SemCluster: Clustering of Programming Assignments based on Quantitative Semantic Features

David Perry
Purdue University
perry74@purdue.edu

Dohyeong Kim
Purdue University
kim1051@purdue.edu

Roopsha Samanta
Purdue University
roopsha@purdue.edu

Xiangyu Zhang
Purdue University
xyzhang@cs.purdue.edu

Abstract

A fundamental challenge in automated reasoning about programming assignments at scale is clustering student submissions based on their underlying algorithms. State-of-the-art clustering techniques are sensitive to control structure variations, cannot cluster buggy solutions with similar correct solutions, and either require expensive pair-wise program analyses or training efforts. We propose a novel technique that can cluster assignments based on their algorithmic essence: (A) how the input space is partitioned into equivalence classes and (B) how the problem is uniquely addressed within individual equivalence classes. We capture these algorithmic aspects as two quantitative semantic program features that are merged into a program's vector representation. Programs are then clustered using their vector representations. The computation of our first semantic feature leverages model counting to identify the number of inputs belonging to an input equivalence class. The computation of our second semantic feature abstracts the program's data flow by tracking the number of occurrences of a unique pair of consecutive values of a variable during its lifetime. The comprehensive evaluation of our tool SemCluster shows that SemCluster (1) generates far fewer clusters than other clustering techniques, (2) precisely identifies distinct solution strategies, and, (3) boosts the performance of automated feedback generation, all within a reasonable amount of time.

Keywords Program clustering, Program analysis, Quantitative reasoning

1 Introduction

Recent years have witnessed skyrocketing enrollments in introductory programming courses offered by universities and as Massive Open Online Courses (MOOCs) [4], as well as increased participation in online judge systems such as CodeChef [1], Codeforces [2], and HackerRank [3]. While this surge in interest in programming is exciting, these massive, new learning environments pose significant challenges in simulating the quality of education in smaller, traditional classrooms. In particular, instructional tasks such as personalized feedback and fair grading, are often prohibitive, if not impossible. Some platforms involve students in providing additional test-cases [11], or, for peer-feedback and peer-grading [46], but such strategies can suffer from inaccuracies, biases and latencies [28, 36].

Recognizing these challenges, researchers in multiple communities have started developing techniques for *automated* reasoning about programming assignments at scale [10, 14–17, 19, 22, 24, 25, 29, 31, 33, 34, 36, 41, 42]. Given large collections of solutions for individual programming assignments, many of these techniques rely on reducing the solution space by first *clustering similar solutions*. For instance, automated feedback generation or grading systems use a representative correct solution from each cluster to generate feedback or a grade for incorrect solutions, respectively [15, 19]. Tools for analyzing student data help instructors (as well as learners) view distinct, pedagogically valuable solutions by visualizing representative correct solutions from each cluster [14, 17, 22]. Unfortunately, the performance of most approaches for program clustering in education [14, 15, 17, 19, 24] is far from satisfactory. Clustering techniques such as [14, 15] place too much emphasis on syntactic program features and, moreover, require the program features to match exactly. This results in an excessive number of clusters, where semantically similar programs with small syntactical differences are placed in different clusters. For instance, Clara [15] generates 51 clusters for a programming assignment, HORSES, with 200 submissions and only 4 different solution strategies. Such fine-grained clustering is ineffectual in assisting instructors or automated reasoning engines in instructional tasks. Clustering techniques such as [17, 24] rely on expensive computations over pairs of programs (tree edit distance between abstract syntax trees and a notion of probabilistic, semantic equivalence, respectively). This greatly hinders the scalability of these clustering techniques. Some clustering techniques [19] are specialized to a particular problem domain (e.g., dynamic programming) and are not broadly applicable. A recent approach [42] successfully uses neural networks to learn *program embeddings* and redefines the state-of-the-art in program clustering. Unfortunately, this

PL'18, January 01–03, 2018, New York, NY, USA
2018.

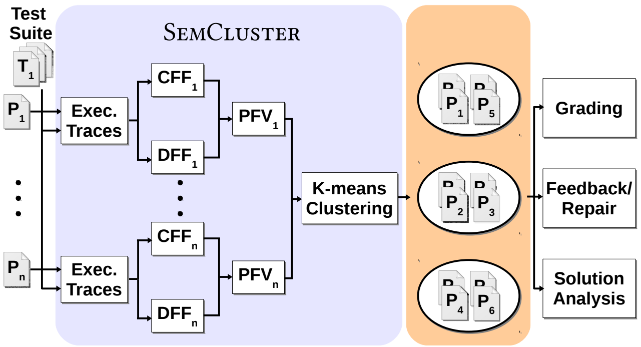


Figure 1. High-level overview of SemCluster.

approach requires a substantial training effort, both in terms of time and in manual effort in selection of training data, that affects its usability.

This paper advances the state-of-the-art in clustering of programming submissions with a new technique based on program analysis. Our main contribution is a vector representation of programs, based on purely semantic program features, which can be used with standard clustering algorithms from the machine learning literature. Our technique (sketched in Fig. 1) enjoys several desirable characteristics. First, it is able to cluster programs based on their high-level algorithmic solution strategy, ignoring syntactical and low-level implementation variations across programs. This results in far fewer clusters than most clustering approaches (our techniques generate 4 clusters for the prior-mentioned assignment HORSES). Second, by generating the vector representation, our technique avoids expensive pair-wise program analyses. Third, our technique is general and applicable to a broad range of programming assignments. Finally, our program analysis-based technique matches the clustering performance (and, in some cases, outperforms) that of the state-of-the-art neural network-based clustering technique [42], without requiring an expensive training effort. Our clustering approach can be used to drive many automated reasoning tasks in programming education and beyond (e.g., personalized feedback, grading, visualization, similarity detection, fault localization and program repair).

Our proposed program representation is based on the key observation that the essence of a solution to a programming problem lies in the way the problem space is partitioned into sub-spaces and how the problem is uniquely addressed within individual sub-spaces. We use *control flow features* (CFFs) to represent the former aspect of a solution strategy, as this aspect typically manifests in the use of control structures that ensure each sub-space corresponds to a particular control flow path. We use *data flow features* (DFFs) to represent the latter aspect of a solution strategy, as this aspect typically manifests in the use of different operations (along the path). Given a programming assignment solution and a test-suite, we compute CFFs by counting inputs that follow the same control flow paths as different tests. We compute DFFs as the frequency of occurrence of distinct pairs of successive values of individual variables in program executions on tests. CFFs and DFFs for each solution are merged to create a *program feature vector* (PFV). Finally, K-means clustering is used to cluster all solutions based on their PFVs.

We have implemented our proposed clustering approach in a tool, SemCluster, and evaluated it on a variety of programs drawn from CodeChef [1], CodeHunt [6] and GitHub. The evaluation on 17 real-world programming problems with 8,989 solutions shows that SemCluster generates 4-15 clusters. This is in sharp contrast to the 27-125 clusters generated by Clara [14] and OverCode [14]. We further demonstrate the high degree of precision with which SemCluster identifies unique algorithms among submissions, the ability of SemCluster to successfully drive Clara's feedback generation system and the reasonable run-time performance of SemCluster (3.6 minutes on average per assignment, with 529 submissions on average per assignment).

In summary, this paper makes the following contributions:

1. We propose an effective, efficient and broadly applicable program clustering technique based on a quantitative, semantic program representation (Sec. 4.3).
2. We present dynamic analyses to compute the control flow- (Sec. 4.1) and data flow-based (Sec. 4.2) components of our program representation
3. We comprehensively evaluate and demonstrate the effectiveness of our tool SemCluster (Sec. 5, Sec. 6) in identifying distinct solution strategies and in boosting the performance of automated feedback generation.

2 Motivating example

We use the example in Fig. 2 to illustrate why many program clustering techniques may fail to cluster programs with similar solution strategies. Existing techniques place great emphasis on syntactical differences instead of focusing on the semantic similarities of programs. We describe how our clustering approach, based on *quantitative semantic program features*, can capture the essence of solution strategies and address issues with existing techniques.

Fig. 2 contains code snippets from 3 student submissions for a programming assignment, "Filling the Maze", from CodeChef [1]. This assignment requires exploring a graph and determining if a given node is reachable from the starting node. The input, `graph`, to the functions `searchA`, `searchB`, and `searchC` is an adjacency matrix representing a graph (with only four nodes for simplicity). The output, `result`, is an array representing the order in which the nodes of the graph are traversed. Array `isChecked` tracks if a node has been traversed.

