2) == 1 ? "true" : "false";
  }
}
```

Figure 4.3: Underlying solution that returns true if the absolute difference between two input integers is one and false otherwise.

Table 4.4: Statistics of correct/total submissions in the University Assignments dataset.

Assignment	**Correct Subs.**	**Correctly Conv. Correct Subs.**	**Total Subs.**
Homework 1	118	63	486
Homework 2	321	210	1422
Homework 3	631	527	3406

University Assignments. The third dataset in our evaluation includes student submissions from a CS1 course taught at a major university. This dataset features three homework problems, three corresponding homework solutions, and over 5000 student submissions. All of the homework solutions and student submissions contain more than one branch condition, meaning that all three of the homework problems in this dataset are eligible to be included in our evaluation. The student submissions are all written in C, so to make them compatible with our C#-specific test generator, Pex, we convert them with Tangible Software Solutions' C/C++ to C# Converter [29], and again take only the correct submissions. Following the conversion, we attempt to compile the correct submissions, keeping only those that successfully compile (i.e., those submissions that are successfully converted from C to C#). Table 4.4 shows the number of correct submissions, correct submissions that are successfully converted from C to C#, and total submissions per homework problem. There are 800 correct answers (from a total of 5314 submissions) that are successfully converted to C#, yielding a ratio of 800:4514 for correct and compilable submissions to incorrect submissions.

Algorithms. Our final dataset in the evaluation includes sorting algorithms written in C#. We select sorting algorithms because (1) they are typically taught in Computer Science curricula, (2) we want to "stress test" our approach by giving it programs that are more complex than those at the introductory level, and (3) we want to see whether our approach produces strategy-based equivalence classes that align with different sorting algorithm types.

We visit RosettaCode [30], a programming chrestomathy site containing solutions to tasks in a myriad of languages, to find a comprehensive list of sorting algorithms implemented in

```
// Submission A
using System;
using System.Linq;
public class Program {
  public static int[] Sort(int[] a) {
    for (int i = 1; i < a.Length; i++) {
      int target = a[i];
      var moveIndex = i;
      int targetInsertLocation = BinarySearch(a, 0, moveIndex, target);
      var b = a.Skip(targetInsertLocation).Take(i - targetInsertLocation).ToArray();
      for (int x = 0; x < b.Length; x++)
        a[targetInsertLocation + 1 + x] = b[x];
      a[targetInsertLocation] = target;
    }
    return a;
  }
  // Manually implemented binary search
  private static int BinarySearch(int[] a, int from, int to, int target) {
    var left = from;
    var right = to;
    while (right > left) {
      var middle = (left + right) / 2;
      if (a[middle] == target)
        return middle + 1;
      if (a[middle] < target)
        left = middle + 1;
      else
        right = middle - 1;
    }
    return a[left] < target ? left + 1 : left;
  }
}
```

Figure 4.4: Binary insertion sort with binary search method implemented manually.

C#. RosettaCode has only one C# implementation per sorting algorithm, so we leverage other sources [31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46] to collect more C# implementations. We identify these sources by first searching each algorithm on Google to find C# implementations whose structures (i.e., use of loops, if-statements, data types) are different from those gathered from RosettaCode. Figures 4.4 and 4.5 show an example of an algorithm's two implementations that are different from each other, where Submission A implements a binary search method manually and Submission B, instead, invokes C#'s built-in `Array.BinarySearch()` method.

If we cannot locate more than five different C# implementations within the first three pages of Google search results for a given algorithm, then we do not include that algorithm in our evaluation. We gather six different implementations for each of the five selected algorithms, yielding 30 C# programs. Apart from ensuring that the gathered implementations compile successfully, we also construct a test wrapper[2] (as simplified in Figure 4.6) to ensure that each implementation produces as output the expected sorted array when given an

[2]A test wrapper is a method whose primary purpose is to call a second method with little or no additional computation.

```
// Submission B
using System;
using System.Linq;
public class Program {
  public static int[] Sort(int[] a) {
    for (int i = 1; i < a.Length; i++) {
      int x = a[i];
      // Built-in binary search
      int j = Math.Abs(Array.BinarySearch(a, 0, i, x) + 1);
      var b = a.Skip(j).Take(i - j).ToArray();
      for (int y = 0; y < b.Length; y++)
        a[j + 1 + y] = b[y];
      a[j] = x;
    }
    return a;
  }
}

```

Figure 4.5: Binary insertion sort using C#'s built-in binary search method.

