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 small imperative programs 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 on benchmarks drawn from solutions to small programming assignments shows that SemCluster (1) generates far fewer clusters than other clustering techniques, (2) precisely identifies distinct solution strategies, and (3) boosts the performance of clustering-based program repair, all within a reasonable amount of time.

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]. Recognizing the challenges posed by these massive, new learning environments, researchers in multiple communities have started developing techniques for automated reasoning about programming assignments at scale [10, 13–16, 18, 19, 22, 24, 25, 28, 30, 32, 33, 35, 40, 41]. 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 [14, 18]. Tools for analyzing student data help instructors (as well as learners) view distinct, pedagogically valuable solutions by visualizing representative correct solutions from each cluster [13, 16, 22]. Unfortunately, the performance of most approaches for program clustering in education [13, 14, 16, 18, 24] is far from satisfactory. Clustering techniques such as [13, 14] 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 [14] generates 51 clusters for a programming assignment, HORSES, with 200 submissions and only 4 different solution strategies. Clustering techniques such as [16, 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 [18] are specialized to a particular problem domain (e.g., dynamic programming) and are not broadly applicable. A recent approach [41] successfully uses neural networks to learn program embeddings and redefines the state-of-the-art in program clustering. Unfortunately, this 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 small imperative programs 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. 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 [41], without requiring an expensive training effort. Our clustering approach can potentially 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 program 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 program are merged to create a program feature vector (PFV). Finally, K-means clustering is used to cluster all programs 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 [13]. 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 program repair 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 and efficient technique for clustering small, imperative programs 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 clustering-based program repair.

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 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 [14] and OverCode [13] 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.

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 $G_c$ 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, 3, 4 (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 imperative programming language. A program $P$ is composed of a function signature $f(i_1, \ldots, i_q) : o$, a set of variables $X$, and a sequence of labeled statements $\sigma = s_1; \ldots; s_m$. The function $f$ is defined over a set of input variables $I = \{i_1, \ldots, i_q\}$ and an output variable $o$ for the function’s returned value. The set of variables $X = \{x_1, \ldots, x_r\}$ 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.

Program statements are skip, return, assignment, conditional, or loop statements. Each statement is designated by a unique location identifier from the set $L = \{(l_0, l_1, \ldots, l_h, \text{exit})\}$ and can use any of the variables in $I \cup X \cup \{o\}$.

A program configuration $\xi$ is a pair $(l, v)$ consisting of a program location $l \in L$ and a valuation function $v$ that assigns values to all program variables. Specifically, $v : I \cup \{o\} \cup X \mapsto U \cup \{\text{nd}\}$ where nd represents an undefined value. We use $(l, v) \rightarrow (l', v')$ to denote the execution of the statement at location $l$ with valuation $v$, resulting in a transfer of control to location $l'$ with valuation $v'$. An input valuation $v_l$ is a valuation such that for all input variables $i \in I$, $v_l(i) \neq \text{nd}$ and for all other variables $x \in X \cup \{o\}$, $v_l(x) = \text{nd}$. A program $P$’s execution, $\pi_P(v_l)$, on input valuation $v_l$ is a sequence of configurations $\xi_0, \xi_1, \ldots, \xi_j$ where $\xi_0 = (l_0, v_l)$, for all $h, \xi_h \rightarrow \xi_{h+1}$, and $\xi_j = (\text{exit}, v_l)$. Thus, all program executions terminate at the exit location.

A test is a pair $(\nu_l, r)$ where $\nu_l$ is an input valuation and $r$ is the expected output. We use $\pi_P(t)$ to denote a program execution of $P$ on the input valuation $\nu_l$ of test $t$. A control flow path is a projection of a program execution onto locations. Thus, if $\pi_P(t) = (l_0, v_l), (l_1, v_l), \ldots, (l_j, v_l)$, the control flow path induced by $t$, denoted $\text{CFP}_P(t)$, is given by $l_0, l_1, \ldots, l_j$. 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., $\text{CFP}_P(t) \equiv \text{CFP}_P(t')$.