The solution strategies employed by functions `searchA` and `searchB` are significantly different from that used in `searchC`. Specifically, `searchA` and `searchB` use iterative *depth-first search* (DFS) to explore the graph and `searchC` uses *breadth-first search* (BFS). The only difference between

```
1 def searchA(graph):
2  stack = [0]
3  isChecked = [1,0,0,0]
4  result = [0]
5  node = 0
6  while node != -1:
7   nodeAdded = False
8   for i in range(4):
9    if graph[node][i] == 1 and
          isChecked[i] == 0:
10     stack.append(i)
11     isChecked[i] = 1
12     nodeAdded = True
13     result.append(i)
14     break;
15   if nodeAdded == False:
16    stack.pop()
17    if len(stack) > 0:
18     node = stack[-1]
19    else:
20     node = -1
21   else:
22     node = stack[-1]
23  return result
```

```
1 def searchB(graph):
2  stack = [0]
3  isChecked = [1,0,0,0]
4  result = [0]
5  node = 0
6  while node != -1:
7   valAdded = False
8   for j in range(4):
9    if graph[node][j] == 1 and
          isChecked[j] == 0 and
          valAdded == False:
10     stack.append(j)
11     isChecked[j] = 1
12     valAdded = True
13     result.append(j)
14   if valAdded == False:
15    stack.pop()
16    if len(stack) > 0:
17     node = stack[-1]
18    else:
19     node = -1
20   else:
21    node = stack[-1]
22  return result
```

```
1 def searchC(graph):
2  queue = [0]
3  isChecked = [1,0,0,0]
4  result = [0]
5  node = 0
6  for i in range(4):
7   node = queue.pop(0)
8   for j in range(4):
9    if graph[node][j] == 1 and
          isChecked[j] == 0:
10     queue.append(j)
11     isChecked[j] = 1
12     result.append(j)
13  return result
```

Figure 2. Two slightly different implementations of DFS, searchA and searchB and one implementation of BFS, searchC.

searchA and searchB is how their if statements (line 9) ensure that at most one unexplored child of the current node is explored and added to the stack. searchA implements this by inserting a break statement within the if's body. searchB conditions the if statement on the value of a Boolean variable, valAdded, indicating if a new node has been added to the stack.

Therefore, an effective clustering technique should cluster searchA and searchB together and place function searchC into a different cluster.

Limitations of existing techniques. Existing clustering techniques such as Clara [15] and OverCode [14] place the functions searchA, searchB, and searchC in separate clusters. This is primarily a limitation of their notions of *program similarity* and choice of program representation. Both Clara and OverCode only consider whether two programs exactly match in some chosen features, or not. Neither clustering technique tracks the *degree* to which two programs match in a feature. Such strict clustering strategies are especially problematic when attempting to cluster buggy programs with the closest correct version. In fact, Clara and OverCode can only cluster correct programs.

Further, while these techniques represent programs using both syntactic and semantic program features, the syntactic program features, in particular, are restrictive. For instance, Clara requires the control flow structures (i.e., loops and branches) of two programs match for them to be placed in the same cluster. The minor (highlighted) implementation differences between searchA and searchB cause a significant difference in their control flow structures. Hence, the programs are not clustered together by Clara and may not be clustered together by any technique that compares programs using syntactic features such as control flow structures. OverCode first *cleans* programs by renaming *common variables* identified using a dynamic analysis. OverCode requires the *set* of program statements of two clean programs exactly match for them to be clustered together. Again, the minor implementation differences between searchA and searchB cause a mismatch in the syntactic feature used by OverCode and hence, the functions are not clustered together.

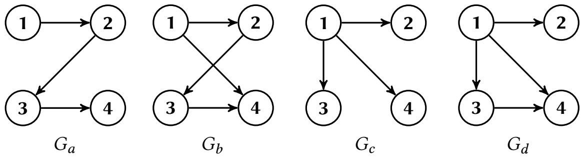


Figure 3. Two equivalent input graphs, G_a and G_b, for searchA and searchB, and two equivalent input graphs, G_c and G_d, for searchC.

Quantitative semantic program features. Our key observation is that the essence of problem-solving in computing is *divide-and-conquer*. Given a problem, the programmer often first partitions the input space into equivalence classes such that each class represents a unique way of addressing the problem. The specific partitioning used, thus, characterizes the underlying solution. Further, within such an *input equivalence class*, the set of operations used and their order also contribute in identifying the algorithm. We encapsulate the former as a *control flow program feature* and the latter as a *data flow program feature*. The overarching idea of our technique is to generate a quantitative program feature based on these two features so that a clustering algorithm can be used to effectively and efficiently cluster (correct and buggy) programs based on their underlying algorithms. Our method avoids an expensive pair-wise comparison of programs, which is prohibitive in our context due to the large number of programs considered.

Informally, the control flow program feature tracks the *volume* of inputs *flowing* through different control flow paths in the program, essentially describing the input space partitioning. To understand how the algorithm implemented in a program impacts the number of inputs corresponding to each control flow path in a program, consider the execution of functions searchA, searchB and searchC on graph G_a from Fig. 3. Regardless of their small implementation differences, searchA and searchB use DFS to visit the nodes of G_a in the order 1, 2, 3, 4, without any backtracking. Similarly, searchC uses BFS to visit G_a in the same order 1, 2, 3, 4. Observe that the order of visiting nodes, including backtracking, directly corresponds to a specific sequence of control flow decisions, i.e., a specific control flow path, in each program. Now consider the executions of the functions on graph G_b. Despite the extra edge in this graph, functions searchA and searchB still visit its nodes in the same order as graph G_a, i.e., 1, 2, 3, 4, without any backtracking. Thus, the executions of searchA and searchB on G_a and G_b follow identical control flow paths. In other words, these two input graphs fall into the same input equivalence class for both searchA and searchB. In contrast, when graph G_b is given as input to searchC, the order in which its nodes are explored changes to 1, 2, 4, 3. That is, G_a and G_b do not belong to the same input equivalence class for searchC. Following similar reasoning, graphs G_c and G_d belong to the same input equivalence class for searchC but not for searchA and searchB. This is because the nodes of G_c and G_d are visited in the same order 1, 2, 4, 3 (i.e. via the same control flow path) by searchC, but not by searchA and searchB.

The control flow feature represents the sizes of the input equivalence classes. For our example, the feature is computed for each program by counting the number of graphs that are explored in the exact same order by the program. Finally, an application of K-means clustering using the control flow feature successfully ignores the small syntactical differences between searchA and searchB and clusters them together, while placing searchC into a different cluster.

Remark. In Sec. 4.1, we show that the control flow feature is not adequate by itself as it only summarizes the control flow signature of a program. Hence, we introduce the data flow program feature, which summarizes the data flow signature of a program. The program representation used by our technique is computed as an aggregate over the control flow feature and the data flow feature.

3 Preliminaries

In this section, we present our program model, and review the concepts of model counting and K-means clustering that form the basis of our approach.

Program model. We introduce a simple programming language. A program P is composed of a function signature $f(i_1, \dots, i_q) : o$, a set of variables X, and a sequence of labeled statements $\sigma = s_1; \dots ; s_m$. The function f is defined over a set of input variables $I = \{i_1, \dots i_q\}$ and an output variable o for the function's returned value. The set of variables $X = \{x_1, \dots , x_r\}$ defines auxiliary variables employed by the programmer for the specific programming task. All variables are associated with a specific type and are only assigned appropriate values from a finite universe U of values[1]. Program statements are skip, return, assignment, conditional, or loop statements. Each statement is designated by a unique location identifier from the set $L = \{l_0, l_1, \dots , l_b, exit\}$ and can use any of the variables in $I \cup X \cup \{o\}$.

A program configuration ζ is a pair (l, ν) consisting of a program location $l \in L$ and a valuation function ν that assigns values to all program variables. Specifically, $\nu : I \cup \{o\} \cup X \mapsto U \cup \{nd\}$ where nd represents an undefined value. We use $(l, \nu) \rightarrow (l', \nu')$ to denote the execution of the statement at location l with valuation ν, resulting in a transfer of control to location l' with valuation ν'. An *input valuation* ν_I is a valuation such that for all input variables $i \in I$, $\nu_I(i) \neq nd$ and for all other variables $x \in X \cup \{o\}$, $\nu_I(x) = nd$. A program P's execution, $\pi_P(\nu_I)$, on input valuation ν_I is a sequence of configurations $\zeta_0, \zeta_1, \dots , \zeta_j$ where $\zeta_0 = (l_0, \nu_I)$, for all h, $\zeta_h \rightarrow \zeta_{h+1}$, and $\zeta_j = (exit, \nu_j)$. Thus, all program executions terminate at the *exit* location.