```
public static void Test(arr, sorted_arr) {
  assert(IsEqual(implementation(array), sorted_arr));
}

```

Figure 4.6: Simplified test wrapper constructed for checking implementation output.

array as input. For ease of experimentation, we choose to use integer arrays as inputs in this dataset.

4.1.2 Research Questions

To answer RQ1, we first invoke our structural test generator (i.e., Pex) on each student submission in a given dataset to generate test data. Afterward, we take the union of the generated test data for all student submissions and place the unioned test data into a set so that we capture only the unique ones. Then, for every submission in the dataset, we invoke Pex again on the student submissions, this time limiting Pex to use only the set of unionized test data as concrete inputs. We limit Pex in this way as a form of normalization such that for each submission, we generate path conditions from the same set of test data. We then parse the path conditions and perform a pairwise comparison between each submission's path conditions per concrete test data. We then group into equivalence classes those submissions whose path conditions are determined to be equivalent (for every seeded input data) according to the Z3 theorem prover. We then use the percentage of false positives per equivalence class to measure our approach's effectiveness. Lastly, we provide insights as to why our approach fails to reduce these false positives.

To answer RQ2, we run two syntax-based clustering approaches—JetBrains' dupFinder (DF) [47] and Microsoft's Near-Duplicate Code Detector (ND) [48]—on the submissions from each cluster (i.e., equivalence class) produced by our approach. Note that our approach places two submissions into the same equivalence classes only if, for each input data in the input space, their path conditions are equivalent.

One thing to note is that our approach is not needed to split the student submissions into one group of correct submissions and the other group of incorrect submissions. Instead, one could simply compare the answer of each student submission, pairwise, to the instructor solution and split the submissions into two groups: correct and incorrect. Our approach becomes useful in going beyond simply splitting the submissions into correct and incorrect groups—each group likely contains submissions that employ different strategies from each other. In our approach, we want to adopt the instructor's point of view and ask, "how many different strategies did the students who answered correctly use?"

4.2 EVALUATION RESULTS

4.2.1 RQ1 - Effectiveness of our Clustering Approach

To reason about our approach's effectiveness, we first need to construct the ground truth by manually examining each submission across each dataset and assigning explicit and consistent labels to these submissions based on the employed strategy. We assign the same label to those submissions in a given equivalence class with the same observed strategy. Consistent labeling is crucial for allowing us to articulate *why* our approach clusters certain submissions together. We call this labeling our taxonomy. Using our taxonomy, we can identify a submission that our approach places into the incorrect equivalence class (i.e., manual inspection reveals that the submission's strategy is better aligned with the submissions of another equivalence class) as a false positive. We can also use our taxonomy to identify an incorrect submission, i.e., a submission that is not correct and whose faults go undetected by the test oracle used in a given dataset. An example of such an incorrect submission for this puzzle is shown in Figure 4.7. In this example, when we supply the integer array `[1, 2]` as input, then the expected result is 2, but this code returns 3.

A snippet of the taxonomy for the Code Hunt Sector2-Level5 puzzle is shown in Table 4.5, and a full version of the taxonomy can be found on our project website[3]. "EC No." denotes the equivalence class number (i.e., the first, second, third, and fourth equivalence classes

[3]https://joseio.github.io/CodeSimilarity-v2/#taxonomy

```
using System;
using System.Linq;
public class Program {
  public static int Puzzle(int[] a) {
    if(a.Length == 1) return a[0];
    var avg = a.Average(); // Avg. the array
    if(avg <= 0) return 0;
    if(avg%1.0 == 0.5)
      return (int)Math.Round(avg) + 1; // Round
    return (int)Math.Round(avg);
  }
}
```

Figure 4.7: Student submission with undetected incorrect behavior.

produced by our approach).

Table 4.5: Snippet of taxonomy for the Code Hunt Sector2-Level5 puzzle.

EC No.	**Label(s)**
0	2x: Use `Sort()`, then iterates to find the biggest difference
1	19x: Use C built-in `Min` and `Max` methods 11x: Implement their own `Min` and `Max` methods or use for-loop to find max and min array elements
2	10x: Use `Sort` method, then return $arr[arr.Length - 1] - arr[0]$
3	2x: Implement bubble sort, then return $arr[arr.Length - 1] - arr[0]$

Table 4.6 shows the effectiveness of our approach by highlighting only the puzzles where false positives and incorrect submissions are found. In the table, "# Subs." denotes the total number of submissions evaluated in a given dataset, "UA" denotes the University Assignments dataset, and "# FPs" and "# Incorrect" denote the number of false positives and incorrect submissions found, respectively. Of the 56 total programming assignments under evaluation, we find false positives and incorrect submissions in only the 12 assignments listed in the table. After running our approach on the 42 total programming submissions of Code Hunt's Sector2-Level1 puzzle, we see that our approach yields no false positives (i.e., we identify no submission whose strategy is better aligned with the submissions of another cluster) and 18 incorrect submissions (i.e., those submissions that are not correct and whose faults go undetected by Code Hunt's test oracles).