Model counting. Given a propositional formula $F$, $\text{SAT}$ 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.
problem. Practical solutions are based on exact counting as well as approximate counting of models [38, 44].

A less investigated problem, #SMT [7], extends the model counting problem to measured logical theories. A theory is measured if for every formula \( \phi \) in the theory, the set of its models \([\phi]\) is measurable. Given a formula \( \phi \) 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, 11, 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 \( \{d_1, d_2, \ldots, d_n\} \) of 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: \( C_{opt} = \arg \min_C \sum_{i=1}^{K} \sum_{d \in C_i} ||d - \mu_i||^2 \). Here, \( \mu_i \), 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 \textsc{SemCluster} workflow from Fig. 1. Given a test suite \( T \) and a set \( \mathcal{P} \) of solutions to a programming problem, 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.

#### Algorithm 1: Computing a CFF for a test

```plaintext
procedure ComputeCFF(P, t)
Input: \( P \) a program \( t \) a test
Output: \( CFF_{P,t} \) the CFF obtained from \( P \) and \( t \)
1. \( CFP_{P,t} = \text{Execute}(P, t) \)
2. \( f = \text{CFF2SMT}(CFP_{P,t}, P) \)
3. \( CFF_{P,t} = \text{ModelCount} \text{SMT2CNF}(f) \)
return \( CFF_{P,t} \)
```

<table>
<thead>
<tr>
<th>Program</th>
<th>CFV</th>
<th>Euclidean Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( G_a )</td>
<td>( G_c )</td>
</tr>
<tr>
<td>searchA</td>
<td>&lt;8192, 1024&gt;</td>
<td>N/A</td>
</tr>
<tr>
<td>searchB</td>
<td>&lt;8192, 1024&gt;</td>
<td>0</td>
</tr>
<tr>
<td>searchC</td>
<td>&lt;1024, 8192&gt;</td>
<td>7798.6</td>
</tr>
</tbody>
</table>

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 hand-off to an exact propositional model counter [38]. This encoding of #SMT into #SAT is exact as our input domain is finite.

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):

\[
CFF_{P,T} = (CFF_{P,t_1}, CFF_{P,t_2}, \ldots, CFF_{P,t_m}).
\]  

**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 \textsc{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 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 \( 2^{113} = 8192 \). 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 programmers 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 [13, 14]. 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 re-ordering 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 \( \text{DFFs}_{P,t} \). 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 \( \zeta_h = (l,v) \) and \( \zeta_{h+1} = (l',v') \) in \( \pi_P(t) \) with \( v(x) = v \) and \( v'(x) = v' \). 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 \( \text{ComputeLocalDFFs} \) 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 \( \text{localDFFs} \). 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 \( \text{ComputeLocalDFFs} \) can result in **DFFs** of different sizes (which, in turn, can significantly complicate
Notice that the value changes used to create the DFFs are

The computation in lines 2-3 in Algo. 2 iterates through all

localDFFS

conditions and intermediate calculations are omitted for brevity.

On its first pass, it makes no swapping as the smallest value is

making swaps each time two adjacent values are out of order.

It needs to make 6 swaps as the algorithm iterates through the array,

representative of the semantics of the programs.

Sorting Example. An example of the simplified DFFs

is a vector consisting of the values in the sorted

changes absent in

d and the corresponding key is added to the

global

Next, in lines 4-6, each entry in

localDFFS

is iterated through

and the corresponding key is added to the
global hash table, localDFFS.

Note the values of entries in

dFFS

that correspond to value changes absent in

πP (t), are 0. In what follows, let us assume

DFFS

is sorted according to keys and let

DFPs

denote a vector consisting of the values in the sorted

DFFS

(thus,

DFP

is a vector of frequencies of value changes for

some fixed ordering of value changes).

