Customizing Feedback for Introductory Programming Courses using Semantic Clusters[⋆]

Victor J. Marin[1], Hadi Hosseini[2], and Carlos R. Rivero[1]

[1] Rochester Institute of Technology
`vxm4964@rit.edu, crr@cs.rit.edu`
[2] Pennsylvania State University
`hadi@psu.edu`

Abstract. The number of introductory programming learners is increasing worldwide. Delivering feedback to these learners is important to support their progress; however, traditional methods to deliver feedback do not scale to thousands of programs. We identify several opportunities to improve a recent data-driven technique to analyze individual program statements. These statements are grouped based on their semantic intent and usually differ on their actual implementation and syntax. The existing technique groups statements that are semantically close, and considers outliers those statements that reduce the cohesiveness of the clusters. Unfortunately, this approach leads to many statements to be considered outliers. We propose to reduce the number of outliers through a new clustering algorithm that processes vertices based on density. Our experiments over six real-world introductory programming assignments show that we are able to reduce the number of outliers and, therefore, increase the total coverage of the programs that are under evaluation.

Keywords: Graph clustering · Approximate graph alignment.

1 Introduction

The number of novice programming learners has been steadily increasing for the last years in both traditional and online settings [1, 4]. Traditional methods mainly rely on manual grading, and, as a result, are tedious, particularly for providing effective feedback to many novice learners [1]. There is also a need to assist instructors with the current "boom" in computing courses while continuing to provide a quality educational experience [9]. Current techniques to analyze learner programs mainly focus on automating feedback delivery, and they usually do not support an active role of the instructor [6]. Such an active role can be reflected in many forms, e.g., by enabling a flexible grading scheme that is refined during the actual grading [3]. Additionally, the delivered feedback is internally decided by the tools and the instructor, which can provide very useful information regarding an assignment, has almost no opportunity to

[⋆] This material is based upon work supported by the National Science Foundation under Grant No. 1915404.

customize their feedback and attune it according to each learner's particular needs [6, 7]. These tools perform several tasks over the set of available programs, for example, automated analysis and repairing [10]. These tasks can help an instructor to gain insights regarding learners' strengths and weaknesses; however, these opportunities have not been fully exploited by existing tools.

To address these issues, Marin and Rivero [8] presented a technique to analyze correct programs that pass a set of test cases. The individual statements of these programs are clustered according to their semantic intent, e.g., checking whether an integer `i` is greater than another integer `m` (to keep track of the maximum number) can be accomplished in several ways: `if (i > m)`, `if (m < i)`, `if (!(i <= m))`, or `if (a[j] > m)`. These individual statements can be clustered together as "Check whether current number is greater than maximum." Statement clusters are promising to enable customized, automated feedback delivery [5, 8]. The existing technique relies on a structural graph clustering algorithm that imposes strong graph connectivity restrictions to form clusters. When individual program statements do not clearly belong to a cluster, they are categorized as outliers. As a result, this technique discovers a small number of statement clusters in real-world programs that are very cohesive, but misses to classify many other individual statements. In the reported experiments, outliers are between 30% and 70% of the total number of program statements.

In this paper, we identify opportunities to cluster additional statements under the same semantic intents without compromising the cohesiveness of the discovered clusters. We assume that a graph that connects all individual program statements of all the programs that are under evaluation is created [8]. Then, we discard edges in such a graph between individual program statements whose similarities are below a certain threshold and, therefore, are noisy. To discover statement clusters, we process vertices in the graph based on their density, which serves to resolve "clear cuts" first, i.e., clusters of individual program statements that are homogenous in their semantic intent, leaving the difficult cases for latter stages. Finally, we avoid clusters that may contain different program statements belonging to the same program. The assumption is that each individual program statement has a semantic intent, and that semantic intent must be different within a certain program. Taking all of these into account, we propose a new statement clustering algorithm that, according to our experiments, is more efficient than the previous technique, and is able to cluster, in the worst case, more than 93% of the individual program statements.

The rest is as follows: preliminaries (Section 2), our proposed algorithm (Section 3), experimental results (Section 4), and conclusions (Section 5).