We also see that our approach yields five false positives and finds one incorrect submission for Homework 3 in the University Assignments dataset. For this homework problem, two clusters are produced, one having 204 submissions and the other having 6 submissions. In cluster two, 5 of the submissions there are classified as false positives because they use the same strategy as those submissions from cluster one but are not placed into cluster one.

```
using System;
public class GlobalMembers{
  public static int Main(int input) {
    // Yields PC: input < 1.5 && input >= 0 && input >= 1
    int n, k = input;
    double sn = 0;
    for (n = 1;sn <= k;n++) // Conditional statement
      sn = sn + 1.0 / n;
    n = n - 1;
    return n;
  }
}
```

Figure 4.8: Student submission in cluster 0 with for-loop construction.

```
using System;
public class GlobalMembers{
  public static int Main(int input) {
    // Yields PC: input < 1.5
    int n = 1, k = input;
    double sn = 1.0;
    do {
      n++;
      sn = sn + (1.0 / n);
    } while (sn <= k); // Conditional statement
    return n;
  }
}
```

Figure 4.9: Student submission in cluster 1 with do-while-loop construction.

We see that the path-condition-based clustering approach fails because of the placement of statements within the different types of loops utilized (e.g., for, while, do-while loops). In Figures 4.8 and 4.9, we see two such submissions that employ the same strategy but are placed into different clusters because of the placement of the conditional statements. More concretely, the value of both programs' variable `sn` is different at the location of their conditional statements.

The one incorrect submission in cluster two goes unnoticed by the test generator, as it does not produce the expected answer when the input is input integer 1.

Overall, Table 4.6 shows that of the 1324 total submissions under evaluation, we find 19 false positives and 26 incorrect submissions. In other words, we find that our approach produces false-positive results on approximately 1.4% of the submissions and detects incorrect submissions among approximately 1.9% of the submissions. These results illustrate that our approach is effective at grouping student submissions by strategy.

Table 4.6: Effectiveness of the CodeSimilarity approach on four datasets.

Dataset	**Assignment**	**# Subs.**	**# FPs**	**# Incorrect**
Code Hunt	Sector2-Level1	42	0	18
Code Hunt	Sector2-Level2	48	1	0
Code Hunt	Sector3-Level6	32	2	5
Code Hunt	Sector4-Level2	10	4	0
Code Hunt	Sector4-Level6	10	1	0
Pex4Fun	Puzzle 34	10	0	1
Pex4Fun	Puzzle 65	4	1	0
Pex4Fun	Puzzle 74	5	0	1
Pex4Fun	Puzzle 110	2	1	0
Algorithms	Heap Sort	6	3	0
Algorithms	Merge Sort	6	1	0
UA	Homework 2	210	5	1

4.2.2 RQ2 - Syntactic Difference of Submissions Clustered by our Approach

Table 4.7 shows the number of equivalence classes produced by our approach (denoted as "CS") for each sorting algorithm, as well as the number of code clones that the DF and ND approaches identify within the equivalences classes (denoted as "EC"). Looking at the results from the Code Hunt Sector2-Level5 puzzle, we see that our approach produces three equivalence classes: the first containing 2 submissions, the second containing 30, and the third containing 10. In the first equivalence class, we see that both DF and ND agree and report 2 syntactic code clones. For the second equivalence class, DF produces two clusters, each containing 2 syntactic clones, while ND produces three clusters of 17, 6, and 4 submissions, respectively. For the third equivalence class, DF produces two clusters containing 2 and 8 submissions, whereas ND agrees with our approach's results, producing one cluster of 10 submissions.

The results from evaluating the Algorithms dataset are shown in Table 4.8. From this table, we see that the DF approach finds no syntactic code clones in any equivalence class, whereas the ND approach does. For the bubble sort algorithm, our approach produces one equivalence class containing all 6 evaluated submissions, whereas the ND approach identifies one cluster of 4 syntactic clones. These results demonstrate a close relationship between program syntax and strategy (i.e., program semantics). This relationship suggests that there are not many different ways (different in terms of program syntax and semantics) that the bubble sort algorithm can be implemented.