For each program

P ∈ P,

Algo. 2 is repeated to generate

DFP

for every test

in the test suite

T. The vectors are then combined into a Data Flow Vector (DFV) for

P:

\[ DFP_{P,T} = \langle DFP_{P,t_1}, DFP_{P,t_2}, \ldots, DFP_{P,t_m} \rangle. \]  

(2)

Sorting Example. An example of the simplified DFFs\(^5\) for

the programs in Fig. 5 on input

Ia

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

and

DFV

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 as many features as the number of unique value changes that occur during the execution of all solutions on the specific test. 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 disproportinate 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

over all tests

T. The normalized CFV, denoted

nCFV

, is given by

\( M \times CFV_{P,T} \). For each test

T, the vector

DFV

is normalized to yield

nDFV

, given by

\( \frac{M}{|DFV_{P,t}|} \times DFV_{P,t} \).

Finally, the PFV is computed as:

\[ PFV_{P,T} = \langle nCFV_{P,T}, nDFV_{P,t_1}, \ldots, nDFV_{P,t_m} \rangle. \]  

(3)

5 Implementation

Control flow features. To compute CFFs we have implemented an LLVM [20] pass that inserts a logging instruction at the beginning of each basic block in a submission. An execution of the instrumented submission produces a trace file containing the control flow path induced by the test input.

---

\(^5\)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.
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.

We create models to encode the behavior of common libraries and data structures. Our models for arrays enforce a maximum array size and include operations for reading, writing, and many functions defined in string.h. Our tool also supports pointer operations by implementing the monolithic memory model described in [31].

Once our tool creates an SMT formula, we use Z3’s bitblast tactic to produce a propositional formula, followed by the state-of-the-art model counter, SharpSAT [38], to produce a CFF for the specific program and test combination. **Data flow features.** To compute DFFs 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.** PFVs are given as input to the K-means clustering algorithm implemented in the 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) [41].

We evaluate the performance of clustering using the following criteria: number of clusters generated (Sec. 6.1), run-time performance of clustering-based program repair (Sec. 6.2), precision of clusters w.r.t. program repair (Sec. 6.3), and precision of clusters w.r.t. known algorithms (Sec. 6.3). Finally, we do an in-depth comparison with the DPE approach whose performance is closest to SemCluster (Sec. 6.5).

**Dataset.** We collected solutions to various programming assignments from the educational platform CodeChef [1] and Microsoft’s competitive programming site CodeHunt [6]. In total, our dataset comprises 17 programming assignments, with a total of 8,989 submissions written in C.

Additionally, we collected 100 array sorting and graph searching implementations written in C from GitHub to perform a ground truth experiment (Sec. 6.3) for assignments with well-defined algorithms. Each of the GitHub programs were modified to accept inputs in a consistent format, executed on a set of tests to ensure correctness, and inspected to ensure they match 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, respectively. 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 SPF 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 combining CFVs and DFVs 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) [36].

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 which weighs CFVs and DFVs equally. Finally, we observed that the run-time performance of SemCluster using PFV is better than CSPA on all benchmarks, performing clustering 1.69x faster on average. 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. Finally, 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 automated reasoning task — program repair. This experiment was performed by using Clara, OverCode, DPE and SemCluster to cluster student submissions and using the respective clusters to drive Clara’s automated program repair engine.3 All resulting repairs were manually inspected to ensure correctness. 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 repairs. 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, respectively. Note 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, 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 SemCluster is always 1. Since these tools cluster semantically similar incorrect and correct submissions together, it often suffices to use a random correct submission from the same cluster when repairing. 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

---

3Note that while Clara uses its program repair engine to automatically generate feedback for students, evaluation of the pedagogical value of the generated feedback is outside the scope of this work. OverCode does not have a repair generation mechanism and the repair engine used by DPE is not publicly available.