2 Preliminaries

We wish to analyze programs solving introductory programming assignments. Assume three programs presented in Figure 1 that solve the following assignment: Read the total number of test cases. Each test case contains the number of cars and a list of space-separated integers, each of which denotes the maximum

```
 1 Scanner sc = new
          Scanner(System.in);
 2 int t = sc.nextInt();
 3 while (t-- > 0) {
 4   int n = sc.nextInt();
 5   int[] mx = new int[n];
 6   for (int i = 0; i < n; i++)
 7     mx[i] = sc.nextInt();
 8   int x = mx[0];
 9   int s = 1;
10   for (int i = 1; i < n; i++) {
11     x = Math.min(mx[i], x);
12     if (x == mx[i])
13       s++;
14   }
15   System.out.println(s);
16 }
```

(a) p_1

```
 1 Scanner sc = new
          Scanner(System.in);
 2 int t = sc.nextInt();
 3 while (t-- > 0) {
 4   int n = sc.nextInt();
 5   int[] ar = new int[n];
 6   for (int i = 0; i < n; i++)
 7     ar[i] = sc.nextInt();
 8   int c = 1;
 9   int small = ar[0];
10   for (int i = 1; i < n; i++)
11     if (ar[i] <= small) {
12       c++;
13       small = ar[i];
14     }
15   System.out.println(c);
16 }
```

(b) p_2

```
 1 Scanner sc = new
          Scanner(System.in);
 2 int t = sc.nextInt();
 3 while (t-- > 0) {
 4   int n = sc.nextInt();
 5   int[] a = new int[n];
 6   for (int i = 0; i < n; i++)
 7     a[i] = sc.nextInt();
 8   int count = 1;
 9   for (int i = 1; i < n; i++)
10     if (a[i - 1] >= a[i])
11       count++;
12     else
13       a[i] = a[i - 1];
14   System.out.println(count);
15 }
```

(c) p_3

Fig. 1: Three programs solving CARVANS (codechef.com/problems/CARVANS)

speed of a car in the order they enter a straight segment. For each test case, output the number of cars which are moving at their maximum speed.

First, we model programs as program dependence graphs to be analyzed. A program is represented by a program dependence graph $G = (V, E, L_V, L_E)$ such that V is a set of vertices, each of which is a program statement, $E : V \to V$ is a bag of directed edges (there can be multiple edges connecting the same vertices), $L_V : V \to \mathcal{P}(\mathcal{V})$ is a vertex labeling function from each vertex to the power set of possible vertex labels $\mathcal{V}$, and $L_E : E \to \{Ctrl, Data\}$ is an edge labeling function that determines whether an edge is control (*Ctrl*) or data (*Data*).

The program dependence graph representing p_1 contains a vertex for each program statement, for instance, a vertex associated to line 7 where a position of a previously declared array is updated with the speed of a car. As a result of these operations, L_V of this specific vertex contains several labels, such as array access, assignment and `nextInt`. All these labels form $\mathcal{V}$ that help identify the semantics of the statements. Additionally, edges between vertices indicate the relationships between the statements in the code. For example, there is a *Ctrl* edge between the statement in line 6 (`for` loop) and the vertex previously discussed. This edge indicates that the statement is executed only if the condition of the loop is true. There is a *Data* edge between the statement that declares variable `i` and the statement that uses `i` to access the array.

A statement cluster C is a set of vertices (statements) that have the same semantic intent but can be implemented in different ways. Programs in Figure 1 initialize console using `Scanner`, which form a statement cluster. They read the total number of test cases using `nextInt`, which form another statement cluster.

The technique by Marin and Rivero [8] discovers statement clusters using a distance $d(v_i, v_j)$ between vertices v_i and v_j, which considers both $L_V(v_i)$ and $L_V(v_j)$ as well as their context. Having two program dependence graphs $G_i = (V_i, E_i, L_{V_i}, L_{E_i})$ and $G_j = (V_j, E_j, L_{V_j}, L_{E_j})$, it finds a correspondence between their vertices, a.k.a. alignment, $A : V_i \to V_j$ ($V_i \subseteq V_j$). To find A, a weighted bipartite graph $B = (V_i, V_j, E_B, W_B)$ is used, where $E_B : V_i \to V_j$ and $W_B : V_i \times V_j \to \mathcal{R}$ is an edge weight function such that $W_B((v_i, v_j)) =$

$1 - d(v_i, v_j)$. B is complete: every vertex in V_i is related to every vertex in V_j by an edge in E_B. An alignment A is a maximum weighted matching in B.

The next step consists of finding alignments between all programs under evaluation (for n programs, the total number of alignments is $1/2\,(n-1)\,n$). The union of the alignments form the pairwise alignment graph $P = (V_P, E_P, W_P)$, where V_P is the union of all vertices in the program dependence graphs, $E_P : V_P \times V_P$ is the set of edges such that each edge belongs to a specific alignment A (maximum weighted matching), and $W_P : V_i \times V_j \rightarrow \mathcal{R}$ is an edge weight function such that $W_P((v_i, v_j))$ corresponds to $W_B((v_i, v_j))$ in the bipartite graph B from which A is computed. Statement clusters are discovered by exploiting structural graph clustering over P, discerning between statement clusters, hubs and outliers. Hubs and outliers are vertices that are connected to other vertices in different statement clusters, but they do not belong themselves to any cluster. A hub is connected to vertices that belong to more than one statement cluster; an outlier is connected to vertices that belong to the same statement cluster.