In another example, we see that for the merge sort algorithm, our approach produces

Table 4.7: Comparison of different clustering approaches on the CodeHunt dataset.

Puzzle	Approach	EC 1	EC 2	EC 3	EC 4	EC 5	EC 6	EC 7	EC 8	EC 9	EC 10	EC 11
Sector2-Level1	CS	1	14	6	2	3	10	1	2	1	1	1
	DF	0	2|3	2	2	2	2	2	2	2	0	0
	ND	0	4|2x2	4|2	2	2	10	0	2	0	0	0
Sector2-Level2	CS	33	11	0	0	0	0	0	0	0	0	0
	DF	4x2	2|2	0	0	0	0	0	0	0	0	0
	ND	12|4|4x2	3x2	0	0	0	0	0	0	0	0	0
Sector2-Level5	CS	2	30	10	2	0	0	0	0	0	0	0
	DF	2	2|2	2|8	0	0	0	0	0	0	0	0
	ND	2	17|6|4	10	0	0	0	0	0	0	0	0
Sector2-Level6	CS	55	0	0	0	0	0	0	0	0	0	0
	DF	3|2|3	0	0	0	0	0	0	0	0	0	0
	ND	36|4|2x3|2x2	2	0	0	0	0	0	0	0	0	0
Sector3-Level1	CS	14	1	0	0	0	0	0	0	0	0	0
	DF	2|3|2	0	0	0	0	0	0	0	0	0	0
	ND	3|11	0	0	0	0	0	0	0	0	0	0
Sector3-Level2	CS	48	0	0	0	0	0	0	0	0	0	0
	DF	2x3|5x2	0	0	0	0	0	0	0	0	0	0
	ND	9|8|6|5|3|3x2	0	0	0	0	0	0	0	0	0	0
Sector3-Level6	CS	13	13	3	1	1	1	0	0	0	0	0
	DF	3	3x2	2	0	0	0	0	0	0	0	0
	ND	7|4|2	5|2x2	2	0	0	0	0	0	0	0	0
Sector4-Level2	CS	4	1	3	1	1	0	0	0	0	0	0
	DF	0	0	3	0	0	0	0	0	0	0	0
	ND	3	0	0	3	0	0	0	0	0	0	0
Sector4-Level3	CS	4	0	0	0	0	0	0	0	0	0	0
	DF	0	0	0	0	0	0	0	0	0	0	0
	ND	3	0	0	0	0	0	0	0	0	0	0
Sector4-Level4	CS	6	1	1	0	0	0	0	0	0	0	0
	DF	5x2	0	0	0	0	0	0	0	0	0	0
	ND	6	0	0	0	0	0	0	0	0	0	0
Sector4-Level6	CS	9	1	0	0	0	0	0	0	0	0	0
	DF	2	0	0	0	0	0	0	0	0	0	0
	ND	3	0	0	0	0	0	0	0	0	0	0

three equivalence classes: the first containing 4 submissions, the second containing 1 submission, and the third also containing 1 submission. On the other hand, neither of the syntactic approaches produce any clusters within these equivalence classes. These results show that although the Merge Sort algorithm can be written in syntactically different ways, the underlying strategy may be the same.

Table 4.9 shows that in Puzzle 33 of the Pex4Fun APCS dataset, our approach produces two equivalence classes—the first containing 12 submissions and the second containing 1 submission. In the first equivalence class, DF yields two clusters, both having 2 submissions each. ND produces three clusters: the first with 5 submissions, the second with 2 submissions, and the third with 4 submissions. Because our approach places only 1 submission in the second equivalence class, there are no syntactic clones found. Interestingly, it may be that the clusters produced by the syntactic approaches in all three datasets may correlate

Table 4.8: Comparison of different clustering approaches on the Algorithms dataset.

Algorithm	**Approach**	**EC 1**	**EC 2**	**EC 3**	**EC 4**	**EC 5**	**EC 6**
Bubble Sort	CS	6	0	0	0	0	0
	DF	0	0	0	0	0	0
	ND	4	0	0	0	0	0
Heap Sort	CS	1	1	1	1	1	1
	DF	0	0	0	0	0	0
	ND	0	0	0	0	0	0
Insertion Sort	CS	5	1	0	0	0	0
	DF	0	0	0	0	0	0
	ND	4	0	0	0	0	0
Merge Sort	CS	4	1	1	0	0	0
	DF	0	0	0	0	0	0
	ND	0	0	0	0	0	0
Quick Sort	CS	2	1	1	1	1	0
	DF	0	0	0	0	0	0
	ND	0	0	0	0	0	0

with the labels that we manually derive in the taxonomy.

