arXiv:1603.04584v1 [cs.SE] 15 Mar 2016

# Semi-Supervised Verified Feedback Generation

Shalini Kaleeswaran Anirudh Santhiar Aditya Kanade
Indian Institute of Science, Bangalore
{shalinik,anirudh_s,kanade}@csa.iisc.ernet.in

Sumit Gulwani
Microsoft Research, Redmond
sumitg@microsoft.com

## ABSTRACT

Students have enthusiastically taken to online programming lessons and contests. Unfortunately, they tend to struggle due to lack of personalized feedback when they make mistakes. The overwhelming number of submissions precludes manual evaluation. There is an urgent need of program analysis and repair techniques capable of handling both the scale and variations in student submissions, while ensuring quality of feedback.

Towards this goal, we present a novel methodology called *semi-supervised verified feedback generation*. We cluster submissions by solution strategy and ask the instructor to identify or add a correct submission in each cluster. We then verify every submission in a cluster against the instructor-validated submission in the same cluster. If faults are detected in the submission then feedback suggesting fixes to them is generated. Clustering reduces the burden on the instructor and also the variations that have to be handled during feedback generation. The verified feedback generation ensures that only correct feedback is generated.

We have applied this methodology to iterative dynamic programming (DP) assignments. Our clustering technique uses features of DP solutions. We have designed a novel *counter-example guided feedback generation* algorithm capable of suggesting fixes to all faults in a submission. In an evaluation on 2226 submissions to 4 problems, we could generate *verified* feedback for 1911 (85%) submissions in 1.6s each on an average. Our technique does a good job of reducing the burden on the instructor. Only one submission had to be manually validated or added for every 16 submissions.

## 1. INTRODUCTION

Programming has become a much sought-after skill for superior employment in today's technology-driven world [1]. Students have enthusiastically taken to online programming lessons and contests, in the hope of learning and improving programming skills. Unfortunately, they tend to struggle due to lack of personalized feedback when they make mistakes. The overwhelming number of student submissions precludes manual evaluation. There is an urgent need of automated program analysis and repair techniques capable of handling both the scale and variations in student submissions, while ensuring quality of feedback.

A promising direction is to cluster submissions, so that the instructor provides feedback for a representative from each cluster which is then propagated automatically to other submissions in the same cluster [20, 41]. This provides scalability while keeping the instructor efforts manageable. Many novel solutions have been proposed in recent times to enable clustering of programs. These include syntactic or test-based similarity [15, 44, 20, 13, 12], co-occurrence of code phrases [36] and vector representations obtained by deep learning [40, 41, 33]. However, clustering can be correct only in a *probabilistic* sense. Thus, these techniques cannot guarantee that the feedback provided manually by the instructor, by looking only at some submissions in a cluster, would indeed be suitable to all the submissions in that cluster. As a result, some submissions may receive incorrect feedback. Further, if submissions that have similar mistakes end up in different clusters, some of them may not receive the suitable feedback. Instead of helping, these drawbacks can cause confusion among the students.

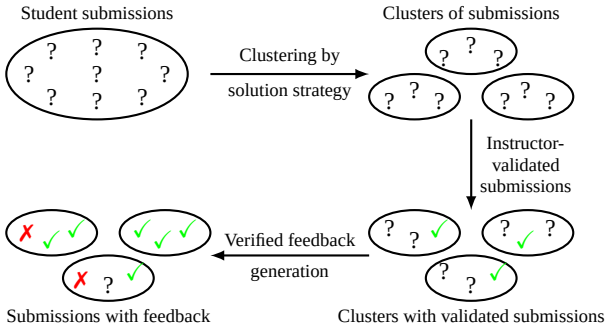



**Figure 1: Semi-supervised verified feedback generation: ✓ is a submission verified to be correct, ✗ is a faulty submission for which feedback is generated and ? is an unlabeled submission for which feedback is not generated.**

To overcome these drawbacks, we propose a novel methodology in which clustering of submissions is followed by an automated feedback generation phase grounded in formal verification. Figure 1 shows our methodology, called *semi-supervised verified feedback generation*. Given a set of unlabeled student submissions, we first cluster them by similarity of solution strategies and ask the instructor to identify[1] a correct submission in each cluster. If none exists, the instructor adds a correct solution similar to the submissions in the cluster; after which we do clustering again. In the next phase, each submission in a cluster is *verified against the instructor-validated submission* in the same cluster. If any faults are detected in the submission then feedback suggesting fixes to them is generated. Because program equivalence checking is an undecidable problem, it may not be possible to generate feedback for every submission. We let the instructor evaluate such submissions manually. This is better than propagating unverified or incorrect feedback indiscriminately.

The proposed methodology has several advantages. First, it uses unsupervised clustering to reduce the burden on the instructor. The supervision from the instructor comes in the form of identifying a correct submission per cluster. Second, the verification phase

[1]To minimize instructor's efforts further, we discuss strategies to suggest potentially correct submissions to the instructor.

```
1  void main() {
2    int i, j, n, max;
3    scanf("%d", &n); // Input
4    int m[n][n], dp[n][n]; // dp is the DP array
5    for (i = 0; i < n; i++)
6      for (j = 0; j <= i; j++)
7        scanf("%d", &m[i][j]); // Input
8    dp[0][0] = m[0][0]; // Initialization
9    for (i = 1; i < n; i++) {
10     for (j = 0; j <= i; j++) {
11       if (j == 0)
12         dp[i][j] = dp[i-1][j] + m[i][j]; // Update
13       else if (j == i)
14         dp[i][j] = dp[i-1][j-1] + m[i][j]; // Update
15       else if (dp[i-1][j] > dp[i-1][j-1])
16         dp[i][j] = dp[i-1][j] + m[i][j]; // Update
17       else dp[i][j] = dp[i-1][j-1] + m[i][j]; // Update
18     }
19   }
20   max = dp[n-1][0];
21   for (i = 1; i < n; i++)
22     if (dp[n-1][i] > max) max = dp[n-1][i];
23   printf("%d", max); // Output
24 }
```

**Figure 2: A correct submission for the matrix path problem.**

```
1  int max(int a, int b) {
2    return a > b ? a : b;
3  }
4  int max_arr(int arr[]) {
5    int i, max;
6    max = arr[0];
7    for (i = 0; i < 100; i++)
8      if (arr[i] > max) max = arr[i];
9    return max;
10 }
11 int main() {
12   int n, i, j, A[101][101], D[101][101]; // D is the DP array
13   scanf("%d", &n);// Input
14   for (i = 0; i < n; i++)
15     for (j = 0; j <= i; j++)
16       scanf("%d", &A[i][j]); // Input
17   D[0][0] = A[0][0] ; // Initialization
18   for (i = 1; i < n; i++)
19     for (j = 0; j <= i; j++)
20       D[i][j] = A[i][j] + max(D[i-1][j], D[i-1][j-1]); // Update
21   int ans = max_arr(D[n-1]);
22   printf("%d", ans); // Output
23   return 0;
24 }
```

**Figure 3: A faulty submission belonging to the same cluster as the correct submission in Figure 2.**

**In the declaration:** 1) Types of A and D should be int[n][n]
**In the update:**
2) Under guard j == 0,
compute D[i][j] = D[i-1][j] + A[i][j]
instead of D[i][j] = A[i][j] + (D[i-1][j]>D[i-1][j-1] ?
D[i-1][j] : D[i-1][j-1]).
3) Under guard (j != 0 && j == i),
compute D[i][j] = D[i-1][j-1] + A[i][j]
instead of D[i][j] = A[i][j] + (D[i-1][j]>D[i-1][j-1] ?
D[i-1][j] : D[i-1][j-1]).
**In the output:** 4) Under guard true, compute maximum
over D[n-1][0],...,D[n-1][n-1] instead of D[n-1][0],...,D[n-1][99].

**Figure 4: The auto-generated feedback for the submission in Figure 3 by verifying it against the submission in Figure 2.**

complements clustering by providing *certainty* about correctness of feedback. Clustering helps by reducing the variations required to be handled during feedback generation. It would be difficult to check equivalence of very dissimilar submissions, *e.g.*, an iterative solution against a recursive solution. Since only submissions that are similar (*i.e.*, belong to the same cluster) are compared, there is a higher chance of making equivalence checking work in practice. Third, feedback is generated for each submission separately so it is personalized. This is superior to propagating a manually-written feedback indiscriminately to all the submissions in a cluster.