A *test* t is a pair (ν_I, res) where ν_I is an input valuation and *res* is the expected output. We use $\pi_P(t)$ to denote a program execution of P on the input valuation ν_I of test t. A *control flow path* is a projection of a program execution onto locations. Thus, if $\pi_P(t)$ is $(l_0, \nu_I), (l_1, \nu_1), \dots , (l_j, \nu_j)$, the *control flow path* induced by t, denoted $CFP_{P,t}$, is given by $l_0, l_1, \dots , l_j$. Note that many input valuations may induce the same control flow path. We say that two tests t and t' belong to the same *input equivalence class* [5] iff the control flow paths induced by them are the same i.e., $CFP_{P,t} \equiv CFP_{P,t'}$.

Model counting. Given a propositional formula F, #SAT or propositional model counting is the problem of computing the number of satisfying assignments to propositions in F. Propositional model counting is the canonical #P-complete problem. Practical solutions are based on *exact* counting as well as *approximate* counting of models [39, 45].

A less investigated problem, #SMT [7], extends the model counting problem to *measured* logical theories. A theory is measured if for every formula ϕ in the theory, the set of its models $[\![\phi]\!]$ is *measurable.* Given a formula ϕ in a measured theory, #SMT is the problem of computing the measure of $[\![\phi]\!]$. This is a well-known hard problem. While algorithms for exact and approximate model counting have been proposed for some theories over integer and real arithmetic [7, 12, 23], the approach used in this paper uses an eager encoding of #SMT into #SAT via *bit-blasting*.

K-means clustering. K-means clustering is a method for partitioning a set of data points into K clusters such that each data point d belongs to the cluster with the closest *centroid* or *mean.* The distance metric typically used is the squared Euclidean distance. Formally, given a set $\{d_1, d_2, \dots , d_n\}$ of

[1] Our method handles programs over scalars, arrays and pointers of types Booleans, integers, and characters.

Algorithm 1: Computing a CFF for a test

```
procedure ComputeCFF($P, t$)
    Input  : $P$: a program
             $t$: a test
    Output : $CFF_{P,t}$: the CFF obtained from $P$ and $t$
    $CFP_{P,t}$ = Execute ($P$, $t$)
    $f$ = CFP2SMT ($CFP_{P,t}$, $P$)
    $CFF_{P,t}$ = ModelCount (SMT2CNF ($f$))
    return $CFF_{P,t}$
```

data points, with each data point $d \in \mathbb{R}^m$ represented using an m-dimensional *feature vector*, K-means clustering seeks to partition the data points into K sets $C_{opt} = \{C_1, \ldots, C_K\}$ such that: $C_{opt} = \arg\min_C \sum_{i=1}^{K} \sum_{d \in C_i} \left\| d - \mu_i \right\|^2$. Here, μ_i, the centroid of cluster C_i, equals $\frac{1}{|C_i|} \sum_{d \in C_i} d$.

K-means clustering is known to be NP-hard. Effective approximate solutions [26] work by choosing K means and assigning data points to clusters with the closest mean. The means for clusters are then recomputed and the data point assignments are updated. This iterative refinement procedure is repeated until no changes occur.

4 Quantitative Semantic Features

Recall our overall SemCluster workflow from Fig. 1. Given a test suite T and a set $\mathcal{P}$ of solutions to a programming assignment, for each solution $P \in \mathcal{P}$, we first compute two classes of quantitative semantic features: control flow features (CFFs) and data flow features (DFFs). These features are then combined together into a single program feature vector (PFV) for each solution. Finally, K-means clustering is used to cluster all solutions based on their PFVs.

In this section, we describe the computation of CFFs, DFFs and PFVs. We fix a test suite $T = \{t_1, t_2, \ldots, t_m\}$.

4.1 Control Flow Features

Recall from Sec. 2 that, informally speaking, CFFs provide a quantitative summary of the way a program partitions the input space into equivalence classes. A plausible design of such a program feature involves counting the number of inputs in each input equivalence class. However, this requires exploring all possible paths of a program and can be intractable in general. Instead, we leverage the available test suite to restrict our focus to a subset of the input equivalence classes. Intuitively, *CFFs only track the number of inputs belonging to input equivalence classes that contain some test input.*

Given a program $P \in \mathcal{P}$ and a test $t \in T$, Algo. 1 computes the corresponding CFF, denoted $CFF_{P,t}$. First, the algorithm executes the program on the test input and computes the control flow path $CFP_{P,t}$ containing all program locations reached during execution. Next, the *path condition* for $CFP_{P,t}$ is generated as an SMT formula, whose satisfying solutions are inputs that drive P's execution through $CFP_{P,t}$. Finally, the algorithm computes $CFF_{P,t}$ by counting the number of satisfying solutions for the path condition. This is the well-known problem of #SMT (Sec. 3). In our implementation, we solve this by first converting the SMT formula to a SAT formula through *bit-blasting*, i.e., encoding every variable into a bitvector and every computation into a set of bit operations. Next, the SAT formula is transformed to *conjunctive normal form* (CNF), and handed off to an exact propositional model counter [39]. This encoding of #SMT into #SAT is exact as our input domain is finite.

Table 1. CFVs and their Euclidean distances for searchA, searchB, and searchC from Fig. 2.

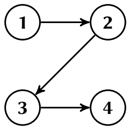
Program	CFV		Euclidean Distance		
	G_a	G_c	searchA	searchB	searchC
searchA	<8192, 1024>		N/A	0	7798.6
searchB	<8192, 1024>		0	N/A	7798.6
searchC	<1024, 8192>		7798.6	7798.6	N/A

For each program $P \in \mathcal{P}$, Algo. 1 is repeated to compute a CFF for every test in the test suite T. The resulting CFFs are then combined into a Control Flow Vector (CFV):

$$CFV_{P,T} = \langle CFF_{P,t_1}, CFF_{P,t_2}, \ldots, CFF_{P,t_m} \rangle. \tag{1}$$

Graph Search Example. The CFVs generated when the programs from Fig. 2 are executed on the input graphs G_a and G_c from Fig. 3 are shown in Table 1. The first dimension of the vectors in the column CFV contains the CFF for input G_a. The second dimension contains the CFF for input G_c. The last three columns in Table 1 indicate the Euclidean distances between each pair of vectors. As expected, the distance between searchA and searchB is small, and the distances between searchA and searchC and between searchB and searchC are large. This enables SemCluster to cluster searchA and searchB together and place searchC in a different cluster.

Let us take a closer look at the CFF value, 8192, computed for programs searchA and searchB on the input graph G_a from Fig. 3. Algo. 1 computes this result because the size of the input equivalence class of G_a for both programs is 8192. To understand this calculation, note that the input equivalence class of G_a for searchA (searchB) consists of all graphs with four nodes which induce program executions in searchA (searchB) that explore the graph nodes in the same order as G_a. Thus, this class contains any graph with edges from nodes 1 to 2, 2 to 3, and 3 to 4, or, in other words, any graph that can be obtained by adding additional edges to G_a. Now consider the adjacency matrix in Fig. 4 corresponding to graph G_a: the entry (i, j) in the matrix is 1



$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 4. Graph G_a and its adjacency matrix.

iff there is an edge from node i to node j, and 0 otherwise. We can calculate the size of the input equivalence class of G_a by counting the number of additional edges that can be added to the graph based on the number of 0's in the matrix. Since every entry that is 0 can be one of two possible values (1 if there is an edge or 0 otherwise) in each graph, the total number of graphs belonging to the equivalence class is $2^{13} = 8192$. Note that this computation is fully automated in SemCluster using constraint generation and model counting.