In Table 4.10, we see the number of equivalence classes produced by our approach and the clusters produced by DF and ND. For Homework 1, we find that our approach produces two equivalence classes with the first containing 62 submissions and the second containing 1 submission. For the first equivalence class, both DF and ND find six clusters, each containing 2 syntactic clones. No syntactic clones are found in the second equivalence class.

Table 4.9: Comparison of different clustering approaches on the Pex4Fun APCS dataset.

Homework	Approach	EC 1	EC 2	EC 3	EC 4	Homework	Approach	EC 1	EC 2	EC 3	EC 4
	CS	12	1	0	0	⋮	⋮	⋮	⋮	⋮	⋮
Puzzle 33	DF	2|2	0	0	0		CS	4	0	0	0
	ND	5|2|4	0	0	0	Puzzle 62	DF	0	0	0	0
	CS	10	0	0	0		ND	2	0	0	0
Puzzle 34	DF	2|2	0	0	0		CS	5	0	0	0
	ND	9	0	0	0	Puzzle 63	DF	2|2	0	0	0
	CS	7	0	0	0		ND	3|2	0	0	0
Puzzle 35	DF	2|2	0	0	0		CS	1	2	1	1
	ND	7	0	0	0	Puzzle 64	DF	0	0	0	0
	CS	2	2	0	0		ND	0	0	0	0
Puzzle 36	DF	0	0	0	0		CS	4	0	0	0
	ND	0	2	0	0	Puzzle 65	DF	2|2	0	0	0
	CS	7	0	0	0		ND	2	0	0	0
Puzzle 38	DF	0	0	0	0		CS	3	0	0	0
	ND	4	0	0	0	Puzzle 73	DF	0	0	0	0
	CS	6	0	0	0		ND	2	0	0	0
Puzzle 40	DF	2|2	0	0	0		CS	5	0	0	0
	ND	2|2	0	0	0	Puzzle 74	DF	0	0	0	0
	CS	5	1	0	0		ND	4	0	0	0
Puzzle 42	DF	0	0	0	0		CS	3	0	0	0
	ND	2	0	0	0	Puzzle 75	DF	0	0	0	0
	CS	5	0	0	0		ND	2	0	0	0
Puzzle 43	DF	0	0	0	0		CS	0	0	0	0
	ND	3	0	0	0	Puzzle 83	DF	0	0	0	0
	CS	4	0	0	0		ND	0	0	0	0
Puzzle 45	DF	0	0	0	0		CS	3	0	0	0
	ND	2|2	0	0	0	Puzzle 105	DF	0	0	0	0
	CS	2	0	0	0		ND	0	0	0	0
Puzzle 47	DF	0	0	0	0		CS	2	0	0	0
	ND	0	0	0	0	Puzzle 107	DF	0	0	0	0
	CS	2	0	0	0		ND	0	0	0	0
Puzzle 48	DF	0	0	0	0		CS	2	0	0	0
	ND	0	0	0	0	Puzzle 110	DF	0	0	0	0
	CS	5	0	0	0		ND	0	0	0	0
Puzzle 49	DF	0	0	0	0		CS	2	0	0	0
	ND	3	0	0	0	Puzzle 111	DF	0	0	0	0
	CS	6	0	0	0		ND	2	0	0	0
Puzzle 50	DF	5|5	0	0	0		CS	3	0	0	0
	ND	6	0	0	0	Puzzle 112	DF	0	0	0	0
	CS	6	0	0	0		ND	0	0	0	0
Puzzle 53	DF	0	0	0	0		CS	6	0	0	0
	ND	6	0	0	0	Puzzle 132	DF	0	0	0	0
	CS	7	0	0	0		ND	4	0	0	0
Puzzle 55	DF	0	0	0	0		CS	3	1	0	0
	ND	6	0	0	0	Puzzle 135	DF	0	0	0	0
	CS	5	1	0	0		ND	0	0	0	0
Puzzle 57	DF	0	0	0	0		CS	3	0	0	0
	ND	3	0	0	0	Puzzle 140	DF	0	0	0	0
	CS	6	0	0	0		ND	2	0	0	0
Puzzle 58	DF	0	0	0	0		CS	0	0	0	0
	ND	5	0	0	0	Puzzle 141	DF	0	0	0	0
	CS	6	0	0	0		ND	0	0	0	0
Puzzle 59	DF	0	0	0	0		CS	0	0	0	0
	ND	5	0	0	0	Puzzle 143	DF	0	0	0	0
	CS	6	0	0	0		ND	0	0	0	0
Puzzle 60	DF	0	0	0	0		CS	3	0	0	0
	ND	5	0	0	0	Puzzle 144	DF	2|2	0	0	0
	CS	4	1	0	0		ND	3	0	0	0
Puzzle 61	DF	0	0	0	0		CS	1	1	0	0
	ND	3	0	0	0	Puzzle 152	DF	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮		ND	0	0	0	0

Table 4.10: Comparison of different clustering approaches on the University Assignments dataset.