To demonstrate the effectiveness of this methodology, we apply it to iterative dynamic programming assignments. Dynamic programming (DP) [8] is a standard technique taught in algorithms courses. Shortest-path and subset-sum are among the many problems that can be solved efficiently by DP. We design features that characterize the DP strategy and extract them from student submissions by static analysis and pattern matching. The features include the types of the DP arrays used for memoizing solutions to sub-problems, and how the sub-problems are solved and reused iteratively. The set of features we need is small and all features are discrete valued. Therefore, our clustering approach is very simple and works directly by checking equality of feature values.

We also propose a novel feedback generation algorithm called *counter-example guided feedback generation*. The equivalence between two submissions is checked using syntactic simplifications and satisfiability-modulo-theories (SMT) based constraint solving. If an equivalence check fails, our algorithm uses the counter-example generated by the SMT solver to refine the equivalence query. This process terminates when our algorithm proves the equivalence, or is unable to refine the query. A trace of refinements leading to a logically valid equivalence query constitutes the feedback.

As an example, consider the matrix path problem[2] taken from a popular programming contest site CodeChef. A lower triangular matrix of n rows is given. Starting at a cell, a path can be traversed in the matrix by moving either directly below or diagonally below to the right. The objective is to find the maximum weight among the paths that start at the cell in the first row and first column, and end in any cell in the last row. The weight of a path is the sum of all cells along that path. Figure 2 shows an example correct submission to this problem and Figure 3 shows a faulty submission. The two programs are syntactically and structurally quite different but both use 2D integer arrays for memoization and iterate over them in the same manner. Our clustering technique therefore puts them into the same cluster. This avoids unnecessarily creating many clusters but requires a more powerful feedback generation algorithm that can handle stylistic variations between submissions, *e.g.*, Figure 3 uses multiple procedures whereas Figure 2 uses only one.

Our algorithm automatically generates the feedback in Figure 4 for the faulty submission by verifying it against the correct submission. The first correction suggests that the submission should use array sizes as int[n][n] instead of hardcoded value of int[101][101]. The update to the DP array D at line 20 in Figure 3 misses some corner cases for which our algorithm generates corrections #2 and #3 above. The computation of output at line 22 should use the correct array bounds as indicated by correction #4. This is a comprehensive list of changes to correct the faulty submission.

We have implemented our technique for C programs and evaluated it on 2226 student submissions to 4 problems from CodeChef. On 1911 (85%) of them, we could generate feedback by either verifying them to be correct, or identifying faults and fixes for them. In addition to faults in wrong answers, we also found faults in 265 submissions accepted by CodeChef as correct answers! Like most online contest sites, CodeChef uses test-based evaluation. Our

[2] http://www.codechef.com/problems/SUMTRIAN

static verification technique has a qualitative advantage over the test-based approach of online judges. The submissions come from 1860 students from over 250 different institutes and are therefore representative of diverse backgrounds and coding styles. Even then, the number of clusters ranged only from 2–80 across the problems. Our technique does a good job of reducing the burden on the instructor. On an average, using one manually validated or added submission, we generated verified feedback on 16 other submissions. We had to add only 7 correct solutions manually. While our technique generated feedback automatically for 1911 submissions, the remaining 315 (15%) submissions require manual evaluation. Our technique is fast and on an average, took 1.6s to generate feedback for each submission.

Work on feedback generation so far has focused on *introductory programming* assignments [46, 17, 47, 21]. In comparison, we address the challenging class of *algorithmic* assignments, in particular, that of dynamic programming. The program repair approaches for developers [27, 37, 24, 31, 29] deal with one program at a time. We work with all student submissions simultaneously. To do so, we propose a methodology inspired by both machine learning and verification. Unlike the developer setting, we have the luxury of calling upon the instructor to identify or add correct solutions. We exploit this to give complete and correct feedback but then our technique must solve the challenging (and in general, undecidable) problem of checking semantic equivalence of programs.

The salient contributions of this work are as follows:

- We present a novel methodology of clustering of submissions followed by program equivalence checking within each cluster. This methodology can pave way for practical feedback generation tools capable of handling both the scale and variations in student submissions, while minimizing the instructor's efforts and ensuring quality of feedback.
- We demonstrate that this methodology is effective by applying it to the challenging class of iterative DP solutions. We design a clustering technique and a counter-example guided feedback generation algorithm for DP solutions.
- We experimentally evaluate the technique on 2226 submissions to 4 problems and generate *verified* feedback for 85% of them. We show that our technique does not require many inputs from the instructor and runs efficiently.

# 2. DETAILED EXAMPLE

We now explain in details how our technique handles the motivating example from the previous section.

## 2.1 Clustering Phase

The two submissions in Figure 2 and Figure 3 are syntactically and structurally quite different. Our technique extracts features of the solution strategy in a submission. These features are more abstract than low-level syntactic or structural features and put the superficially dissimilar submissions into the same cluster.

The solution strategy of a DP program is characterized by the DP recurrence being solved [8]. The DP recurrence for the correct submission in Figure 2 is as follows:

$$\texttt{dp[i][j]} = \begin{cases} \texttt{m[0][0]} & \text{if } \texttt{i}=0, \texttt{j}=0 \\ \texttt{dp[i-1][j]+m[i][j]} & \text{if } \texttt{i}\neq 0, \texttt{j}=0 \\ \texttt{dp[i-1][j-1]+m[i][j]} & \text{if } \texttt{j}=\texttt{i}\neq 0 \\ \texttt{max(dp[i-1][j],dp[i-1][j-1])+m[i][j]} & \text{otherwise} \end{cases}$$

where `dp` is the DP array, `m` is the input matrix of `n` rows, and `i` goes from 0 to `n-1`, `j` goes from 0 to `i` and `max` returns the maximum of the two numbers. The DP recurrence of the submission in Figure 3 is similar but misses the second and third cases above.

Comparing the recurrences directly would be ideal but extracting them is not easy. Students can implement a recurrence formula in different imperative styles. They may use multiple procedures (as in Figure 3) and arbitrary temporary variables to hold intermediate results. Rather than attempting to extract the precise recurrence, we extract some features of the solution to find submissions that use similar solution strategies. For this, our analysis identifies and labels the DP arrays used in each submission. It also identifies and labels key statements that 1) read inputs, 2) initialize the DP array, 3) update the DP array elements using previously computed array elements, and 4) generate the output. The comments in Figure 2 and Figure 3 identify the DP arrays and key statements.

We call a loop which is not contained within any other statement as a *top-level loop*. For example, the loop at lines 9–19 in Figure 2 is a top-level loop but the loop at lines 10–18 is not. More generally, a statement which is not contained within any other statement is a *top-level statement*. The *features extracted by our technique* and their values for the submission in Figure 2 are as follows:

1. Type and the number of dimensions of the DP array: $\langle \texttt{int}, 2 \rangle$
2. Whether the input array is reused as the DP array: No
3. The number of top-level loops which contain update statements for the DP array: 1
4. For each top-level loop containing updates to the DP array,
   (a) The loop nesting depth: 2
   (b) The direction of loop indices: $\langle +, + \rangle$ (indicating that the respective indices are incremented by one in each iteration of the corresponding loops)
   (c) The DP array element updated inside the loop: `dp[i][j]`

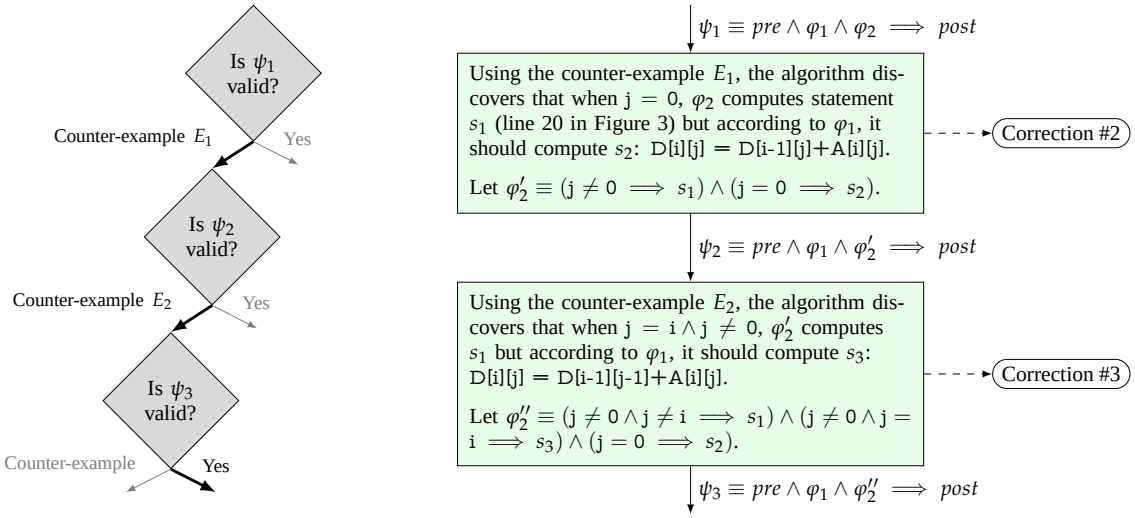
Extracting these features requires static analysis and syntactic pattern matching. The most challenging part is identifying which array serves as a DP array. An array which is defined in terms of itself is identified as a DP array since in the DP recurrence, the DP array appears on both sides. However, a student may use some temporary variables to store intermediate results of DP computation and pass values across procedures. As explained in Section 3.1, we track data dependences inter-procedurally. In Figure 3, the array elements `D[i-1][j]` and `D[i-1][j-1]` are passed to the procedure `max`. Through inter-procedural analysis, our technique infers that the return value of `max` is indeed defined in terms of these arguments and hence, `D` is defined in terms of itself at line 20. Thereby, it discovers that `D` is a DP array.