Inadequacy of CFF. While CFFs capture the partitioning of the input space, they alone may not suffice to make distinctions between all solution strategies. Consider the two programs `bubSort` and `selSort` in Fig. 5(a). Both programs take an n-size array of integers as input and return an array sorted in ascending order. While the algorithms employed by the two programs are very different — bubble sort and selection sort — they have the exact same set of input equivalence classes. To see this, consider the inputs in Fig. 5(b). Inputs I_a and I_b belong to the same equivalence class for both programs. This is because I_a and I_b have the same size and the same *relative ordering* of elements: smallest, largest, second largest, and smallest. Similarly, inputs I_c and I_d belong to the same equivalence class for both programs. As a result, the CFF for I_a (and I_c) is the same for `bubSort` and `selSort`, as shown in Fig. 5(c), and the distance between the CFVs of the programs is 0. Hence, the programs will be clustered together if we only rely on CFFs.

4.2 Data Flow Features

To cope with this problem, we propose another feature that provides a quantitative summary of a program's data flow. Indeed, when students design their programs, they not only need to design suitable control structures to partition the input space, but must also decide what operations to use and define how they interact with inputs and memory.

Existing techniques cluster programs based on a strategy that attempts to align variables across different student submissions [14, 15]. Two variables from different programs are aligned if the variables have the exact same sequence of values in program executions on the same input. This requirement for clustering is very rigid and prevents programs with slight implementation differences from being clustered together. Additionally, this variable alignment computation requires expensive pair-wise analysis across programs.

Therefore, we propose a quantitative program feature, DFF, that *abstracts* a program's data flow, is resilient to reordering of semantically equivalent operations, and is computable *locally* for each program without requiring pair-wise analysis. Informally, *DFFs track how many times a specific value in memory is changed to another specific value.* Intuitively, by modeling frequencies of specific value changes, we allow the feature computation to be local. By considering value changes (of variables), we encode part of the sequence of values of a variable in a program execution. While more complex data flow features can be designed, our DFFs were found to be highly effective when combined with CFFs (Sec. 6).

Given a program $P \in \mathcal{P}$, a test $t \in T$, and the set $\mathcal{P}$ of solutions, Algo. 2 computes the corresponding *set of* DFFs, as a hash table $DFFS_{P,t,\mathcal{P}}$. Note that given a test, while there is exactly one CFF for each program, there are multiple DFFs for each program. Let us first formalize our notion of value changes tracked by Algo. 2. Given a program execution $\pi_P(t)$, a value change, $v \rightarrow v'$, is a pair of distinct values $v \neq v'$ such that there exists variable $x \in X \cup \{o\}$ and successive program configurations $\zeta_h = (l, \nu)$ and $\zeta_{h+1} = (l', \nu')$ in $\pi_P(t)$ with $\nu(x) = v$ and $\nu'(x) = v'$. Observe that a value change does not track the variable or program configurations associated with it. Hence, there can be multiple instances of the same value change along a program execution (each associated with a different variable or program configuration). Given a program P and a test t, the ComputeLocalDFFS function in Algo. 2 computes the set of unique value changes and the number of instances of each unique value change in $\pi_P(t)$. The algorithm first executes an instrumented version of the program to collect a trace containing all value changes that occur during execution (lines 2-3). Next, this trace is scanned to find the number of times each value change occurs (lines 5-9). These frequencies are stored in the hash table *localDFFS*. The hash table's key is a string identifying a unique value change and the value is the number of occurrences of the value change in the program execution.

To compute *DFFs* for program P given test t, it is not enough to restrict our focus to the unique value changes in P's execution on t. Since the number of such unique value changes can vary across the executions of different programs on the same test t, computing DFFs of different programs for test t using ComputeLocalDFFS can result in DFFs of different sizes (which, in turn, can significantly complicate the computation of a uniformly-sized program feature vector for all programs). Hence, instead, ComputeDFFS computes $DFFS_{P,t,\mathcal{P}}$, by tracking *all value changes that occur in the executions of all programs in* $\mathcal{P}$ *on test* t. This ensures that the size of $DFFS_{P,t,\mathcal{P}}$ is the same for all $P \in \mathcal{P}$ for a given test t. The computation in lines 2-3 in Algo. 2 iterates through all programs in $\mathcal{P}$ calculating their *local* hash tables, *localDFFS*. Next, in lines 4-6, each entry in *localDFFS* is iterated through and the corresponding key is added to the *global* hash table $DFFS_{P,t,\mathcal{P}}$ with an initial value of zero. Finally, in lines 8-12, the *localDFFS* is recomputed for the target program P and merged with the previously computed $DFFS_{P,t,\mathcal{P}}$. Note the values of entries in $DFFS_{P,t,\mathcal{P}}$, that correspond to value changes absent in $\pi_P(t)$, are 0. In what follows, let us assume that $DFFS_{P,t,\mathcal{P}}$ is sorted according to keys and let $DFV_{P,t}$ denote a vector consisting of the values in the sorted $DFFS_{P,t,\mathcal{P}}$ (thus, $DFV_{P,t}$ is a vector of frequencies of value changes for some fixed ordering of value changes).

```
void swap(int *xp, int *yp)
{
 int temp = *xp;
 *xp = *yp;
 *yp = temp;
}

void bubSort(int arr[], int n)
{
 int i, j;
 for (i = 0; i < n-1; i++)
  for (j = 0; j < n-i-1; j++)
   if (arr[j] > arr[j+1])
    swap(&arr[j], &arr[j+1]);
}
```

```
void swap(int *xp, int *yp)
{
 int temp = *xp;
 *xp = *yp;
 *yp = temp;
}

void selSort(int arr[], int n)
{
 int i, j, min_idx;
 for (i = 0; i < n-1; i++)
 {
  min_idx = i;
  for (j = i+1; j < n; j++)
   if (arr[j] < arr[min_idx])
    min_idx = j;
  swap(&arr[min_idx],&arr[i]);
 }
}
```

(a)

I_a	1	5	3	1
I_b	2	7	3	2
I_c	4	3	2	1
I_d	9	7	2	1

(b)

Program	CFV		Euclidean Distance	
	I_a	I_c	bubSort	selSort
bubSort	<120,210>		N/A	0
selSort	<120,210>		0	N/A

(c)

Program	5→3	3→5	5→1	1→5	1→3	3→1
bubSort	1	1	1	1	1	1
selSort	0	0	1	1	0	0

(d)

Figure 5. (a) Two sorting programs, (b) example inputs, (c) CFVs of the programs on I_a, I_c, (d) the DFFs of the programs on I_a.

For each program $P \in \mathcal{P}$, Algo. 2 is repeated to generate $DFV_{P,t}$ for every test t in the test suite T. The vectors are then combined into a Data Flow Vector (DFV) for P:

$$DFV_{P,T} = \langle DFV_{P,t_1}, DFV_{P,t_2}, \ldots, DFV_{P,t_m} \rangle. \quad (2)$$

Sorting Example. An example of the simplified DFFs[2] for the programs in Fig. 5 on input I_a from Fig. 5 can be seen in Fig. 5 (d). Notice that the value changes used to create the DFFs are representative of the semantics of the programs. `bubSort` has to make 6 swaps as the algorithm iterates through the array making swaps each time two adjacent values are out of order. On the other hand, `selSort` only needs to make one swap. On its first pass, it makes no swaps as the smallest value is already in the 0th position of the array. On the next pass, it swaps the values in the 1st and 3rd positions, 5 and 1 respectively, which results in a sorted array. These differences in DFFs make it possible for SemCluster to distinguish the two sorting algorithms and place them in different clusters.

4.3 Program Feature Vector

Finally, we describe how to combine $CFV_{P,T}$ and $DFV_{P,T}$ for a program P into a single program feature vector (PFV) $PFV_{P,T}$. Unfortunately, this combination cannot be done by simply concatenating the two feature vectors. As stated earlier, for each test, CFVs contain one feature, while DFVs contain multiple features. Thus, a simple concatenation of CFV and DFV would generate a PFV with many more dimensions related to data flow. This would result in DFFs having a disproportionate impact on how programs are clustered.