Assignment	**Approach**	**EC 1**	**EC 2**
Homework 1	CS	62	1
	DF	6x2	0
	ND	6x2	0
Homework 2	CS	204	6
	DF	21|5|4|19x2	0
	ND	55|50|23|17|8|2x7|3x3|6x2	2|3
Homework 3	CS	517	9
	DF	2x7|6|5|2x4|14x3|69x2	2
	ND	151|115|28|23|21|20|19|18|11|7|6|3x5|4x4|3|9x2	2x3

CHAPTER 5: 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 [6, 11, 12, 13, 14, 15, 16, 49, 50] exists to cluster programming assignments based on program analysis effectively. Next, we discuss several closely related approaches.

5.1 SYNTAX-BASED CLUSTERING

OverCode. OverCode [6] is a multi-stage approach of information visualization for assessing variations in correct student submissions. It employs dynamic and static analyses to cluster correct student submissions and applies visualization principles to show syntactic similarities and differences among the clusters. To conduct clustering, OverCode first uses dynamic analysis to produce *cleaned code*. It does so by dynamically renaming 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.

MistakeBrowser. MistakeBrowser [11] 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 (i.e., 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, our approach does not require pairs of incorrect and correct submissions, but only requires a single correct solution (which can be an instructor-provided solution) and correct submissions to produce clusters.

TipsC. Similar to MistakeBrowser, TipsC [9] is an approach that enables the provision of feedback to student submissions at scale. It normalizes C program Abstract Syntax Trees (ASTs) and clusters "similar" programs according to the Levenshtein distance between them.

SolMiner. SolMiner first leverages data mining techniques to extract distinct solutions from student code submissions (i.e., sequences of ASTs representing each submission) and then uses unsupervised machine learning techniques to automatically group those programs that implement the same solution.

5.2 SEMANTIC-BASED CLUSTERING

SEMCLUSTER. SEMCLUSTER [4] is an approach that clusters small imperative pro-

grams based on their algorithmic essence (i.e., strategy). SEMCLUSTER captures this algorithmic essence as a vector representation of programs composed of two semantic features: control flow and data flow features. Given a program and a test suite, SEMCLUSTER uses the control flow features to track the volume of inputs flowing through different control flow paths in the program, and the data flow features to track the number of times that a specific value in memory is changed to another specific value.

QLOSE. QLOSE [8] is an approach (built on top of the SKETCH synthesis system [51]) that aims to use the instructor reference solution to repair incorrect student submissions based on *program distances* (where distances are measured by the differences in programs' syntax and semantics).

CLARA. Similar to OverCode, CLARA [7] 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 a *total bijective relation* between the variables of both programs exists. Two variables are said to be in such a relationship if they take the same values in the same order during the programs' executions on the same input data for all input data in a given test suite. CLARA's clustering technique imposes even stronger requirements. Program variables must have the same values in the same order across all program executions.

Grasa. The main objective of Grasa [17] is to augment a given instructor test suite with a minimal set of generated tests that aim to detect a maximum number of incorrect submissions. To accomplish this objective, Grasa first detects and clusters incorrect submissions by approximating their behavioral equivalence to each other. To augment the existing instructor test suite, Grasa generates a minimal set of additional tests to detect the incorrect submissions.

CHAPTER 6: CONCLUSION AND FUTURE WORK

In this thesis, we have proposed CodeSimilarity, a new approach for clustering student submissions for introductory-level programming assignments based on semantic program features, i.e., path conditions. We have evaluated our approach on four datasets and created a taxonomy for enabling us to reason about the effectiveness of our approach. Our evaluation results show that CodeSimilarity is effective at clustering student submissions by strategy, generating few equivalence classes, and identifying incorrect submissions. Future work includes improving upon three main limitations of our current work:

- **Unsoundness.** Our approach is unsound, as Pex has trouble generating path conditions for some methods (e.g., some methods defined on the StringBuilder class and the ToCharArray method defined in the String class). This limitation yields false positives in our results because Pex (1) loses the mapping of symbolic variables and (2) concretizes return variables. We create function hooks that map the problematic methods to lambda functions that are interpretable by Z3 in an attempt to overcome Pex's limitation. In future work, we plan to consider either (1) modifying Pex to maintain the mapping of symbolic variables so that they are not concretized or (2) use a different test generator that is as powerful as Pex and maintains tracking of symbolic variables.