In Figure 1a, the statement in line 11 in p_1 is a hub since it relates statements in the cluster formed by statements checking the current speed (lines 11 and 10 in p_2 and p_3, respectively), and statements in the cluster formed by statements updating the current minimum speed (lines 13 in both p_2 and p_3).

3 A new clustering algorithm

Low weights in alignments introduce noise [8]. These weights are the distance between two statements in an alignment graph. The algorithm to compute maximum weighted matchings focuses on large weights first, i.e., statements that are very related and, therefore, it is desirable to have correspondences between them. Unfortunately, since the algorithm aims to compute a maximum matching, there are certain vertices (the "leftovers") that are forced to match, even though their weight is low, i.e., they are probably not semantically related. We propose a user-defined threshold δ to avoid low weights in alignments as follows: let v_i and v_j be two vertices, (v_i, v_j) is discarded from an alignment A if $W_B((v_i, v_j)) < \delta$. By introducing δ, we expect to mitigate such noisy correspondences.

The processing order of the vertices may have an impact in the clustering process. Depending on which vertex is selected first for processing, statement clusters may contain a different set of statements. We propose to rely on the concept of the core number to determine such processing order. A k-core is a maximal subgraph of a graph in which all vertices have at least k neighbors [2]. The core number of v is the largest k such that v belongs to the k-core but not to the $(k+1)$-core. We thus process first vertices that are expected to be dense, i.e., they are semantically cohesive. These vertices should be "clear cuts" and the unraveling of posterior vertices should benefit from these early decisions.

A duplicated statement cluster contains at least two vertices that belong to the same program [8]. Since our goal is to detect statements across programs that have the same semantic intent, duplicated statement clusters are thus harmful. For instance, lines 11 and 12 in p_1 can be part of the same statement cluster. As

Algorithm 1: Mine statement clusters

```
Input: $P = (V_P, E_P, W_P)$, $\delta$ $\beta$, $\iota$
Output: A statement cluster function $X : V_P \to \mathbb{N}$
1  $E_P := E_P \setminus \{(v_i, v_j) \mid (v_i, v_j) \in E_P \wedge W_P((v_i, v_j)) < \delta\}$
2  $clnumber := 0$
3  foreach $v \in V_P$ sorted by core number do
4      $N := \hat{N}(v, P)$, $N' := \emptyset$
5      foreach $v \in N$ do
6          $N' := N' \cup \hat{N}(n, P)$
7      if $overlap(N, N') \geq \beta$ then
8          $X(v) := clnumber$
9          foreach $n \in N \cap N'$ do
10             $X(n) := clnumber$
11         $clnumber := clnumber + 1$
12     else
13         $X(v) := -1$
14 foreach $i \in ran\ X$ do
15     $V := \{v \mid X(v) = i\}$
16     if $|V| < \iota$ then
17         foreach $v \in V$ do
18             $X(v) := -1$
```

a result, we avoid duplicated statement clusters by defining a $\hat{N}(v, G)$ function that receives a vertex v and a graph G as input, and outputs all the neighbors of v in G such that every neighbor belongs to a different program than v. Forming clusters based on $\hat{N}(v, G)$ prevents duplicated statement clusters.

Algorithm 1 uses all of these ingredients to discover statement clusters.

4 Experiments

We evaluate our technique over six different introductory programming assignments. Five of them are from CodeChef (BUYING2, CARVANS, CONFLIP, LAPIN and STONES), which were also studied by Marin and Rivero [8]. The sixth assignment corresponds to P327A from Codeforces[3]. Table 1 presents our results, where $|P|$ represents the total number of correct programs available, $|V_P|$ is the total number of statements in the pairwise alignment graph, $|E_P|$ is the total number of edges that meet the weight threshold criterion ($W_P((v_i, v_j)) < \delta = .5$) in the pairwise alignment graph, $|\mathcal{C}|$ is the number of statement clusters discovered that meet both overlap and pervasiveness criteria based on $\beta = .8$, $|U|$ is the number of vertices that are non-clustered, Cov is the mean coverage of the program statements under evaluation, μ_V is the mean (and standard deviation) number of program statements that are contained in each statement cluster, and T is the total time in seconds to discover statement

[3] https://codeforces.com/problemset/status/327/problem/A

Table 1: Statement clusters and program coverage obtained for six different introductory programming assignments using $\delta = .5$, $\beta = .8$ and $\iota = .05\,|P|$