Hence, we design the PFV for program P by normalizing each feature as follows. Let $M = max(|DFV_{P,t_1}|, |DFV_{P,t_2}|, \ldots, |DFV_{P,t_m}|)$ denote the maximum length of $DFV_{P,t}$ over all tests t. The normalized CFV, denoted $nCFV_{P,T}$, is given by $M \times CFV_{p,T}$. For each test $t \in T$, the vector $DFV_{P,t}$ is normalized to yield $nDFV_{P,t}$, given by $\frac{M}{|DFV_{P,t}|} \times DFV_{P,t}$. Finally, the PFV is computed as:

$$PFV_{P,T} = \langle nCFV_{P,T}, nDFV_{P,t_1}, \ldots, nDFV_{p,t_m} \rangle. \quad (3)$$

5 Implementation

Control flow features. To help compute a CFF for a program and test combination, we have implemented an LLVM [20] pass that instruments a submission by inserting a logging instruction at the beginning of each basic block. An execution of the instrumented submission produces a trace file containing the control flow path induced by the test input. We have also implemented an LLVM pass that walks through the target submission, using the trace file, to generate an SMT formula compatible with Microsoft's Z3 solver[9]. In addition to the program constraints, additional constraints are included in the SMT formula to enforce bounds, provided by an instructor, on the values of symbolic input variables. These bounds ensure the result of model counting is finite and allows the instructor to have a more fine-grained control over the set of inputs the clustering is based on.

We create models to encode the behavior of common libraries and data structures found in programs. Our models for arrays enforce a maximum array size and include operations for reading, writing, and commonly used functions defined in `string.h`. Our tool also supports pointer operations by implementing the monolithic memory model described in [32]. This model strikes a nice balance between performance and the ability to model common programming practices.

Once our tool creates an SMT formula, we use Z3's *bit-blast* tactic to produce a propositional formula, followed by

[2] For this figure, DFFs are calculated using only value changes that occur on the arrays themselves. Changes that occur on variables used for loop conditions and intermediate calculations are omitted for brevity.

Algorithm 2: Computing DFFs for a test

```
procedure ComputeDFFS($P$, $t$, $\mathcal{P}$)
    Input  : $P$: a program
             $t$: a test
             $\mathcal{P}$: a set of programs
    Output : $DFFS_{P,t,\mathcal{P}}$: a hash table containing value
             changes and their frequencies
    foreach $P \in \mathcal{P}$ do
        $localDFFS$ = ComputeLocalDFFS ($P$, $t$)
        foreach $key, val \in localDFFS$ do
            $DFFS_{P,t,\mathcal{P}} = \{key : 0\}$
        end
    end
    $localDFFS = \{\}$
    $localDFFS$ = ComputeLocalDFFS ($P$, $t$)
    foreach $key, val \in localDFFS$ do
        $DFFS_{P,t,\mathcal{P}}[key] = val$
    end
    return $DFFS_{P,t,\mathcal{P}}$
procedure ComputeLocalDFFS($P$, $t$)
    $instProg$ = InstDF ($P$)
    $vcTrace$ = Execute ($instProg$)
    $localDFFS = \{\}$
    foreach $valueChange \in vcTrace$ do
        if $valueChange \in localDFFS$ then
            $localDFFS[valueChange] + = 1$
        else
            $localDFFS[valueChange] = 1$
        end
    end
    return $localDFFS$
```

the state-of-the-art model counter, SharpSAT[39], to produce a CFF for the specific program and test combination.

Data flow features. To compute DFFs for a student submission and test input based on value changes, we have implemented an LLVM pass that inserts a logging function before and after any instruction that modifies memory. When the instrumented program is executed, a trace file containing the values of memory before and after each update is produced and used to compute DFFs.

Clustering. The PFVs computed from the CFFs and DFFS are given as input to the K-means clustering algorithm implemented in the machine learning library scikit-learn [26].

6 Evaluation

We present the results of a comprehensive evaluation of SemCluster's clustering performance. We compare SemCluster against the state-of-the-art — two program analysis-based approaches, Clara and OverCode, and a deep learning-based approach, Dynamic Program Embeddings (DPE). We evaluate the performance of clustering using the following criteria: number of clusters generated (Sec. 6.1), run-time performance of clustering-based instructional tasks (Sec. 6.2), precision of clusters w.r.t. known algorithms (Sec. 6.3), and precision of clusters w.r.t. instructional tasks (Sec. 6.4). Finally, we do an in-depth comparison with the DPE approach whose performance is closest to SemCluster (Sec. 6.5).

Table 2. Number of clusters generated by different clustering techniques.

Problem	Avg. LOC	# of Subs	C	OC	DPE	SC CFV	 DFV	 PFV	 CSPA
COINS	38	1033	89	101	10	4	9	8	8
PRIME1	59	920	120	125	9	14	12	9	8
CONFLIP	34	212	27	27	5	7	6	4	5
MARBLES	40	200	82	85	5	12	9	6	6
HORSES	36	200	42	51	6	9	7	5	5
LEPER	49	195	50	54	7	8	11	7	7
LAPIN	65	175	62	62	9	9	11	7	8
MARCHA1	45	100	37	37	6	6	7	4	5
BUYING2	32	100	33	33	5	7	4	5	5
SetDiff	16	273	52	59	5	4	5	5	6
MostOne	29	297	76	78	8	12	11	7	6
Comb	14	706	85	87	9	12	15	10	10
K-th Lar	11	949	120	125	15	17	20	14	13
ParenDep	18	820	101	111	16	22	21	15	16
LCM	15	806	99	103	12	17	24	13	12
ArrayInd	3	973	27	27	5	10	12	5	5
FibSum	14	1030	30	32	12	14	17	13	14

Dataset. We collected solutions to various programming assignments from the educational platform CodeChef [1] and Microsoft's competitive programming site CodeHunt [6]. To solve these assignments, students need a basic understanding of various algorithms, control structures, and data structures. Moreover, these assignments are representative of common challenges students encounter in an introductory programming course. In total, our dataset comprises 17 programming assignments, with a total of 8,989 submissions.

To perform a ground truth experiment for assignments with well-defined algorithms, we also collected 100 array sorting and graph searching implementations from GitHub. Each program was modified to accept inputs in a consistent format and executed on a set of tests to ensure correctness. Additionally, all implementations were manually inspected to ensure they matched their repositories' descriptions.

6.1 Number of Clusters

For our first evaluation, we track the number of clusters produced by different clustering techniques. We show that SemCluster clusters correct student submissions into a significantly smaller number of clusters than Clara and OverCode, while achieving results similar to DPE. Note that for this experiment only correct solutions are clustered as Clara and OverCode are incapable of clustering incorrect solutions.

The results can be found in Table 2. The first three columns contain the names of programming assignments, average number of lines of code per submission, and the number of submissions for each assignment. Assignments above the horizontal line are from CodeChef and the ones below are from CodeHunt (in all tables henceforth).

The number of clusters generated by Clara, OverCode and DPE are reported in the columns **C**, **OC** and **DPE**, resp. We report multiple results for SemCluster in the **SC** columns, corresponding to different strategies for combining CFVs and DFVs. The **CFV** and **DFV** columns show the results when the program feature vector simply equals the CFV (eq. 1) and DFV (eq. 2), respectively (and does not combine them). The **PFV** column shows the results when the CFV and DFV are normalized and combined into a single vector (Sec. 4.3). The **CSPA** column displays the results of using *cluster ensembles*, a machine learning approach for clustering data with multiple representations. The specific algorithm used is the cluster-based similarity partitioning algorithm (CSPA) [37].

Notice that, for the majority of assignments, CSPA and PFV return a smaller number of clusters than when using CFVs or DFVs individually. This justifies our choice in Sec. 4.3 to combine these two classes of features. Also note that CSPA and PFV achieve very similar results. This further justifies our PFV design that weighs CFVs and DFVs equally. Finally, we observed that the run-time performance of SemCluster is better on average when using PFV than CSPA. For all these reasons, the rest of our evaluation is performed on the version of SemCluster that uses the PFV representation.

The number of clusters generated by SemCluster (PFV) is dramatically lower than Clara and OverCode. This is expected as our approach is insensitive to syntactical differences among submissions and only considers semantic features when clustering. This reduction in the number of generated clusters is crucial in enabling instructors to feasibly analyze distinct solution strategies and in scaling down the solution space for automated reasoning engines for feedback, grading etc. Additionally, note that the cluster sizes reported by SemCluster and DPE are similar. This speaks volumes about the performance of SemCluster as it avoids the expensive task of training a neural network like DPE.

6.2 Run-time

To evaluate the scalability of SemCluster, we track the total amount of time required to compute clusters and perform a specific instructional task. This experiment was performed by using Clara, OverCode, DPE and SemCluster to cluster student submissions and using the respective clusters to drive Clara's feedback generation mechanism[3]. All resulting feedback was manually inspected to ensure correctness.