- **Human error.** As our taxonomy is developed by manually inspecting each submission to see whether it belongs to the equivalence class that our approach places it in, human error may exist in our labeling of false-positive results. In future work, we plan to mitigate such human error by using a tool that automatically produces labels based on the programs' syntactic or semantic features.

- **Clustering incorrect student submissions.** Our current work focuses on the correct submissions because, in general, there are fewer ways for students to get a problem correct than incorrect. While focusing on this subset of submissions makes it easier for our approach to handle during our initial experimentation, we realize that providing feedback to incorrect submissions is more valuable (in terms of the students' learning experience). In future work, we plan to extend our approach to cluster incorrect student submissions.

REFERENCES

[1] T. Camp, W. R. Adrion, B. Bizot, S. Davidson, M. Hall, S. Hambrusch, E. Walker, and S. Zweben, "Generation CS: The growth of computer science," *ACM Inroads*, vol. 8, no. 2, pp. 44–50, 2017.

[2] J. M. Allen, F. Vahid, A. Edgcomb, K. Downey, and K. Miller, "An analysis of using many small programs in CS1," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2019, pp. 585–591.

[3] 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., 2010.

[4] M. D. Perry, D. Kim, R. Samanta, and X. Zhang, "SemCluster: Clustering of imperative programming assignments based on quantitative semantic features," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019, pp. 860–873.

[5] A. Drummond, Y. Lu, S. Chaudhuri, C. Jermaine, J. Warren, and S. Rixner, "Learning to grade student programs in a massive open online course," in *Proceedings of the 2014 IEEE International Conference on Data Mining (ICDM)*, 2014, pp. 785–790.

[6] 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 Transactions on Computer-Human Interaction*, vol. 22, no. 2, pp. 7:1–7:35, 2015.

[7] 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 (PLDI)*, 2018, pp. 465–480.

[8] L. D'Antoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantitative objectives," in *Proceedings of the 27th International Conference on Computer-Aided Verification (CAV)*, 2016, pp. 383–401.

[9] S. Sharma, P. Agarwal, P. Mor, and A. Karkare, "TipsC: Tips and corrections for programming MOOCs," in *Proceedings of the 19th International Conference on Artificial Intelligence in Education (AIED)*, 2018, pp. 322–326.

[10] L. Luo and Q. Zeng, "SolMiner: Mining distinct solutions in programs," in *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE)*, 2016, pp. 481–490.

[11] 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 ACM Conference on Learning @ Scale (L@S)*, 2017, pp. 89–98.

[12] J. Huang, C. Piech, A. Nguyen, and L. Guibas, "Syntactic and functional variability of a million code submissions in a machine learning MOOC," *Workshop on Massive Open Online Courses at the 16th Annual Conference on Artificial Intelligence in Education (AIED)*, pp. 25–32, 2013.

[13] 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 (CHESE)*, 2016, pp. 1–6.

[14] V. J. Marin and C. R. Rivero, "Clustering recurrent and semantically cohesive program statements in introductory programming assignments," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM)*, 2019, pp. 911–920.

[15] S. Li, X. Xiao, B. Bassett, T. Xie, and N. Tillmann, "Measuring code behavioral similarity for programming and software engineering education," in *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE)*, 2016, pp. 501–510.

[16] D. Joyner, R. Arrison, M. Ruksana, E. Salguero, Z. Wang, B. Wellington, and K. Yin, "From clusters to content: Using code clustering for course improvement," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2019, pp. 780–786.

[17] J. Osei-Owusu, A. Astorga, L. Butler, T. Xie, and G. Challen, "Grading-based test suite augmentation," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 226–229.

[18] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[19] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, May 2018.

[20] N. Tillmann and P. de Halleux, "Pex- white box test generation for .NET," in *Proceedings of the 2nd International Conference on Test and Proofs (TAP)*, 2008, pp. 134–153.