The submission in Figure 3 yields similar feature values and is clustered along with the submission in Figure 2. Note that our objective is not to use clustering to distinguish between correct and incorrect submissions. We therefore do not encode the exact nature of the initialization or update of the DP array in the features. Analyzing these is left to the verification and feedback phase.

## 2.2 Verification and Feedback Phase

After clustering, suppose the instructor identifies the submission in Figure 2 as correct. Our objective is to verify the submission in Figure 3 against it and suggest fixes if any faults are found. For brevity, we will refer to the submission in Figure 2 as the *reference* and the submission in Figure 3 as the *candidate*.

By analyzing the sequence in which inputs are read, our technique infers that the candidate uses two input variables: an integer variable `n` and a 2D integer array `A`, where `n` is read first (line 13) and `A` second (line 16). Their types respectively match the types of the input variables `n` and `m` of the reference except that the candidate uses a hardcoded array size for `A`. Both submissions use 2D




**a) Successive equivalence queries and results** **b) Refinement steps and corrections suggested**

**Figure 5: Steps of the verified feedback generation algorithm for the DP update in the faulty submission in Figure 3.**

integer DP arrays but the candidate hardcodes array size of the DP array D also. While the reference can handle arbitrarily large input matrices, the candidate can handle input matrices only up to size $101 \times 101$. For smaller input matrices, the reference is more space efficient than the candidate. Our technique therefore emits correction #1 in Figure 4 suggesting size declarations for A and D.

We check equivalence of matching code fragments of the two submissions one-by-one. The matching code fragments are easy to identify given the statement labels computed during feature extraction. For our example, line 20 of the candidate is an "update" statement and lines 12, 14, 16 and 17 of the reference are also "update" statements. Therefore, the top-level loop (say $L_2$) at lines 18–20 of the candidate matches the top-level loop (say $L_1$) at lines 9–19 of the reference. The question is whether they are equivalent.

We check equivalence of the loop headers first. The input variables n in both submissions correspond to each other and are not re-assigned before they are used in the respective loop headers. Therefore, the loop headers of $L_1$ and $L_2$ are equivalent. Thus, the corresponding loop indices are equal in each iteration.

To check equivalence of loop bodies, our algorithm formulates an *equivalence query* $\psi_1$ which asserts that in an (i,j)th iteration, if the two DP arrays are equal at the beginning then they are equal at the end of the iteration. The equivalence query is of the form:

$$\psi_1 \equiv pre \wedge \varphi_1 \wedge \varphi_2 \implies post$$

where 1) *pre* encodes the equality of DP arrays, loop indices and input variables at the beginning of the iteration, and the lower and upper bounds on the loop index variables in the reference, 2) *post* encodes the equality of DP arrays at the end of the iteration (we syntactically check that input variables are not changed), 3) $\varphi_1$ is a formula encoding the statements in the loop body of the reference, and 4) $\varphi_2$ is a formula encoding the statements in the loop body of the candidate. Converting a loop-free sequence of statements into a formula is straightforward. For example, an if-statement such as if(p) x = e is converted to a guarded equality constraint p : $x' = e$ where $x'$ is a fresh variable. The predicates in an if-else statement are propagated so as to make the guards mutually disjoint and finally, the conjunction of all guarded equality constraints is taken. We defer other technical details to Section 3.2.

As shown in Figure 5.a, the algorithm checks whether $\psi_1$ is a logically valid formula. The SMT solver finds the following counter-example $E_1$ which shows that the formula is not valid:

j = 0, dp[i][j] = 1, D[i][j] = 2, m[i][j] = A[i][j] = 1,
dp[i-1][j] = D[i-1][j] = 0, dp[i-1][j-1] = D[i-1][j-1] = 1

Following the usual convention, we use $=$ as equality in formulae and use == as equality in code. Similarly, $\neq$ and != denote disequality symbols in formulae versus code.

Our algorithm, called *counter-example guided feedback generation* algorithm, uses $E_1$ to localize the fault in the candidate. It first identifies which guards are satisfied by the counter-example in the candidate and the reference, and whether they are equivalent. The guard $j = 0$ is satisfied in $\varphi_1$ and the implicit guard *true* for line 20 is satisfied in $\varphi_2$. Since they are not equivalent, the algorithm infers that the faulty submission is missing a condition. On the contrary, if the guards turn out to be equivalent, the fault is localized to the assignment statement. It then derives a formula $\varphi_2'$ given in Figure 5.b which lets the candidate compute line 20 under the guard j!=0 and makes it compute D[i][j] = D[i-1][j]+A[i][j] under j==0. This assignment statement is obtained from the assignment at line 12 under the guard j==0 of the reference in Figure 2 by substituting the variables from the candidate. The algorithm records this refinement in the form of correction #2 of Figure 4.

As shown in Figure 5.a, it checks validity of $\psi_2 \equiv pre \wedge \varphi_1 \wedge \varphi_2' \implies post$ obtained by replacing $\varphi_2$ (the encoding of candidate's loop body) by $\varphi_2'$ (defined in Figure 5.b). This results in a counter-example $E_2$ using which the algorithm discovers the missing case of j==i and generates correction #3 of Figure 4. For brevity, we do not show the counter-example $E_2$. A refined equivalence query $\psi_3$ shown in Figure 5.b is computed. As shown in Figure 5.a, this formula is valid and establishes that the faults in the candidate can be fixed using the synthesized feedback.

The input and initialization parts of the two submissions are found to be equivalent. In our experiments, we observed certain repeating iterative patterns. The computation of a maximum over an array in lines 6–8 in Figure 3 is one such example. We encode syntactic patterns to lift these to certain predefined functions. We define _max which takes the first and the last elements of a contiguous array segment as arguments and returns the maximum over the array segment. In Figure 3, the output expression in terms of _max is _max(D[n-1][0], D[n-1][99]) and in Figure 2, the output is _max(dp[n-1][0], dp[n-1][n-1]). A syntactic comparison between the two leads to correction #4 in Figure 4.

# 3. TECHNICAL DETAILS

We now explain our approach for clustering submissions, and the algorithm for verified feedback generation.

## 3.1 Clustering by Solution Strategy

The first phase of our technique is to cluster submissions by the solution strategy so that each cluster can be analyzed separately.

### *3.1.1 Feature Design*

Section 2.1 has already introduced the features of a submission that we use. Typically, in machine learning, a large number of features are obtained and then the learning algorithm finds the important ones (called feature selection). In our case, since the domain is well-understood, we design a small number of suitable features that provide enough information about the solution strategy.

In particular, we cluster two submissions together if 1) they use the same type and dimensions for the DP arrays, 2) either both use DP arrays distinct from the input arrays or not, and 3) there is a one-to-one correspondence between top-level loops which contain DP update statements — the loops should have the same depth, direction and the DP array element being updated. Two submissions in the same cluster can differ in all other aspects.

The rationale behind these features is simple: Checking equivalence of two submissions which use the same types of DP arrays and similar DP update loops is easier than if they do not share these properties. For example, the subset-sum problem can be solved by using either a boolean DP array or an integer DP array, but the two implementations are hard to compare algorithmically. Recall the matrix path problem stated in the Introduction. Consider a submission which traverses the matrix from top-to-bottom and another which traverses it from bottom-to-top. Using one to validate the other is difficult and perhaps, even undesirable. The features of DP update loops will prevent these submissions from being part of the same cluster. Imposing further restrictions (by adding more features) can make verification simpler but will increase the burden on the instructor by creating additional clusters.