Table 3. Run-time performance of feedback generation using clusters generated by different tools. (T in minutes)

Problem	**C**			**OC**			**DPE**			**SC**		
	T	**M**	**A**	**T**	**M**	**A**	**T**	**M**	**A**	**T**	**M**	**A**
COINS	104.2	39	62.0	112.0	42	64.0	2.0	1	1.9	6.9	1	1.8
PRIME1	89.5	55	77.3	93.2	64	83.2	1.8	1	1.7	9.5	1	1.5
CONFLIP	5.5	8	10.1	5.5	8	10.1	.6	1	1.1	2.3	1	1.2
MARBLES	3.9	37	40.3	4.4	45	55.8	.5	1	1.6	2.1	1	1.4
HORSES	4.9	23	31.4	5.6	29	40.6	.7	1	1.7	2.8	1	1.9
LEPER	5.4	22	30.2	5.9	24	32.5	.7	1	2.3	4.4	1	2.3
LAPIN	5.9	35	47.8	6.1	35	47.8	.7	1	1.5	5.7	1	1.7
MARCHA1	2.5	15	22.1	2.3	15	22.1	.4	1	1.7	2.3	1	1.5
BUYING2	2.4	12	18.7	2.4	12	18.7	.4	1	1.3	2.6	1	1.3
SetDiff	3.5	22	30.3	4.2	32	38.6	.6	1	1.2	2.8	1	1.4
MostOne	6.7	35	47.2	6.9	35	49.8	.7	1	2.7	6.0	1	2.9
Comb	10.1	46	59.4	10.5	49	63.3	1.4	1	2.1	2.4	1	1.8
K-th Lar	11.8	63	78.2	13.4	68	81.2	1.8	1	1.7	2.2	1	2.1
ParenDep	12.6	50	69.7	15.3	59	77.3	1.5	1	1.7	3.1	1	1.9
LCM	12.2	45	59.2	13.4	47	62.3	1.4	1	2.0	2.8	1	2.2
ArrayInd	6.3	12	17.3	6.5	12	17.3	1.3	1	1.3	0.9	1	1.3
FibSum	6.3	16	21.2	6.8	17	25.3	1.6	1	2.1	3.0	1	1.4

The total time taken in minutes can be seen in Table 3 in the **T** column for each tool. Notice that for most assignments, Clara and OverCode take an order of magnitude more time than DPE and SemCluster.

To understand why the run-times of Clara and OverCode are worse than both DPE and SemCluster we recorded the number of program comparisons required by each to generate effective feedback. The results are also reported in Table 3 for each tool in the **M** and **A** columns, where M and A show the median and average number of required comparisons, resp. Notice that the number of comparisons required for Clara and OverCode are much higher than those for SemCluster and DPE. This is expected as Clara and OverCode cannot cluster incorrect and correct submissions together. Therefore, to find a correct submission that can be used to fix an incorrect submission, these tools need to compare the incorrect submission to submissions from each cluster of correct submissions, until a correct submission with an almost identical control structure is found. In contrast, the median number of comparisons required when using DPE and tool is always 1. Since these tools cluster semantically similar incorrect and correct submissions together, to repair an incorrect submission, it often suffices to use a random correct submission from the same cluster. Note that the average number of comparisons for both DPE and SemCluster are not 1. This occurs because there are some incorrect submissions that cannot be fixed using any correct submissions in the dataset. When (unsuccessfully) attempting to repair such submissions, DPE and SemCluster end up comparing them against all correct programs in their cluster, driving up the average number of comparisons.

[3] We use Clara for generating feedback as OverCode does not have a feedback generation mechanism and the repair/feedback tool used by DPE is not publicly available.

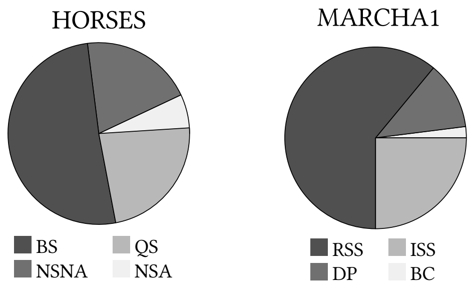


Figure 6. Solution strategy distributions for two assignments.

Note that the overall amount of time is the smallest when using DPE. This is because the reported run-time for DPE only includes the generation of embeddings for every submission and the application of K-means clustering. The most computationally expensive part of DPE — training — is omitted. An evaluation of DPE's training time and a discussion of its issues related to deployment can be found in Sec. 6.5.

6.3 Precision of Clusters: Solution Strategy

To judge the quality of clustering, it is not enough to simply examine the number of clusters generated. It is also important to evaluate the precision with which the clustering can identify high-level solution strategies across submissions

Manual review of CodeChef submissions. In our first experiment to evaluate the precision of SemCluster's clustering, we manually reviewed the clusters of two programming assignments, HORSES and MARCHA1. Our evaluation considers 100 randomly chosen submissions of both assignments from the CodeChef dataset.

HORSES requires students to read in a list of integers and find two values with the smallest difference. Both DPE and SemCluster classified the 100 solutions into 4 different clusters, while Clara and OverCode generated 42 and 51 clusters, respectively. A manual review of the clustering generated by SemCluster revealed the common high-level solution strategies in submissions within clusters (see Fig. 6). The first and second clusters, BS and QS, sort the list of numbers and then do a linear traversal of the sorted list, calculating the differences between adjacent values. The only difference is that BS uses bubble sort and QS uses quicksort. The third (NSNA) and fourth (NSA) clusters do not employ a sorting algorithm. As a result, they must perform an $O(n^2)$-traversal through the list, comparing all differences between pairs of values in the array. Their implementations differ in how they handle subtractions that yield negative numbers. NSNA uses an `if` statement to determine if differences are negative and multiplies them by −1. In contrast, NSA uses an absolute value function to handle these cases.

MARCHA1 is essentially the subset sum problem: given a list of integers and another integer m, is it possible to obtain a combination of list elements whose sum is m? While DPE and SemCluster generated 4 clusters, both Clara and OverCode generated 37 clusters. We repeated a manual review of the clustering generated by SemCluster to analyze the results. The breakdown of the four high-level solution strategies corresponding to the four clusters shown in Fig. 6 are as follows: iterative subset sum (ISS), recursive subset sum (RSS), dynamic programming (DP), and binary conversion (BC). The two most common strategies, RSS and ISS, explore all possible value combinations of various sizes until the desired combination sum is observed, in a recursive or iterative fashion, respectively. The third most common strategy is a more efficient implementation and employs dynamic programming. The final strategy uses binary conversion and is not as straight-forward a solution as the others. These submissions iterate from 0 to $2^n - 1$ in binary notation, using the binary numbers to index into the list and select elements for examination in each iteration (e.g. the binary number 1001 corresponds to a combination consisting of list elements at index 0 and 3). This approach is typically more space-efficient when the set of numbers is large.

Problem	Avg. LOC	# of Subs	Algs.	# Clusters			
				C	OC	DPE	SC
Sorting	72	100	4	45	51	4	4
Search	47	100	2	39	43	2	2

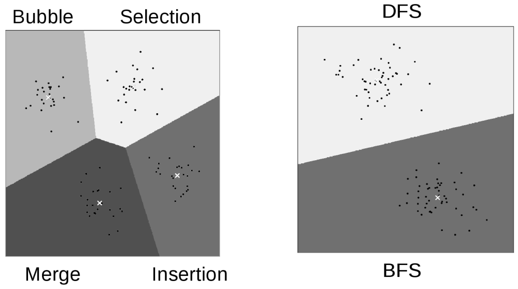


Figure 7. The clustering results for four sorting algorithms and two graph search algorithms.

This ability to cluster together a large number of student submissions with similar solution strategies, while still distinguishing such esoteric solutions, illustrates the effectiveness and precision of our clustering approach.

Ground truth experiment with GitHub programs. We further evaluated the precision of SemCluster's clustering with a ground truth experiment using a collection of programs from GitHub that implement well-known sorting algorithms (bubble sort, selection sort, insertion sort, merge sort) and graph search algorithms (DFS, BFS). The results can be seen in Fig. 7. Notice that SemCluster and DPE are able to perfectly partition the programs into clusters that correspond to the algorithm they implement. These results display the true potential of our clustering system.