[21] N. Tillmann, J. de Halleux, J. Bishop, T. Xie, R. N. Horspool, and D. Perelman, "Code Hunt: Context-driven interactive gaming for learning programming and software engineering," in *Proceedings of the 1st International Workshop on Context in Software Development (CSD)*, 2014.

[22] N. Tillmann, J. de Halleux, T. Xie, S. Gulwani, and J. Bishop, "Teaching and learning programming and software engineering via interactive gaming," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 1117–1126.

[23] M. Research, "The Z3 theorem prover." [Online]. Available: https://github.com/Z3Prover/z3

[24] J. Celko, *Joe Celko's SQL for Smarties: Advanced SQL Programming*, 5th ed. Morgan Kaufmann Publishers Inc., 2014.

[25] "Random sort." [Online]. Available: https://programm.top/en/c-sharp/algorithm/array-sort/bogosort/

[26] J. Bishop, R. N. Horspool, T. Xie, N. Tillmann, and J. de Halleux, "Code Hunt: Experience with coding contests at scale," in *Proceedings of 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, vol. 2, 2015, pp. 398–407.

[27] N. Tillmann, J. D. Halleux, T. Xie, S. Gulwani, and J. Bishop, "Teaching and learning programming and software engineering via interactive gaming," in *Proceedings of 35th International Conference on Software Engineering (ICSE 2013), Software Engineering Education (SEE)*, 2013, pp. 1117–1126.

[28] A. Thompsoh, "Pex4fun, an online lab for apcs." [Online]. Available: https://blogs.msdn.microsoft.com/alfredth/2012/03/16/pex4fun-an-online-lab-for-apcs/

[29] "C++ to C# converter." [Online]. Available: https://www.tangiblesoftwaresolutions.com/product_details/cplusplus_to_csharp_converter_details.html

[30] R. Code, "Category: Sorting algorithms." [Online]. Available: http://www.rosettacode.org/wiki/Category:Sorting_Algorithms

[31] "Bubble sort in C#." [Online]. Available: https://codereview.stackexchange.com/questions/142720/bubble-sort-in-c

[32] "C# program to perform bubble sort." [Online]. Available: https://www.csharpstar.com/csharp-program-to-perform-bubble-sort/

[33] "Bubble sort." [Online]. Available: https://www.geeksforgeeks.org/bubble-sort/

[34] A. Donkov, "Data structures and algorithms in c." [Online]. Available: https://github.com/abdonkov/DSA

[35] "Bubble sort." [Online]. Available: https://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Bubble_sort#C#

[36] M. A. Weiss, "Sort." [Online]. Available: http://users.cis.fiu.edu/~weiss/cs/Sort.cs

[37] A. Alhour, "C# algorithms." [Online]. Available: https://github.com/aalhour/C-Sharp-Algorithms

[38] "Tutorials point." [Online]. Available: https://www.tutorialspoint.com

[39] "Heapsort." [Online]. Available: https://www.growingwiththeweb.com/2012/11/algorithm-heapsort.html

[40] P. Publishing, "C# data structures and algorithms." [Online]. Available: https://github.com/PacktPublishing/C-Sharp-Data-Structures-and-Algorithms

[41] "C# sharp searching and sorting algorithm exercises: Insertion sort." [Online]. Available: https://www.w3resource.com/csharp-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-6.php

[42] "Merge sort." [Online]. Available: https://programm.top/en/c-sharp/algorithm/array-sort/merge-sort/

[43] B. Adebowale, "Merge sort in C#." [Online]. Available: https://www.c-sharpcorner.com/blogs/a-simple-merge-sort-implementation-c-sharp

[44] Karthik.AMR, "An easy guide to merge sorting algorithm." [Online]. Available: https://codeburst.io/stupids-guide-to-merge-sorting-algorithm-dfeca6094d7f

[45] "C# quick sort algorithm implementation." [Online]. Available: http://csharpexamples.com/c-quick-sort-algorithm-implementation/

[46] "Quick sort in C# for practice." [Online]. Available: https://gist.github.com/lbsong/6833729

[47] JetBrains, "dupFinder command-line tool." [Online]. Available: https://www.jetbrains.com/help/resharper/dupFinder.html

[48] Microsoft, "Near-duplicate code detector." [Online]. Available: https://github.com/microsoft/near-duplicate-code-detector

[49] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013, pp. 15–26.

[50] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2019, pp. 143–153.

[51] A. Solar-Lezama, "Program synthesis by sketching," Ph.D. dissertation, University of California at Berkeley, 2008.