<table>
<thead>
<tr>
<th>Problem</th>
<th>C</th>
<th>T</th>
<th>M</th>
<th>A</th>
<th>OC</th>
<th>T</th>
<th>M</th>
<th>A</th>
<th>DPE</th>
<th>T</th>
<th>M</th>
<th>A</th>
<th>SC</th>
<th>T</th>
<th>M</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>CINS</td>
<td>104.2</td>
<td>39</td>
<td>62.0</td>
<td>112.0</td>
<td>42</td>
<td>64.0</td>
<td>2.0</td>
<td>1</td>
<td>1.9</td>
<td>6.9</td>
<td>1</td>
<td>1.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRIME1</td>
<td>89.5</td>
<td>55</td>
<td>77.3</td>
<td>93.2</td>
<td>64</td>
<td>83.2</td>
<td>1.8</td>
<td>1</td>
<td>1.7</td>
<td>9.5</td>
<td>1</td>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONF1L</td>
<td>5.5</td>
<td>8</td>
<td>10.1</td>
<td>5.5</td>
<td>8</td>
<td>10.1</td>
<td>.6</td>
<td>1</td>
<td>1.1</td>
<td>2.3</td>
<td>1</td>
<td>1.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MARBLES</td>
<td>3.9</td>
<td>37</td>
<td>40.3</td>
<td>4.4</td>
<td>45</td>
<td>55.8</td>
<td>5.1</td>
<td>1</td>
<td>1.6</td>
<td>2.1</td>
<td>1</td>
<td>1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HORSES</td>
<td>4.9</td>
<td>23</td>
<td>31.4</td>
<td>5.6</td>
<td>29</td>
<td>40.6</td>
<td>.7</td>
<td>1</td>
<td>1.7</td>
<td>2.8</td>
<td>1</td>
<td>1.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEPER</td>
<td>5.4</td>
<td>22</td>
<td>30.2</td>
<td>5.9</td>
<td>24</td>
<td>32.5</td>
<td>.7</td>
<td>1</td>
<td>2.3</td>
<td>4.4</td>
<td>1</td>
<td>2.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPIN</td>
<td>5.9</td>
<td>35</td>
<td>47.8</td>
<td>6.1</td>
<td>35</td>
<td>47.8</td>
<td>.7</td>
<td>1</td>
<td>1.5</td>
<td>5.7</td>
<td>1</td>
<td>1.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MARCHA1</td>
<td>2.5</td>
<td>15</td>
<td>22.1</td>
<td>2.3</td>
<td>15</td>
<td>22.1</td>
<td>.4</td>
<td>1</td>
<td>1.7</td>
<td>2.3</td>
<td>1</td>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUYING2</td>
<td>2.4</td>
<td>12</td>
<td>18.7</td>
<td>2.4</td>
<td>12</td>
<td>18.7</td>
<td>.4</td>
<td>1</td>
<td>1.3</td>
<td>2.6</td>
<td>1</td>
<td>1.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetDiff</td>
<td>3.5</td>
<td>22</td>
<td>30.3</td>
<td>4.2</td>
<td>32</td>
<td>38.6</td>
<td>.6</td>
<td>1</td>
<td>1.2</td>
<td>2.8</td>
<td>1</td>
<td>1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MostOne</td>
<td>6.7</td>
<td>35</td>
<td>47.2</td>
<td>6.9</td>
<td>35</td>
<td>49.8</td>
<td>.7</td>
<td>1</td>
<td>2.7</td>
<td>6.0</td>
<td>1</td>
<td>2.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comb</td>
<td>10.1</td>
<td>46</td>
<td>59.4</td>
<td>10.5</td>
<td>45</td>
<td>63.3</td>
<td>1.4</td>
<td>1</td>
<td>2.1</td>
<td>2.4</td>
<td>1</td>
<td>1.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kth Lar</td>
<td>11.8</td>
<td>63</td>
<td>78.2</td>
<td>13.4</td>
<td>68</td>
<td>81.2</td>
<td>1.8</td>
<td>1</td>
<td>1.7</td>
<td>2.2</td>
<td>1</td>
<td>2.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParenDep</td>
<td>12.6</td>
<td>50</td>
<td>69.7</td>
<td>13.5</td>
<td>59</td>
<td>73.7</td>
<td>1.5</td>
<td>1</td>
<td>1.7</td>
<td>3.1</td>
<td>1</td>
<td>1.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCM</td>
<td>12.2</td>
<td>45</td>
<td>59.2</td>
<td>13.4</td>
<td>47</td>
<td>62.3</td>
<td>1.4</td>
<td>1</td>
<td>2.0</td>
<td>2.8</td>
<td>1</td>
<td>2.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ArrayInd</td>
<td>6.3</td>
<td>12</td>
<td>17.3</td>
<td>6.5</td>
<td>12</td>
<td>17.3</td>
<td>1.3</td>
<td>1</td>
<td>1.3</td>
<td>0.9</td>
<td>1</td>
<td>1.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FibSum</td>
<td>6.3</td>
<td>16</td>
<td>21.2</td>
<td>6.8</td>
<td>17</td>
<td>25.3</td>
<td>1.6</td>
<td>1</td>
<td>2.1</td>
<td>3.0</td>
<td>1</td>
<td>1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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 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.