6.4 Precision of Clusters: Feedback and Repair

To evaluate the usefulness of our approach in driving instructional tasks, we performed an experiment, similar to the one in Sec. 6.2, that uses Clara's feedback generation mechanism. For this task, Clara takes as input a correct and incorrect version of a student submission, aligns them, and generates feedback in the form of suggestions for repairing the buggy submission. For this to work effectively, the provided correct solution must implement an algorithm similar to the buggy submission.

All submissions from our dataset were first clustered using DPE and SemCluster. For each cluster, every correct submission was used to generate feedback for every incorrect submission belonging to the same cluster. The percentages of correct submissions that allowed Clara to generate correct feedback are reported in the **In** columns of Table 4. The suggested feedback/repair was applied to the buggy submissions and checked to ensure it passed all test cases. Note that the percentages reported for successful *in-cluster repairs* are quite high for both DPE and SemCluster, thereby indicating the usefulness of these clustering approaches in driving instructional tasks.

We further examined the performance of Clara's repair mechanism when aligning buggy submissions with correct submissions from different clusters. For each cluster, every correct submission was used to generate feedback for every incorrect submission belonging to a different cluster. The percentages of correct submissions that allowed Clara to generate correct feedback are reported in the **Out** columns of Table 4. As expected, the percentages reported for successful *out-of-cluster repairs* for both tools are small.

Clara's feedback mechanism targets generation of feedback based on *minimal repairs*. In the above experiment, we did not explore the question of how to choose correct submissions to minimally repair an incorrect submission from the same cluster. We hypothesize that minimal repairs can be generated using the correct submissions whose program representations are closest (in terms of Euclidean distance) to the incorrect submission. We test this hypothesis by tracking the average percentage of incorrect submissions for which the minimal repair is generated from the Top-1 and Top-3 *closest* correct submissions in the same cluster. The results in the **Top-1** and **Top-3** columns of Table 4 show that both SemCluster and DPE can be effectively used to generate minimal repairs based on our hypothesis. Note that SemCluster has a higher Top-1 accuracy for all but 3 programming assignments.

Table 4. Using clusters to drive feedback and repair.

Program	In		Out		Top-1		Top-3	
	DPE	SC	DPE	SC	DPE	SC	DPE	SC
COINS	85.2	83.2	7.3	9.4	85.3	87.2	93.4	95.2
PRIME1	77.2	80.9	14.2	12.5	81.9	82.1	88.7	89.3
CONFLIP	82.9	82.7	9.7	10.8	77.8	85.3	96.7	96.2
MARBLES	78.7	81.1	12.3	9.4	81.2	79.2	84.3	87.7
HORSES	89.9	84.3	10.3	12.1	85.2	88.4	93.7	95.4
LEPER	82.1	82.1	9.7	9.7	83.5	87.3	95.4	97.2
LAPIN	88.9	87.7	10.8	11.3	81.3	82.1	88.5	89.2
MARCHA1	82.8	79.3	6.6	7.2	83.2	85.4	90.7	92.5
BUYING2	88.2	88.2	11.1	11.1	75.3	77.2	84.7	85.3
SetDiff	86.1	87.2	16.9	15.8	90.1	88.2	96.8	94.3
MostOne	78.4	75.7	11.3	12.2	77.2	79.3	86.5	89.2
Comb	84.9	84.3	8.7	9.2	84.5	86.5	93.7	94.2
K-th Lar	77.2	79.9	17.2	14.3	74.8	73.2	91.2	88.5
ParenDep	88.2	87.3	11.9	12.9	71.4	73.8	84.3	87.2
LCM	77.4	79.1	20.2	18.2	83.4	82.8	93.2	91.4
ArrayInd	89.7	89.7	13.2	13.2	91.4	93.2	97.1	97.4
FibSum	77.2	87.9	9.1	5.2	87.3	91.2	97.3	96.9

Table 5. Run-time performance of clustering in minutes.

Assignment	Training		Rep.		*K*-means		Total	
	DPE	SC	DPE	SC	DPE	SC	DPE	SC
COINS	69.1	0	1.6	6.7	0.4	0.3	71.1	6.9
PRIME1	74.8	0	1.5	9.1	0.3	0.4	76.6	9.5
CONFLIP	37.7	0	0.3	2.0	0.3	0.2	38.3	2.3
MARBLES	35.0	0	0.3	1.8	0.2	0.2	35.6	2.1
HORSES	34.1	0	0.4	2.4	0.3	0.4	34.7	2.8
LEPER	71.7	0	0.3	4.0	0.4	0.4	72.4	4.4
LAPIN	101.1	0	0.3	5.2	0.5	0.5	101.9	5.7
MARCHA1	38.0	0	0.2	2.0	0.3	0.3	38.4	2.3
BUYING2	27.4	0	0.2	2.2	0.2	0.3	27.8	2.6
SetDiff	44.5	0	0.4	2.5	0.2	0.2	45.1	2.8
MostOne	65.3	0	0.5	5.7	0.2	0.3	65.9	6.0
Comb	37.9	0	1.2	2.2	0.2	0.2	39.3	2.4
K-th Lar	33.5	0	1.6	2.0	0.2	0.2	35.3	2.2
ParenDep	32.1	0	1.3	2.8	0.2	0.3	33.6	3.1
LCM	29.5	0	1.2	2.5	0.2	0.2	31.0	2.8
ArrayInd	23.9	0	1.2	0.7	0.2	0.2	25.3	0.9
FibSum	38.0	0	1.4	2.7	0.2	0.2	39.6	3.0

6.5 Comparison With DPE

As seen in the previous evaluations, DPE and SemCluster produce a similar number of clusters and have comparable precision when performing instructional tasks. The aspect in which the largest difference occurs is run-time. To better understand this difference, we take a closer look at the run-time behavior of DPE and SemCluster in Table 5. The **Training**, **Rep.**, ***K*-means** and **Total** columns depict the time taken for training, for generating representations for all submissions, for clustering using *K*-means and in total, respectively (in minutes).

Observe that the total times required by SemCluster are the same as Table 3; however, the total times for DPE have increased drastically. This is because DPE needs thousands of seconds for its very expensive training phase. First, the training data is generated. This requires identifying common mistakes made in programming assignment submissions and using this information to generate hundreds of thousands of mutants that implement an incorrect version of the solution. Next, each one of these mutated programs is executed to collect data that captures its semantic behavior. Finally, this data is used to train a neural network that generates program representations (i.e. embeddings) from programs.

However, an increase in run-time is not the only drawback of program clustering techniques requiring training. Another weakness affects the possibility of its deployment as training data may not always be available or possible to automatically generate. These situations can arise when programming assignments are being assigned for the first time or even if small changes are made to previously used assignments. To highlight this drawback, we show that the amount of training data available to DPE directly affects the number of clusters it report (see Table 6). Each column indicates the number of clusters reported by DPE when using the respective percentage of assignment submissions for training their model. Notice that the number of clusters is much larger when the amount of available training data is smaller. Finally, the last column reports the number of clusters generated by DPE when using a synthetic training set. This is the training strategy used in [42]. These mutants are used as training data to the neural network that generates the embeddings.

We emphasize that the number of clusters generated by DPE are similar to SemCluster only when using a difficult-to-generate synthetic training set. Therefore, it is very impressive that SemCluster is able to achieve this level of accuracy without any training. Additionally, because SemCluster does not require a training phase, the approach is more generalizable and does not overfit to a specific domain. SemCluster can be applied to any set of student submissions, whereas DPE must be fine-tuned for each application.

7 Related Work

Program clustering, similarity and representations in education. Early clustering approaches for student submissions represent programs using abstract syntax trees (ASTs) and compute their similarity using edit distances [17, 33], or canonicalization, i.e., application of semantics-preserving transformations [33, 47]. Codewebs [24] uses a notion of probabilistic semantic equivalence that clusters functionally equivalent but syntactically different AST sub-graphs. Clustering techniques such as OverCode [14] and Clara [15] employ a combination of control flow structures and variables values. However, these clustering techniques place a great deal of emphasis on syntactic details of programs, resulting in the generation of far too many clusters.

A recent direction in program clustering is the use of deep learning to learn program embeddings based on representing programs as ASTs, sequences of tokens, control flow structures, Hoare triples and sequences of program states [22, 27, 29, 31, 41, 42]. While this is a promising direction, these techniques require substantial training efforts and careful selection of training inputs.

Table 6. Number of clusters generated by DPE when using different training sets.