The feature 4.c described in Section 2.1 requires a bit more explanation. We want to get the DP array element being updated inside each loop containing a DP update statement. If two submissions use different names for DP arrays and loop indices, we cannot compare them. To compare them across submissions, our technique uses canonical names for them: `dp` for the DP array and loop indices `i`, `j`, `k`, etc. from the outer to inner loops. If a submission uses multiple DP arrays then we assign subscripts to `dp`.

### *3.1.2 Feature Extraction*

Identifying input statements and variables is simple. We look for the common C library functions like `scanf`. The case of output statements is similar. A variable `x` is identified as a loop index variable if 1) `x` is a scalar variable, 2) `x` is initialized before the loop is entered, 3) `x` updated inside the loop and 4) `x` is used in the loop guard. Identifying DP arrays requires more subtle analysis discussed below. We call DP arrays, input variables and loop indices in a submission as *DP variables*. All other variables are called temporary variables.

To eliminate the use of a temporary variable `x` at a control location $l$, we compute a set of guarded expressions

$$\{\mathtt{g_1} : \mathtt{e_1}, \ldots, \mathtt{g_n} : \mathtt{e_n}\}$$

where the guards and expressions are defined only over DP variables, and the guards are mutually disjoint. We denote this set by $\Sigma(l, \mathtt{x})$ and call $\Sigma$ the *substitution store*. Semantically, if $\mathtt{g_k} : \mathtt{e_k} \in \Sigma(l, \mathtt{x})$ then `x` and $\mathtt{e_k}$ evaluate to the same value at $l$ whenever $\mathtt{g_k}$ evaluates to true at $l$. The substitution store $\Sigma$ is lifted in a natural manner to expressions and statements. For instance, for an assignment statement $s \equiv \mathtt{x = e}$, $\Sigma(l, s) = \{\mathtt{g_1} : \mathtt{x = e_1}, \ldots, \mathtt{g_n} : \mathtt{x = e_n}\}$ where $\{\mathtt{g_1} : \mathtt{e_1}, \ldots, \mathtt{g_n} : \mathtt{e_n}\} = \Sigma(l, \mathtt{e})$.

Gulwani and Juvekar [16] developed an *inter-procedural* backward symbolic execution algorithm to compute symbolic bounds on values of expressions. While we are not interested in the bounds, the equality mode of their algorithm suffices to compute substitution stores. We refer the reader to [16] for the details.

To determine whether a statement $s$ at location $l$ is an initialization or an update statement, we perform pattern matching over $\Sigma(l, s)$. If the same array appears on both sides of an assignment statement then the array is identified as a DP array and the statement is labeled as an update statement. A statement where the LHS is a DP array and RHS is an input variable or a constant is labeled as an initialization statement. In $\Sigma(l, s)$, the temporary variables in $s$ are replaced by the guarded expressions from the substitution store. This makes the labeling part of our tool robust even in presence of temporaries and procedure calls. For example, suppose we have `t = x[i-1]; x[i] = t;`. The second statement can be identified as an update statement through pattern matching only if we substitute `x[i-1]` in place of `t` on the RHS of the statement. In general, $\Sigma(l, s)$ may contain multiple guarded statements. If $\Sigma(l, s) = \{s_1, \ldots, s_n\}$, we require that all of $s_1, \ldots, s_n$ satisfy the same pattern and get the same label. Extracting the feature values is now straightforward.

### *3.1.3 Clustering and Identifying Correct Submissions*

All our features are discrete valued. Therefore, our clustering algorithm is very simple and works directly by checking equality of feature values. Once the clustering is done, we ask the instructor to identify a correct submission from each cluster. To reduce instructor's efforts, we can employ some heuristics to rank candidates in a cluster and present them one-by-one to the instructor. For example, we can use a small set of tests or majority voting on some other features of submissions like the loop bounds of update loops.

The instructor can accept a submission as correct or add a modified version of an existing submission. If none of this is possible, the instructor can write a correct solution similar to the solutions in the cluster. If a new submission is added, we perform clustering again. The instructor may have correct solutions from a previous offering of the course if the assignment is repeated from a previous offering. The instructor can add them to the dataset even before we apply clustering to the submissions.

## 3.2 Verified Feedback Generation

Once the submissions are clustered and the instructor has identified a valid submission for each cluster, we proceed to the verified feedback generation phase. We check semantic equivalence of a submission from a cluster (called the *candidate*) with the instructor-validated submission from the same cluster (called the *reference*).

### *3.2.1 Variable and Control Correspondence*

Program equivalence checking is an undecidable problem. In practice, a major difficulty is establishing correspondence between variables and control locations of the two programs [34]. We exploit the analysis information computed during feature extraction to solve this problem efficiently.

Let $\sigma$ be a one-to-one function, called a *variable map*. $\sigma$ maps the input variables and DP arrays of the reference to the corresponding ones of the candidate. To obtain a variable map, the input variables of the two submissions are matched by considering the order in which they are read and their types. The DP arrays are matched based on their types. If there are multiple DP arrays

with the same type in both submissions then all type-compatible pairs are considered. This generates a set of potential variable maps and equivalence checking is performed for each variable map separately. The one which succeeds and produces the minimum number of corrections is used for communicating feedback to the student. In equivalence checking, we eliminate the occurrences of temporary variables using the substitution store computed during feature extraction. We therefore do not need to derive correspondence between temporary variables – which simplifies the problem greatly.

The feature extraction algorithm labels the input, initialization, update and output statements of a submission. We refer to these statements as *labeled statements*. The labeled statements give an easy way to establish control correspondence between the submissions. We now use the notion of top-level statements defined in Section 2.1. Let $\hat{R} = [s_1^1, \ldots, s_1^k]$ be the list of all top-level statements of the reference such that 1) each statement in $\hat{R}$ contains at least one labeled statement and 2) the order of statements in $\hat{R}$ is consistent with their order in the reference submission. It is easy to see that the top-level statements in a submission are totally ordered. Let $\hat{C} = [s_2^1, \ldots, s_2^n]$ be the similar list for the candidate submission. Without loss of generality, from now on, we assume that there is only one DP array in a submission and the top-level statements are (possibly nested) loops.

A (top-level) loop in $\hat{R}$ or $\hat{C}$ may contain multiple statements which have different labels. For example, a loop may read the input and also update the DP array. We call it a *heterogeneous* loop. If a loop reads two different input variables then also we call it a heterogeneous loop. Heterogeneous loops make it difficult to establish control correspondence between the statement lists $\hat{R}$ and $\hat{C}$. Fortunately, it is not difficult to canonicalize the statement lists using *semantics-preserving* loop transformations, well-known in the compilers literature [3]. Our algorithm first does loop splitting to split a heterogeneous loop into different homogeneous loops. It then does loop merging to coalesce different loops operating on the same variable. Specifically, it merges two loops reading the same input array. It also merges loops performing initialization to the same DP array. During merging, we ensure that there is no loop in between the merged loops such that it reads from or writes to the same variable or array as the merged loops. In our experience, in most cases, these transformations work because loops reading inputs or performing initialization of DP arrays do *not* have loop-carried dependences or ad-hoc dependences between loops.

In contrast, by definition, loops performing DP updates do have loop-carried dependences. We therefore do not attempt loop merging for such loops. The feature 3 in Section 2.1 tracks the number of loops containing DP updates. Therefore, two submissions in the same cluster already have the same number of loops containing DP updates. Thus, clustering helps in reducing the variants that need to be considered during feedback generation.

Let $R$ and $C$ be the resulting statement lists for the reference and candidate submissions respectively. If they have the same length and at each index $i$, the $i$th loops in the two lists 1) operate on the variables related by a variable map $\sigma$, 2) the statements operating on the variables carry the same labels and 3) the loops have the same nesting depth and directions then we get the *control correspondence* $\pi : R \to C$. If our algorithm fails to compute variable or control correspondence for the candidate then it exits without generating feedback, implicitly delegating it to the instructor.

### 3.2.2 Equivalence Queries

Let $s_1'$ and $s_2'$ be the top-level loops from the reference and the candidate such that $\pi(s_1') = s_2'$. We first use the substitution map computed during feature extraction to eliminate temporary variables and procedure calls in $s_1'$ and $s_2'$ by equivalent guarded expressions over only DP arrays, loop indices and input variables. Let $s_1 = \Sigma(l_1, s_1')$ and $s_2 = \Sigma(l_2, s_2')$ where $l_1$ and $l_2$ are control locations of $s_1'$ and $s_2'$.