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 compute these differences. NSNA uses an if statement to determine if differences are negative and multiplies them by $-1$. In contrast, NSA uses an absolute value function.

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

<table>
<thead>
<tr>
<th>Problem</th>
<th>Avg. LOC</th>
<th># of Subs</th>
<th>Algs.</th>
<th># Clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorting</td>
<td></td>
<td>100</td>
<td>4</td>
<td>45</td>
</tr>
<tr>
<td>Search</td>
<td>47</td>
<td>100</td>
<td>2</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>43</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

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

DPE and SemCluster generated 4 clusters, both Clara and OverCode generated 37 clusters. 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 straightforward 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 can be more space-efficient.

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.

6.4 Precision of Clusters: Program Repair

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

All submissions from our dataset were first clustered using DPE and SemCluster. For each cluster, every correct submission was used to generate repairs for every incorrect submission belonging to the same cluster. The percentages of correct submissions that allowed CLARA to generate correct repairs are reported in the In columns of Table 4. The suggested 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 automated program repair.

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 repairs for every incorrect submission belonging to a different cluster. The percentages of correct submissions that allowed CLARA to generate correct repairs 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 targets generation of 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 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.

### 6.5 Comparison With DPE

As seen in the previous evaluations, DPE and SemCluster produce a similar number of clusters and have comparable precision. 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 of DPE’s 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.