Problem	# of Clusters				
	60%	70%	80%	90%	Syn.
COINS	44	32	19	16	10
PRIME1	41	32	20	15	9
CONFLIP	31	24	16	9	5
MARBLES	37	22	12	10	5
HORSES	33	20	14	9	6
LEPER	41	23	11	9	7
LAPIN	55	32	21	12	9
MARCHA1	39	22	12	10	6
BUYING2	41	25	14	10	5
SetDiff	22	15	10	6	5
MostOne	33	25	16	8	8
Comb	45	31	19	12	9
K-th Lar	57	35	21	17	15
ParenDep	55	31	22	19	16
LCM	43	29	17	13	12
ArrayInd	34	20	10	7	5
FibSum	57	39	22	17	12

There are clustering approaches developed for specialized usage scenarios. CoderAssist [19] is a clustering technique that clusters student submissions for dynamic programming assignments based on domain-specific characteristics of various solution strategies. The approach in [10] is a statistical approach for classifying interactive programs using a combination of syntactic features. The clustering approach in [16] clusters programs by inferring transformations to fix incorrect programs. The transformations are learned from examples of students fixing their code.

Finally, outside of the context of clustering, different notions of syntactic as well as semantic program similarity have been proposed to drive automated feedback generation and repair [8, 36].

Code similarity, code cloning. Approaches for code similarity analysis and clone detection are based on static analyses for comparing ASTs [40, 48], tokens [18, 30, 35], and program dependence graphs [13, 21] to find similar code. However, these approaches may fail to detect similar code because of differences in syntactical features. Other methods use dynamic analysis to extract characteristics of programs by observing execution behaviors, or birthmarks, that can be used to identify program similarities [38, 43, 44]. However, they tend to (intentionally) ignore algorithmic differences of individual components.

8 Conclusion

We develop a novel clustering technique for programming assignments that makes use of model counting to identify the number of inputs that follow a specific control flow path, and leverages a dynamic analysis to collect the frequencies of each unique pair of consecutive values of a variable during its lifetime. These features are merged into a program's vector representation, which is then used for clustering programs. Our results show that SemCluster is highly effective in generating far fewer clusters than most existing techniques, precisely identifies distinct solution strategies, and, boosts the performance of automated feedback generation, all within a reasonable amount of time.

References

[1] [n. d.]. CodeChef. https://www.codechef.com/.

[2] [n. d.]. Codeforces. http://codeforces.com/.

[3] [n. d.]. HackerRank. https://www.hackerrank.com//.

[4] 2017. The 50 most popular MOOCs of all time. https://www.onlinecoursereport.com/the-50-most-popular-moocs-of-all-time/.

[5] Boris Beizer. 2003. *Software testing techniques*. Dreamtech Press.

[6] Judith Bishop, R Nigel Horspool, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2015. Code Hunt: Experience with coding contests at scale. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 398–407.

[7] Dmitry Chistikov, Rayna Dimitrova, and Rupak Majumdar. 2017. Approximate counting in SMT and value estimation for probabilistic programs. *Acta Informatica* 54, 8 (2017), 729–764.

[8] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification*. Springer, 383–401.

[9] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[10] A. Drummond, Y. Lu, S. Chaudhuri, C. Jermaine, J. Warren, and S. Rixner. 2014. Learning to Grade Student Programs in a Massive Open Online Course. In *2014 IEEE International Conference on Data Mining*. 785–790. https://doi.org/10.1109/ICDM.2014.142

[11] Stephen H Edwards and Manuel A Perez-Quinones. 2008. Web-CAT: automatically grading programming assignments. In *ACM SIGCSE Bulletin*, Vol. 40. ACM, 328–328.

[12] Matthew Fredrikson and Somesh Jha. 2014. Satisfiability Modulo Counting: A New Approach for Analyzing Privacy Properties. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (CSL-LICS '14)*. ACM, New York, NY, USA, Article 42, 10 pages. https://doi.org/10.1145/2603088.2603097

[13] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*. ACM, 321–330.

[14] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 7.

[15] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2016. Automated clustering and program repair for introductory programming assignments. *arXiv preprint arXiv:1603.03165* (2016).

[16] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale (L@S '17)*. 89–98. https://doi.org/10.1145/3051457.3051467

[17] Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas. 2013. Syntactic and functional variability of a million code submissions in a machine learning MOOC. (2013), 25.

[18] Jeong-Hoon Ji, Gyun Woo, and Hwan-Gue Cho. 2007. A source code linearization technique for detecting plagiarized programs. In *ACM SIGCSE Bulletin*, Vol. 39. ACM, 73–77.

[19] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-supervised verified feedback generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 739–750.

[20] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[21] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. 2006. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 872–881.

[22] Lannan Luo and Qiang Zeng. 2016. SolMiner: Mining Distinct Solutions in Programs. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 481–490. https://doi.org/10.1145/2889160.2889202

[23] Feifei Ma, Sheng Liu, and Jian Zhang. 2009. Volume computation for boolean combination of linear arithmetic constraints. In *International Conference on Automated Deduction*. Springer, 453–468.

[24] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. 2014. Codewebs: Scalable Homework Search for Massive Open Online Programming Courses. In *Proceedings of the 23rd International Conference on World Wide Web*. 491–502. https://doi.org/10.1145/2566486.2568023

[25] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. 2017. Automatic Grading and Feedback Using Program Repair for Introductory Programming Courses. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*. 92–97. https://doi.org/10.1145/3059009.3059026

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[27] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. 2015. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*. Springer, 547–553.

[28] Chris Piech, Jonathan Huang, Zhenghao Chen, Chuong Do, Andrew Ng, and Daphne Koller. 2013. Tuned models of peer assessment in MOOCs. (2013).

[29] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning program embeddings to propagate feedback on student code. *arXiv preprint arXiv:1505.05969* (2015).

[30] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. 2002. Finding plagiarisms among a set of programs with JPlag. *J. UCS* 8, 11 (2002), 1016.

[31] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. Skp: A Neural Program Corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for*

Humanity (SPLASH Companion 2016). 39–40. https://doi.org/10.1145/2984043.2989222

[32] Zvonimir Rakamarić and Alan J Hu. 2009. A scalable memory model for low-level code. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 290–304.

[33] Kelly Rivers and Kenneth R Koedinger. 2013. Automatic generation of programming feedback: A data-driven approach. In *The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)*, Vol. 50.

[34] Kelly Rivers and Kenneth R. Koedinger. 2015. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education* (2015), 1–28. https://doi.org/10.1007/s40593-015-0070-z

[35] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. 2003. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 76–85.

[36] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices* 48, 6 (2013), 15–26.

[37] Alexander Strehl and Joydeep Ghosh. 2002. Cluster ensembles—a knowledge reuse framework for combining multiple partitions. *Journal of machine learning research* 3, Dec (2002), 583–617.

[38] Haruaki Tamada, Keiji Okamoto, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. 2004. Dynamic software birthmarks to detect the theft of windows applications. In *International Symposium on Future Software Technology*, Vol. 20. Citeseer.

[39] Marc Thurley. 2006. sharpSAT–counting models with advanced component caching and implicit BCP. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 424–429.

[40] Nghi Truong, Paul Roe, and Peter Bancroft. 2004. Static analysis of students' Java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30*. Australian Computer Society, Inc., 317–325.

[41] Ke Wang, Rishabh Singh, and Zhendong Su. 2017. Search, Align, and Repair: Data-Driven Feedback Generation for Introductory Programming Exercises. In *Proceedings of Programming Language Design and Implementation (PLDI)*.

[42] Ke Wang, Zhendong Su, and Rishabh Singh. 2018. Dynamic Neural Program Embeddings for Program Repair. In *International Conference on Learning Representations*.

[43] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. 2009. Behavior based software theft detection. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 280–290.

[44] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. 2009. Detecting software theft via system call based birthmarks. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 149–158.

[45] Wei Wei and Bart Selman. 2005. A new approach to model counting. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 324–339.

[46] Daniel S. Weld, Eytan Adar, Lydia Chilton, Raphael Hoffmann, Eric Horvitz, Mitchell Koch, James Landay, Christopher H. Lin, and Mausam. [n. d.]. Personalized Online Education: ĂŤA Crowdsourcing Challenge. In *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*.

[47] Songwen Xu and Yam San Chee. 2003. Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering* 29, 4 (2003), 360–384.

[48] Wuu Yang. 1991. Identifying syntactic differences between two programs. *Software: Practice and Experience* 21, 7 (1991), 739–755.