We formulate an equivalence query $\Phi$ for the iteration spaces of $s_1$ and $s_2$. Let *corr* be the correspondence between the input variables, DP arrays, and loop indices of $s_1$ and $s_2$ at the matching nesting depths. We define $iter_1$ to be the range of the loop indices in $s_1$ and $guards_1$ to be the disjunction of all guards present in the loop body of $s_1$. Similarly, we have $iter_2$ and $guards_2$ for $s_2$. The equivalence query $\Phi$ is defined as follows:

$$\Phi \equiv corr \implies (iter_1 \wedge guards_1 \iff iter_2 \wedge guards_2)$$

This query provides more flexibility than using direct syntactic checking between the loop headers. For example, suppose $s_1$ is `for(i=1, i<=n, i++){true: s}` and $s_2$ is `for(i'=0, i'<=n, i'++){i' > 0: s'}`. $s_1$ executes `s` for $1 \leq$ `i` $\leq$ `n` and $s_2$ also executes `s'` for $1 \leq$ `i'` $\leq$ `n`. A syntactic check will end up concluding that $s_2$ executes one additional iteration when `i'` is 0. But our equivalence query establishes equivalence between the iteration spaces as desired.

The formulation of the query $\Psi$ to establish equivalence between loop bodies of $s_1$ and $s_2$ is as discussed in Section 2.2. Even though the submissions use arrays, we eliminate them from the queries. A loop body makes use of only a finite number of symbolic array expressions. We substitute each unique array expression in a query by a scalar variable while encoding correspondence between the scalar variables in accordance with the variable map $\sigma$. We overcome some stylistic variations when the order of operands of a commutative operation differs between the two submissions. For example, say $s_1$ uses `x[i+j]` and $s_2$ uses `y[b+a]` such that $\sigma($`x`$) =$ `y`, $\sigma($`i`$) =$ `a` and $\sigma($`j`$) =$ `b`. The expressions `i+j` and `b+a` are not identical under renaming but are equivalent due to commutativity. To take care of this, we force a fixed ordering among variables in the two submissions for commutative operators. Sometimes, the instructor may include some constraints over input variables as part of the problem statement. In the equivalence queries, our algorithm takes input constraints into account and also adds array bounds checks. We omit these details due to space limit.

### 3.2.3 Counter-Example Guided Feedback Generation

Algorithm 1 is our counter-example guided feedback generation algorithm. Its input is a list $Q$ of equivalence queries where each query $(\Phi_i, \Psi_i)$ corresponds to the $i$th statements in the two submissions. $\Phi_i$ encodes the equivalence of iteration spaces and $\Psi_i$ of the loop bodies. If the $i$th statements are not loops, $\Phi_i$ is *true* and $\Psi_i$ just checks equivalence of the loop-free statements. The output of the algorithm is a list of corrections to the candidate submission.

Algorithm 1 iterates over the query list (line 1). For a query $(\Phi_i, \Psi_i)$, it first checks whether $\Phi_i$ is (logically) valid or not. If it is not then the algorithm suggests a correction to make the iteration spaces of the $i$th statements (loops) of the two submissions equal (lines 2-4). It then enters a refinement loop for $\Psi_i$ at lines 7-23.

During each iteration of the refinement loop, it checks whether $\Psi_i$ is valid. If yes, it exits the loop (line 9). Otherwise, it gets a counter-example $\alpha$ from the SMT solver and finds the guarded statements that are satisfied by $\alpha$. Let $g_1 : s_1 \in \varphi_1$ and $g_2 : s_2 \in \varphi_2$ be those statements (line 11). The formulae $\varphi_1$ and $\varphi_2$ correspond to the encodings of the loop bodies of the reference and the candidate respectively. Note that the conversion of statements to guarded equality constraints (Section 2.2) ensures that the guards within $\varphi_1$ and within $\varphi_2$ are pairwise disjoint.

Let $\hat{\sigma}$ be the variable map which is same as the variable correspondence $\sigma$ but augmented with the correspondence between loop

**Algorithm 1:** Algorithm GENFEEDBACK

**Input**: A list $Q = [(\Phi_1, \Psi_1), \ldots, (\Phi_k, \Psi_k)]$ of equivalence queries
**Output**: A list of corrections to the candidate submission

1 **foreach** $(\Phi_i, \Psi_i) \in Q$ **do**
2  **if** $\exists \alpha \not\models \Phi_i$ **then**
3  Suggest corrections to make the iteration spaces of the $i$th statements of the two submissions equal
4  **end**
5  Let $\Psi_i \equiv pre \wedge \varphi_1 \wedge \varphi_2 \implies post$
6  $k \leftarrow 0$
7  **repeat**
8  $k \leftarrow k + 1$
9  **if** $\models \Psi_i$ **then break else**
10  Let $\alpha \not\models \Psi_i$ be a counter-example
11  Let $g_1 : s_1 \in \varphi_1$ and $g_2 : s_2 \in \varphi_2$ s.t. $\alpha \models g_1$ and $\alpha \models g_2$
12  **if** $\models pre \implies (g_1 \iff g_2)$ **then**
13  $\varphi_2' \leftarrow \varphi_2[g_2 : s_2 / g_2 : \hat{\sigma}(s_1)]$
14  $\Psi_i \leftarrow \Psi_i[\varphi_2 / \varphi_2']$
15  Suggest computation of $\hat{\sigma}(s_1)$ instead of $s_2$ under $g_2$
16  **else**
17  $h_2 \leftarrow g_2 \wedge \hat{\sigma}(g_1)$; $h_2' \leftarrow g_2 \wedge \hat{\sigma}(\neg g_1)$
18  $\varphi_2' \leftarrow \varphi_2[g_2 : s_2 / h_2 : \hat{\sigma}(s_1) \wedge h_2' : s_2]$
19  $\Psi_i \leftarrow \Psi_i[\varphi_2 / \varphi_2']$
20  Suggest computation of $\hat{\sigma}(s_1)$ instead of $s_2$ under $h_2$
21  **end**
22  **end**
23  **until** $k < \delta$
24  **if** $k = \delta$ **then** Suggest a correction to replace $\varphi_2$ by $\hat{\sigma}(\varphi_1)$
25 **end**

indices at the same nesting depths for the $i$th statements. The function $\hat{\sigma}$ is lifted in a straightforward manner to expressions and assignments. The algorithm checks whether the guards $g_1$ and $g_2$ are equivalent (line 12). If they are then the fault must be in the assignment statement $s_2$. It therefore defines $\varphi_2'$ by substituting $s_2$ by $\hat{\sigma}(s_1)$ in $\varphi_2$ (line 13) and refines $\Psi_i$ by replacing $\varphi_2$ by $\varphi_2'$ (line 14). It suggests an appropriate correction for the candidate submission (line 15). The other case when the guards are not equivalent leads to the other branch (lines 16-21). The algorithm now splits the guarded assignment $g_2 : s_2$ to make it conform to the reference under $h_2 \equiv g_2 \wedge \hat{\sigma}(g_1)$, whereas, for $h_2' \equiv g_2 \wedge \hat{\sigma}(\neg g_1)$, the candidate can continue to perform $s_2$ (line 17). It computes $\varphi_2'$ by replacing $g_2 : s_2$ by $h_2 : \hat{\sigma}(s_1)$ and $h_2' : s_2$ (line 18). It then refines $\Psi_i$ by replacing $\varphi_2$ by $\varphi_2'$ (line 19) and suggests an appropriate correction for the candidate submission (line 20). The refinement loop terminates when no more counter-examples can be found (line 9) and thus, progressively finds *all* semantic differences between $i$th statements of the two submissions.

Each iteration of the refinement loop eliminates a semantic difference between a pair of statements from the two submissions and the loop terminates after a finite number of iterations. In practice, giving a long list of corrections might not be useful to the student if there are too many mistakes in the submission. A better alternative might be to stop generating corrections after a threshold is reached. We use a constant $\delta$ to control how many refinements should be attempted (line 23). If this threshold is reached then the algorithm suggests a total substitution of $\hat{\sigma}(\varphi_1)$ in place of $\varphi_2$ (line 24). In our experiments, we used $\delta = 10$.

Due to the explicit verification of equivalence queries, our algorithm only generates correct feedback. The feedback for the declarations of the candidate are obtained by checking dimensions of the corresponding variables according to $\sigma$.

# 4. IMPLEMENTATION

**Table 1: Summary of submissions and clustering results.**

| Problem | Total subs. | Clusters with correct sub. | Clusters with manually added correct sub. |
|---|---|---|---|
| SUMTRIAN | 1983 | 78 | 2 |
| MGCRNK | 144 | 23 | 3 |
| MARCHA1 | 58 | 4 | 2 |
| PPTEST | 41 | 2 | 0 |
| Total | 2226 | 107 | 7 |

We consider C programs for experimental evaluation. We have implemented the source code analysis using the Clang front-end of the LLVM framework [26] and use Z3 [10] for SMT solving. We presently do not support pointer arithmetic.