	$\|P\|$	$\|V_P\|$	$\|E_P\|$	$\|\mathcal{C}\|$	$\|U\|$	Cov	μ_V	T (s)
BUYING2	861	24,566	8,350,147	80	2,700	95.10%	273.33 ± 249.05	44
CARVANS	719	17,487	4,855,564	62	2,270	94.32%	245.44 ± 222.60	22
CONFLIP	1,203	26,685	12,155,327	68	3,555	93.77%	340.15 ± 340.27	75
LAPIN	561	18,126	3,856,061	107	1,890	94.77%	151.74 ± 135.09	21
P327A	750	22,384	6,948,266	93	1,116	96.26%	228.69 ± 201.79	34
STONES	152	4,312	252,174	98	405	96.67%	39.87 ± 40.03	1

clusters. We set ι to 5% of the total number of programs ($\iota = .05\,|P|$). The timings presented in Table 1 were obtained using commodity hardware.

Comparing our results with those obtained by Marin and Rivero [8], we observe that the coverage we obtain with the statement clusters computed by our technique significantly outperforms the previous coverage. For instance, in the LAPIN assignment, the previous coverage was above 30% based on 20 statement clusters. In our experiments, we obtain a coverage of 94% using 107 statement clusters. LAPIN has a fewer number of programs that have more implementation variability than other assignments. This can be determine by measuring the number of statement clusters as well as the average number of program statements per cluster. Because of this variability, the technique by Marin and Rivero [8] marks many program statements as outliers or hubs since there is no enough evidence to include them in a specific cluster. In our technique, these program statements are "forced" to belong to a given cluster, which will result in more diverse program statements clustered together.

5 Conclusions

Introductory programming learners need to receive constant feedback to improve their computational problem solving skills. It is currently a challenge to deliver feedback to the large number of learners in both traditional and online settings. Existing techniques focus on the automated analysis and delivery of feedback, and do not generally support an active role of the instructor neither in the feedback nor in its delivery. A promising direction to enable instructor-on-the-loop feedback delivery is to group program statements into clusters with a similar semantic intent. A previous technique focused on guaranteeing the semantic cohesiveness of the clusters rather than covering a large number of individual program statements. As a result, many program statements in the long tail are not clustered and, therefore, do not receive feedback. In this paper, we analyze several opportunities to increase the coverage of individual program statements with the goal of delivering feedback to the long tail. Our experiments show that we are able to cover, in the worst case, more than 93% of the program statements available for the assignments under evaluation. This increasing coverage comes with the penalty of less semantically-cohesive statement clusters.

References

1. Camp, T., Zweben, S.H., Buell, D.A., Stout, J.: Booming enrollments: Survey data. In: ACM Technical Symposium on Computing Science Education, SIGCSE 2016. pp. 398–399 (2016)
2. Cheng, J., Ke, Y., Chu, S., Özsu, M.T.: Efficient core decomposition in massive networks. In: IEEE International Conference on Data Engineering, ICDE 2011. pp. 51–62 (2011)
3. Fitzgerald, S., Hanks, B., Lister, R., McCauley, R., Murphy, L.: What are we thinking when we grade programs? In: ACM Technical Symposium on Computer Science Education, SIGCSE 2013. pp. 471–476 (2013)
4. Huang, J., Piech, C., Nguyen, A., Guibas, L.J.: Syntactic and functional variability of a million code submissions in a machine learning MOOC. In: Workshops at the International Conference on Artificial Intelligence in Education, AIED Workshops 2013 (2013)
5. Jawalkar, M.S., Hosseini, H., Rivero, C.R.: Learning to recognize semantically similar program statements in introductory programming assignments. In: ACM Technical Symposium on Computer Science Education, SIGCSE 2021. p. 1264 (2021)
6. Keuning, H., Jeuring, J., Heeren, B.: A systematic literature review of automated feedback generation for programming exercises. ACM Transactions on Computing Education (TOCE) **19**(1), 3:1–3:43 (2019)
7. Marin, V.J., Pereira, T., Sridharan, S., Rivero, C.R.: Automated personalized feedback in introductory Java programming MOOCs. In: IEEE International Conference on Data Engineering, ICDE 2017. pp. 1259–1270 (2017)
8. Marin, V.J., Rivero, C.R.: Clustering recurrent and semantically cohesive program statements in introductory programming assignments. In: ACM International Conference on Information and Knowledge Management, CIKM 2019. pp. 911–920 (2019)
9. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013. pp. 15–26 (2013)
10. Wang, K., Singh, R., Su, Z.: Search, align, and repair: data-driven feedback generation for introductory programming exercises. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018. pp. 481–495 (2018)