### Table 4. Using clusters to drive program repair.

<table>
<thead>
<tr>
<th>Program</th>
<th>In</th>
<th>Out</th>
<th>Top-1</th>
<th>Top-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPE</td>
<td>SC</td>
<td>DPE</td>
<td>SC</td>
<td>DPE</td>
</tr>
<tr>
<td>COINS</td>
<td>85.2</td>
<td>83.2</td>
<td>7.3</td>
<td>9.4</td>
</tr>
<tr>
<td>PRIME1</td>
<td>77.2</td>
<td>80.9</td>
<td>14.2</td>
<td>12.5</td>
</tr>
<tr>
<td>CONFLIP</td>
<td>82.9</td>
<td>82.7</td>
<td>9.7</td>
<td>10.8</td>
</tr>
<tr>
<td>MARBLES</td>
<td>78.7</td>
<td>81.1</td>
<td>12.3</td>
<td>9.4</td>
</tr>
<tr>
<td>HORSES</td>
<td>89.9</td>
<td>84.3</td>
<td>10.3</td>
<td>12.1</td>
</tr>
<tr>
<td>LEPER</td>
<td>82.1</td>
<td>82.1</td>
<td>9.7</td>
<td>9.7</td>
</tr>
<tr>
<td>LAPIN</td>
<td>88.9</td>
<td>87.7</td>
<td>10.8</td>
<td>11.3</td>
</tr>
<tr>
<td>MARCHA1</td>
<td>82.8</td>
<td>79.3</td>
<td>6.6</td>
<td>7.2</td>
</tr>
<tr>
<td>BUYING2</td>
<td>88.2</td>
<td>88.2</td>
<td>11.1</td>
<td>11.1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Training</th>
<th>Rep.</th>
<th>K-means</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPE</td>
<td>SC</td>
<td>DPE</td>
<td>SC</td>
<td>DPE</td>
</tr>
<tr>
<td>COINS</td>
<td>69.1</td>
<td>0.1</td>
<td>6.7</td>
<td>0.4</td>
</tr>
<tr>
<td>PRIME1</td>
<td>74.8</td>
<td>1.5</td>
<td>9.1</td>
<td>0.3</td>
</tr>
<tr>
<td>CONFLIP</td>
<td>37.7</td>
<td>0.3</td>
<td>2.0</td>
<td>0.3</td>
</tr>
<tr>
<td>MARBLES</td>
<td>35.0</td>
<td>0.3</td>
<td>1.8</td>
<td>0.2</td>
</tr>
<tr>
<td>HORSES</td>
<td>34.1</td>
<td>0.4</td>
<td>2.4</td>
<td>0.3</td>
</tr>
<tr>
<td>LEPER</td>
<td>71.7</td>
<td>0.3</td>
<td>4.0</td>
<td>0.4</td>
</tr>
<tr>
<td>LAPIN</td>
<td>101.1</td>
<td>0.3</td>
<td>5.2</td>
<td>0.5</td>
</tr>
<tr>
<td>MARCHA1</td>
<td>38.0</td>
<td>0.2</td>
<td>2.0</td>
<td>0.3</td>
</tr>
<tr>
<td>BUYING2</td>
<td>27.4</td>
<td>0.2</td>
<td>2.2</td>
<td>0.2</td>
</tr>
<tr>
<td>SetDiff</td>
<td>44.5</td>
<td>0.4</td>
<td>2.5</td>
<td>0.2</td>
</tr>
<tr>
<td>MostOne</td>
<td>65.3</td>
<td>0.5</td>
<td>5.7</td>
<td>0.2</td>
</tr>
<tr>
<td>Comb</td>
<td>37.9</td>
<td>0.2</td>
<td>1.2</td>
<td>0.2</td>
</tr>
<tr>
<td>K-th Lar</td>
<td>33.5</td>
<td>0.1</td>
<td>1.6</td>
<td>0.2</td>
</tr>
<tr>
<td>ParenDep</td>
<td>32.1</td>
<td>0.1</td>
<td>1.3</td>
<td>0.2</td>
</tr>
<tr>
<td>LCM</td>
<td>29.5</td>
<td>0.1</td>
<td>1.2</td>
<td>0.2</td>
</tr>
<tr>
<td>ArrayInd</td>
<td>23.9</td>
<td>0.1</td>
<td>1.2</td>
<td>0.2</td>
</tr>
<tr>
<td>FibSum</td>
<td>38.0</td>
<td>1.4</td>
<td>2.7</td>
<td>0.2</td>
</tr>
</tbody>
</table>
Unfortunately, training data may not always be available or possible to generate, affecting the possibility of deployment. 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 [31]. 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. Additionally, because SemCluster does not require a training phase, the approach is more generalizable and does not overfit to a specific domain.

### 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 [16, 32], or canonicalization [32, 45]. Codewebs [24] uses a notion of probabilistic semantic equivalence that clusters functionally equivalent but syntactically different AST sub-graphs. Clustering techniques such as OverCode [13] and CLARA [14] 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, 28, 30, 40, 41]. While this is a promising direction, these techniques require substantial training efforts and careful selection of training inputs.

There are clustering approaches developed for specialized usage scenarios. CoderAssist [18] 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 [15] 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, 35].

#### Code similarity, code cloning

Static analysis based approaches for code similarity compare ASTs [39, 46], tokens [17, 29, 34], and program dependence graphs [12, 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 benchmarks [37, 42, 43]. However, they tend to (intentionally) ignore algorithmic differences of individual components.

### 8 Conclusion

We develop a novel clustering technique for small imperative programs based on program features that are quantitative summaries of a program’s partitioning of its input space and a program’s data flow. 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 program repair, all within a reasonable amount of time.

### Acknowledgments

We thank the shepherd, Shriram Krishnamurthi, and the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards CCF-1846327, 1748764 and 1409668, ONR under contracts N000141410468 and N000141712947, and Sandia National Lab under award 1701331. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.