In the pre-processing step, our tool performs some syntactic transformations. It rewrites compound assignments into regular assignments. For example, `x += y` is rewritten to `x = x + y`. A code snippet of the form: `scanf("%d", &a[0]); for (i = 1; i < n; i++) scanf("%d", &a[i]);`, where the input array is read in multiple statements is transformed to use a single read statement. The above snippet will be rewritten to `for (i = 0; i < n; i++) scanf("%d", &a[i]);`. Sometimes, students read a scalar variable and then assign it to an array element. Our tool eliminates the use of the scalar variable and rewrites the submission so that the input is read directly into the array element. Another common pattern is to read a sequence of input values into a scalar one-by-one and then use it in the DP computation. For example, consider the code snippet: `for (i = 0; i < n; i++) for (j = 0; j < n; j++) { scanf("%d", &x); dp[i][j] = dp[i-1][j] + x; }`. It does not use an array to store the sequence of input values. We declare an array and rewrite the snippet to use it. When feedback is generated for the submission, an explanatory note about the input array is added. In each of the syntactic transformations, we ensure that the program semantics is not altered.

Many students, especially beginners, write programs with convoluted conditional control flow, and unnecessarily complex expressions. In addition, the refinement steps of our counter-example guided feedback generation algorithm may generate complex guards. To present clear and concise feedback even in the face of these possibilities, in the post-processing step, our tool simplifies guards in the feedback using the SMT solver. We use Z3's tactics to remove redundant clauses, evaluate sub-expressions to Boolean constants and simplify systems of inequalities.

# 5. EXPERIMENTAL EVALUATION

To assess the effectiveness of our technique, we collected submissions to the following 4 DP problems[3] on CodeChef:

1. SUMTRIAN – Described in the Introduction section.
2. MGCRNK – Find a path from (1,1) to (N,N) in an N $\times$ N matrix, so that the average of all integers in cells on the path, excluding the end-points, is maximized. From each cell, the path can extend to cells to the right or below.
3. MARCHA1 – The subset sum problem.
4. PPTEST – The knapsack problem.

We selected submissions to these problems that implemented an iterative DP strategy in the C language. A user can submit solutions any number of times. We picked the latest submissions from individual users. These represent their best efforts and can benefit

[3] http://www.codechef.com/problems/<problem-name>

**Table 2: Results of feedback generation.**

| Problem | Verified as correct (✓) | Corrections suggested (✗) | Average corrections | Unlabeled (?) |
|---|---|---|---|---|
| SUMTRIAN | 1049 | 659 | 3.3 | 275 |
| MGCRNK | 61 | 66 | 6.8 | 17 |
| MARCHA1 | 9 | 35 | 10.3 | 14 |
| PPTEST | 3 | 29 | 12.7 | 9 |
| Total | 1122 | 789 | 4.3 | 315 |

from feedback. We do not consider submissions that either do not compile or crash on CodeChef's tests. To enable automated testing on CodeChef, the submissions had an outermost loop to iterate over test cases – we identified and removed this loop automatically before further analysis.

Table 1 shows the number of submissions for each problem. SUMTRIAN had the maximum number of submissions (1983) and PPTEST had the minimum (41). There were a total of 2226 submissions from 1860 students representing over 250 institutions. These submissions employ a wide range of coding idioms and many possible solution approaches, both correct and incorrect. This is a fairly large, diverse and challenging set of submissions.

## 5.1 Effectiveness of Clustering

Our features were quite effective in clustering submissions by their solution strategies. Since we do not include features representing low-level syntactic or structural aspects of submissions, the clustering resulted in only a few clusters for each problem, without compromising our ability to generate verified feedback. Table 1 gives the number of clusters. The number of clusters increased gracefully from the smallest problem (by the number of submissions) to the largest one. The smallest problem PPTEST yielded only 2 clusters for 41 submissions, whereas, the largest problem SUMTRIAN yielded 80 clusters for 1983 submissions. Our manual evaluation revealed that in each cluster, the solutions were actually following the same DP strategy.

The small number of clusters reduces the burden on the instructor significantly. Instead of evaluating 2226 submissions separately, the instructor is required to look at representatives from only 114 clusters. CodeChef uses test suites to classify problems into correct and incorrect. As a simple heuristic, we randomly picked one of the submissions marked as correct by CodeChef in each cluster and manually validated it. As shown in Table 1, this gave us correct representatives for 107/114 clusters across the problems. The remaining 7 clusters seemed to follow some esoteric strategies and we manually added a correct solution to each of them.

Clustering also helps the instructor get a bird's eye view of the multitude of solution strategies. For example, it can be used to find the most or least popular strategy used in student submissions. In SUMTRIAN, the most popular strategy (with 677 submissions) was the one that traverses the matrix rows bottom up, traverses the columns left to right and updates the element `(i,j)`.

## 5.2 Effectiveness of Feedback Generation

Our tool verifies a submission from a cluster against the manually validated or added correct submission from the same cluster. Table 2 shows the number of 1) submissions verified as correct (✓), 2) submissions for which faults were identified and corrections suggested (✗) and 3) submissions which our algorithm could not handle (?). Across the problems, 1122 submissions amounting to 50% were verified to be correct, with the maximum at 53% for SUMTRIAN and the minimum at 7% for PPTEST. For a total of 789 submissions amounting to 35%, some corrections were suggested by our tool. The maximum percentage of submissions with corrections were for PPTEST at 71% and the minimum was 33% for SUMTRIAN. Many submissions had multiple faults. Table 2 shows the average number of corrections over faulty submissions for each problem. PPTEST required the maximum number of corrections of 12.7 on average. In all, our tool succeeded in either verifying or generating verified feedback for 85% submissions.

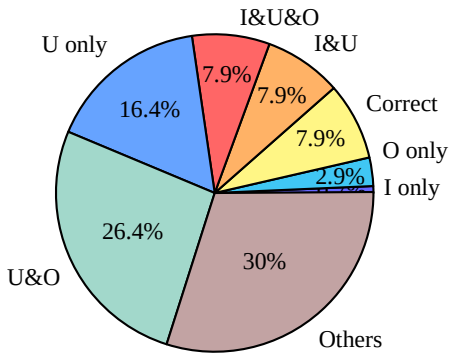



**Figure 6: Distribution of submissions in a cluster of SUMTRIAN by the type of feedback.**

For the remaining 315 (15%) submissions, our tool could neither generate feedback nor verify correctness. These submissions need manual evaluation. MARCHA1 had the maximum percentage of unlabeled submissions at 24% and MGCRNK had the minimum at 12%. These arise either because the SMT solver times out (we kept the timeout of 3s for each equivalence query), or due to the limitations of the verification algorithm or the implementation.

These results on the challenging set of DP submissions are encouraging and demonstrate effectiveness of our methodology and technique. Even if we assume that all 315 unhandled submissions are faulty, we could generate verified feedback for 71% faulty submissions. In comparison, on a set of *introductory* programming assignments, Singh et al. [46] report that 64% of faulty submissions could be fixed using *manually provided error models*. Our counter-example guided feedback generation technique guarantees correctness of the feedback. In addition, we would have liked to communicate the feedback to the students and assess their responses. Unfortunately, their contact details were not available to us.

### *Diversity of Feedback and Personalization.*

The feedback propagation approaches [20, 41] suggest that the same feedback text written by the instructor can be propagated to all submissions within a cluster. We found that this is not practical and the submissions within the same cluster require heterogeneous feedback. Figure 6 shows the distribution of submissions in a cluster of SUMTRIAN by the type of feedback. We only highlight feedback over the logical components of a submission: initialization (I), update (U) and output (O). Feedback related to type declarations and input statements (possibly, in conjunction with feedback on the logical components) is summarized under the category "Others".

While only 7.9% submissions were verified to be correct, 20% submissions had faults in one of the logical components of the DP strategy: initialization (0.7%), update (16.4%) and output (2.9%). As shown in Figure 6, a large percentage of submissions had faults in two logical components, and 7.9% had them in all three components. 30% of the submissions were in the others category. Clearly, it would be difficult for the instructor to predict faults in other submissions in a cluster by looking only at some submissions in the cluster and write feedback applicable to all. We do admit that Figure 6 is based on our clustering approach and other approaches may yield different clusters. Even then, the clusters would be correct

**Table 3: Submissions by faulty components.**

| Faulty comp. | SUMTRIAN | MGCRNK | MARCHA1 | PPTEST |
|---|---|---|---|---|
| I only | 36 | 15 | 0 | 0 |
| U only | 229 | 7 | 5 | 2 |
| O only | 31 | 1 | 6 | 0 |
| I&U | 29 | 18 | 2 | 8 |
| I&O | 10 | 0 | 1 | 0 |
| U&O | 97 | 1 | 2 | 0 |
| I&U&O | 30 | 0 | 11 | 0 |
| Others | 197 | 24 | 8 | 19 |
| Total | 659 | 66 | 35 | 29 |

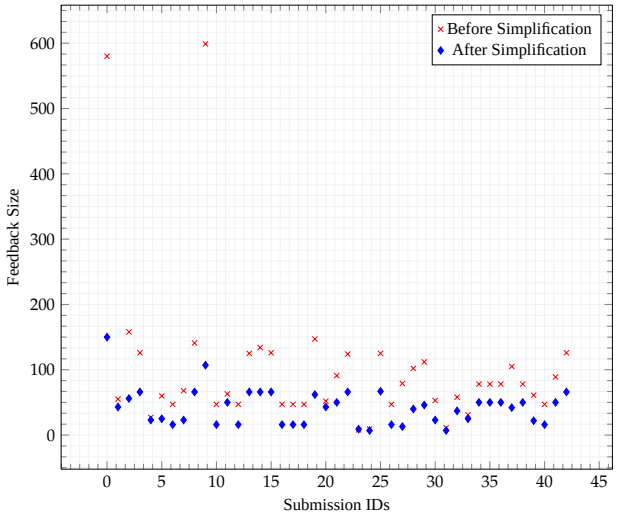



**Figure 7: Effect of simplification on feedback size for MGCRNK**

only in a *probabilistic* sense and the verification phase, we suggest, would add *certainty* about correctness of feedback.

Our technique generated personalized feedback depending on which components of a submission were faulty. Table 3 shows the number of submissions by the faulty components. Across the problems, PPTEST had the maximum percentage 53.7% of submissions requiring corrections to multiple logical components and SUMTRIAN had the minimum percentage 17.5%. The most common faulty components varied across problems.

### *Types of Faults Found and Corrected.*

Our tool found a wide range of faults and suggested appropriate corrections for them. This is made possible by availability of a correct submission to verify against and the ability of our verification algorithm to refine the equivalence queries to find all faults. The faults found and corrected include: incorrect loop headers, initialization mistakes including missing or spurious initialization, missing cases in the DP recurrence, errors in expressions and guards, incorrect dimensions, etc.

### *Conciseness of Feedback.*

To reduce the size of formulae in the generated feedback, we perform simplifications outlined in Section 4. We measure the effectiveness of the simplifications by disabling them and using the sum of AST sizes (#nodes in the AST) of the guards in our feedback text as *feedback size*. Figure 7 shows the impact of the simplifications on feedback size in the case of MGCRNK by plotting submission IDs versus feedback size. The figure excludes cases where simplification had no impact on feedback size. Simplifications ensured that the feedback size was at most 150, and 42.1 on average. Without simplifications, the maximum feedback size was 599. Simplifications, where applicable, reduced feedback size by 63.1% on an average across the problems.

## 5.3 Comparison with CodeChef

Our tool was able to verify 12 submissions as correct that were tagged by CodeChef as incorrect. This was surprising because CodeChef uses tests which should not produce such *false positives*. On investigation, we found that the program logic was indeed correct, as verified by our tool. The faults were localized to output formatting, or in custom input/output functions. Understandably, black-box testing used by CodeChef cannot distinguish between formatting and logical errors. However, being able to distinguish between these types of faults would save time for the students. Our tool finds logical faults but not formatting errors.

Due to the incompleteness of testing, CodeChef did not identify all faulty submissions (*false negatives*). This can hurt students since they may not realize their mistakes. We checked the cases when CodeChef tagged a submission as correct but our tool issued some corrections. For 64 submissions, our tool identified that the submissions were making spurious initializations to the DP array. For 112 submissions, our tool identified that the DP udpate was performed for additional iterations than required and generated feedback to fix the bounds of loops containing update statements. Importantly, our tool detected *out-of-bounds array accesses* in 99 submissions, and suggested appropriate corrections. In 265 distinct submissions, our tool was able to identify one or more of the faults described above, whereas CodeChef tagged them as correct! Thus, our static technique has a qualitative advantage over the test-based approach of online judges.

## 5.4 Performance

We ran our experiments on an Intel Xeon E5-1620 3.60 GHz machine with 8 cores and 24GB RAM. Out tool runs only on a single core. On an average, our tool generated feedback in 1.6s including the time for clustering and excluding the time for identifying correct submissions manually.

## 5.5 Limitations and Threats to Validity

Our technique fails for submissions that have loop-carried dependencies over scalar variables apart from the loop index variables, submissions that use auxiliary arrays and submissions for which pattern matching fails to label statements. We inherit the limitations of SMT solvers in reasoning about non-linear constraints and program expressions with undefined semantics, such as division by 0. Most of the unhandled cases arise from these limitations.

Our approach cannot suggest feedback for errors in custom input/output functions, output formatting, typecasting, etc. Our approach may provide spurious feedback enforcing stylistic conformance with the instructor-validated submission. For example, if a submission starts indexing into arrays from position 1 but the instructor-validated submission indexes from position 0, our tool generates feedback requiring the submission to follow 0 based indexing. This may correct some misconception about array indexing that the student may have. Nevertheless, these differences can be either compiled away during pre-processing or through SMT solving with additional annotations. We will investigate these in future. Finally, our implementation currently handles only a frequently used subset of C constructs and library functions.

There can be faults in our implementation that might have affected our results. To address this threat, we manually checked the feature values and feedback obtained, and did not encounter any error. Threats to external validity arise because our results may not generalize to other problems and submissions. We mitigated this

threat by drawing upon submissions from more than 1860 students on 4 different problems. While our technique is able to handle most constructs that introductory DP coursework employs, further studies are required to validate our findings in the case of other problems. In Section 5.3, we compared our tool with the classification available on CodeChef. The tests used by CodeChef are not public and hence, we cannot ascertain their quality. By improving the test suites, some false negatives of CodeChef may disappear but black-box testing will not be able to distinguish between logical faults and formatting errors (discussed in Section 5.3).

# 6. RELATED WORK

*Program Representations and Clustering.*

In order to cluster submissions effectively, we need strategies to represent both the syntax and semantics of programs. Many clustering approaches use only edit distance between submissions [15, 44], while others use edit distance along with test-based similarity [20, 36, 12]. We use neither of these. Glassman et al. [13] advocate a hierarchical technique where the submissions are first clustered using high-level (abstract) features and then using low-level (concrete) features. An interesting recent direction is to use deep learning to compute and use vector representations of programs [40, 41, 33]. Peng et al. [40] propose a pre-training technique to automatically compute vector representations of different AST nodes which is then fed to a tree-based convolution neural network [33] for a classification task. Piece et al. [41] propose a recursive neural network to capture both the structure and functionality of programs. The functionality is learned using input-output examples. But the class of programs considered in [41] is very simple. It only handles programs which do not have any variables.

Since our experiments were focused on iterative DP solutions, we designed features that capture the DP strategy. The above approaches are more general but unlike us, they may not put the submissions in Figure 2 and 3 in the same cluster. Our algorithm extracts features in the presence of temporary variables and procedures, and might be useful in other contexts as well.

*Feedback Generation and Propagation.*

The idea of comparing instructor provided solutions with student submissions appears in [2]. It uses graph representation and transformations for comparison of Fortran programs. Xu and Chee [49] use richer graph representations for object-oriented programs. Rivers and Koedinger [44] use edit distance as a metric to compare graphs and generate feedback. Gross et al. [15] cluster student solutions by structural similarity and perform syntactic comparisons with a known correct solution to provide feedback. Feedback generated by pattern matching may not always be correct. In contrast, we generate *verified* feedback but for the restricted domain of DP.

Alur et al. [4] develop a technique to automatically grade automata constructions using a pre-defined set of corrections. Singh et al. [46] apply sketching based synthesis to provide feedback for introductory programming assignments. In addition to a reference implementation, the tool takes as input an error model in the form of correction rules. Their error model is too restrictive to be adapted to our setting that requires more sophisticated repairs and that too for a more challenging class of programs. Gulwani et al. [17] address the orthogonal issue of providing feedback to address performance issues, while Srikant and Aggarwal [47] use machine learning to assess coding quality of prospective employees and do not provide feedback on incorrect solutions.

The idea of exploiting the common patterns in DP programs has been used by Pu et al. [43] but for synthesis of DP programs. The clustering-based approaches [20, 41] propagate the instructor-provided feedback to all submissions in the same cluster, whereas we generate personalized and verified feedback for each submission in a cluster separately. OverCode [12] also performs clustering of submissions and provides a visualization technique to assist the instructor in manually evaluating the submissions.

*Program Repair and Equivalence Checking.*

Genetic programming has been used to automatically generate program repairs [5, 11, 27]. These approaches are not directly applicable in our setting as the search space of mutants is very large. Further, GenProg [27] relies on redundancy present in other parts of the code for fixing faults. This condition is not met in our setting. Software transplantation [18, 6] transfers functionality from one program to another through genetic programming and slicing. Prophet [30] learns a probabilistic, application independent model of correct code from existing patches, and uses it to rank repair candidates from a search space. These are generate-and-validate approaches which rely on a test suite to validate the changes. In comparison, we derive corrections for a faulty submission by program equivalence checking with a correct submission.

Konighopher et. al. [25] present a repair technique using reference implementations. Their fault model is restrictive and only considers faulty RHS. Many approaches rely on program specifications for repair, including contracts [39, 48], LTL [23], assertions [45] and pre-post conditions [14, 28, 19]. Recent approaches that use tests to infer specifications and propose repairs include SemFix [37], MintHint [24], DirectFix [31] and Angelix [32]. These approaches use synthesis [22], symbolic execution [9] and partial MaxSAT [10] respectively. Both DirectFix and Angelix use partial MaxSAT but Angelix extracts more lightweight repair constraints to achieve scalability. SPR [29] uses parameterized transformation schemas to search over the space of program repairs. In contrast, we use instructor-validated submissions and a combination of pattern matching, static analysis and SMT solving.

Automated equivalence checking between a program and its optimized version has been studied in translation validation [42, 35, 7]. Partush and Yahav [38] design an abstract interpretation based technique to check equivalence of a program and its patched version. In comparison, our technique performs equivalence check between programs written by different individuals independently.

All these approaches are designed for developers and deal with only one program at a time. Our tool targets iterative DP solutions written by students and works on a large number of submissions simultaneously. It combines clustering and verification to handle both the scale and variations in student submissions.

# 7. CONCLUSIONS AND FUTURE WORK

We presented semi-supervised verified feedback generation to deal with both scale and variations in student submissions, while minimizing the instructor's efforts and ensuring feedback quality. We also designed a novel counter-example guided feedback generation algorithm. We successfully demonstrated the effectiveness of our technique on 2226 submissions to 4 DP problems.

Our results are encouraging and suggest that the combination of clustering and verification can pave way for practical feedback generation tools. There are many possible directions to improve clustering and verification by designing sophisticated algorithms. We plan to investigate these for more problem domains.

# 8. REFERENCES

[1] www.acm.org/public-policy/education-policy-committee.

[2] A. Adam and J.-P. Laurent. LAURA, a system to debug student programs. *Artificial Intelligence*, 15(1-2):75 – 122, 1980.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, 2nd edition, 2006.

[4] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of DFA constructions. In *IJCAI*, pages 1976–1982, 2013.

[5] A. Arcuri. On the Automation of Fixing Software Bugs. In *ICSE Companion*, pages 1003–1006, 2008.

[6] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *ISSTA*, pages 257–269, 2015.

[7] C. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck. TVOC: A translation validator for optimizing compilers. In *CAV*, pages 291–295. Springer-Verlag, 2005.

[8] R. E. Bellman. *Dynamic Programming*. Dover Publications, Incorporated, 2003.

[9] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, pages 209–224, 2008.

[10] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[11] V. Debroy and W. E. Wong. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *ICST*, pages 65–74, 2010.

[12] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Trans. Comput.-Hum. Interact.*, 22(2):7:1–7:35, Mar. 2015.

[13] E. L. Glassman, R. Singh, and R. C. Miller. Feature engineering for clustering student solutions. In *Proceedings of the First ACM Conference on Learning @ Scale Conference*, L@S '14, pages 171–172. ACM, 2014.

[14] D. Gopinath, Z. M. Malik, and S. Khurshid. Specification-based program repair using SAT. In *TACAS*, pages 173–188, 2011.

[15] S. Gross, X. Zhu, B. Hammer, and N. Pinkwart. Cluster based feedback provision strategies in intelligent tutoring systems. In *Intelligent Tutoring Systems*, pages 699–700. Springer, 2012.

[16] S. Gulwani and S. Juvekar. Bound analysis using backward symbolic execution. Technical report, October 2009.

[17] S. Gulwani, I. Radiček, and F. Zuleger. Feedback generation for performance problems in introductory programming assignments. In *FSE*, pages 41–51, 2014.

[18] M. Harman, W. B. Langdon, and W. Weimer. Genetic programming for reverse engineering. In *WCRE*, pages 1–10, 2013.

[19] H. He and N. Gupta. Automated debugging using path-based weakest preconditions. In *FASE*, pages 267–280, 2004.

[20] J. Huang, C. Piech, A. Nguyen, and L. J. Guibas. Syntactic and functional variability of a million code submissions in a machine learning MOOC. In *AIED*, 2013.

[21] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93, 2010.

[22] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided Component-based Program Synthesis. In *ICSE*, pages 215–224, 2010.

[23] B. Jobstmann, A. Griesmayer, and R. Bloem. Program Repair as a Game. In *CAV*, pages 287–294, 2005.

[24] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. MintHint: Automated synthesis of repair hints. In *ICSE*, pages 266–276, 2014.

[25] R. Konighofer and R. Bloem. Automated Error Localization and Correction for Imperative Programs. In *FMCAD*, pages 91–100, 2011.

[26] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, Palo Alto, California, Mar 2004.

[27] C. Le Goues, T. N., S. Forrest, and W. Weimer. Genprog: A Generic Method for Automatic Software Repair. *IEEE Trans. on Software Engineering*, pages 54 –72, 2012.

[28] F. Logozzo and T. Ball. Modular and Verified Automatic Program Repair. In *OOPSLA*, pages 133–146, 2012.

[29] F. Long and M. Rinard. Staged program repair with condition synthesis. In *ESEC/FSE*, 2015.

[30] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *POPL*, pages 298–312, 2016.

[31] S. Mechtaev, J. Yi, and A. Roychoudhury. DirectFix: Looking for simple program repairs. In *ICSE*, 2015.

[32] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *ICSE*, 2016.

[33] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, 2016.

[34] I. Narasamdya and A. Voronkov. Finding basic block and variable correspondence. In *Static Analysis*, pages 251–267. Springer, 2005.

[35] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94. ACM, 2000.

[36] A. Nguyen, C. Piech, J. Huang, and L. Guibas. Codewebs: Scalable homework search for massive open online programming courses. In *WWW*, pages 491–502, 2014.

[37] H. D. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program Repair via Semantic Analysis. In *ICSE*, 2013.

[38] N. Partush and E. Yahav. Abstract semantic differencing via speculative correlation. In *OOPSLA*, pages 811–828. ACM, 2014.

[39] Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer. Code-based Automated Program Fixing. In *ASE*, pages 392–395, 2011.

[40] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin. Building program vector representations for deep learning. In *Knowledge Science, Engineering and Management*, pages 547–553. Springer, 2015.

[41] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. J. Guibas. Learning program embeddings to propagate feedback on student code. In *ICML*, pages 1093–1102, 2015.

[42] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, pages 151–166. Springer-Verlag, 1998.

[43] Y. Pu, R. Bodik, and S. Srivastava. Synthesis of first-order dynamic programming algorithms. In *OOPSLA*, pages 83–98, 2011.

[44] K. Rivers and K. Koedinger. Automatic generation of

programming feedback: A data-driven approach. In *AIED*, pages 4:50–4:59, 2013.

[45] R. Samanta, O. Olivo, and E. Emerson. Cost-aware automatic program repair. In *Static Analysis*, Lecture Notes in Computer Science. 2014.

[46] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26, 2013.

[47] S. Srikant and V. Aggarwal. A system to grade computer programming skills using machine learning. In *KDD*, pages 1887–1896, 2014.

[48] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated Fixing of Programs with Contracts. In *ISSTA*, pages 61–72, 2010.

[49] S. Xu and Y. S. Chee. Transformation-based diagnosis of student programs for programming tutoring systems. *Software Engineering, IEEE Transactions on*, 29(4):360–384, April 2003.