DPAssist: Automated Feedback Generation for Iterative Dynamic Programming Assignments

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

Dynamic programming is a core algorithmic technique, taught commonly in algorithms courses. While a powerful tool when mastered, students —who are learning it for the first time— may struggle with the decomposition of the problem into overlapping subproblems and iterative reuse of the solutions to the subproblems.

In this paper, we present DPAssist, an automated technique to generate feedback on student submissions written using iterative dynamic programming strategy. DPAssist checks a student submission against reference implementations provided by the instructor. Checking program equivalence is difficult in this setting because of stylistic variations and use of procedures, loops and arrays. DPAssist therefore follows a novel staged approach. It performs a combination of static analysis and pattern matching on a program to compute a high-level, precise summary and then checks equivalence of summaries using an SMT solver. DPAssist identifies a reference implementation whose summary is closest to that of the student submission. If they are not equivalent, it reports a complete list of semantic differences between the two, using a novel counter-example guided iterative feedback generation algorithm.

We have evaluated DPAssist on 518 programs submitted to two problems posed on a contest site. DPAssist generated correct feedback on 79.5% submissions in average 5.3s each. It required only one reference implementation for every 11 submissions. This shows that the staged approach of DPAssist successfully handles a multitude of stylistic variations in submissions without requiring proportionately many reference implementations.

1. INTRODUCTION

The increasing realization about the importance of CS education has resulted in several websites that offer programming lessons, several MOOCs on various CS subjects and even adoption of computing curricula as part of school education. Unfortunately, providing good feedback on programs submitted by students is extremely challenging because of the differences in high-level strategies or low-level syntactic and stylistic variations across submissions [17]. This has motivated recent work on automating feedback generation, which until now, has focused on *introductory programming* assignments [40, 14, 41, 18]. In this paper, we investigate automatic feedback generation for the more sophisticated subject matter of *algorithms*, and in particular, dynamic programming.

Dynamic Programming. Among a variety of algorithmic techniques taught in algorithms courses, Dynamic Programming (DP) is an essential one as it forms the basis of many powerful algorithms on strings (e.g., longest common subsequence) and graphs (e.g., shortest path). These algorithms find applications in many areas such as text processing and bioinformatics.

```
void main() {
  int i, j, n, max;
  scanf("%d", &n); // Input
  int m[n][n], dp[n][n];
  for (i = 0; i < n; i++)
    for (j = 0; j <= i; j++)
      scanf("%d", &m[i][j]); // Input
  dp[0][0] = m[0][0]; // Init
  for (i = 1; i < n; i++) {
    for (j = 0; j <= i; j++) {
      if (j == 0)
        dp[i][j] = dp[i-1][j] + m[i][j]; // Update
      else if (j == i)
        dp[i][j] = dp[i-1][j-1] + m[i][j]; // Update
      else if (dp[i-1][j] > dp[i-1][j-1])
        dp[i][j] = dp[i-1][j] + m[i][j]; // Update
      else dp[i][j] = dp[i-1][j-1] + m[i][j]; // Update
    }
  }
  max = dp[n-1][0];
  for (i = 1; i < n; i++)
    if (dp[n-1][i] > max) max = dp[n-1][i];
  printf("%d", max); // Output
}
```

Figure 1: A reference implementation for matrix path problem

The essence of DP lies in its ability to efficiently exploit the structure of the problem, namely, overlapping subproblems and optimal substructure [6]. DP is considered more advanced than other strategies like divide-and-conquer and greedy algorithms, which do not entail overlapping subproblems. As an example, consider the *matrix path* problem[1] taken from a popular programming contest site CodeChef. We are given a lower triangular matrix m of n rows. Starting at a cell, we can traverse a path 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. To solve this problem, we identify (1) the optimal substructure: the solution for m[i][j] (i.e., the maximum weight of paths reaching m[i][j]) can be obtained by selecting the maximum from the solutions for m[i-1][j] and m[i-1][j-1] and adding m[i][j], and (2) overlapping subproblems: the solution for m[i][j] is used for computing solutions for m[i+1][j] and m[i+1][j+1]. Such observations are formalized as DP recurrences [6]. A DP algorithm computes an optimal solution to a problem by solving the DP recurrence efficiently.

Figure 1 shows a C program that implements a DP algorithm for the matrix path problem. The array dp stores the solutions to the subproblems. The program implements an iterative strategy for computing solutions starting from the top-left matrix cell. It iterates

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

```
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];
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] ; // Init
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 2: A faulty student submission for matrix path problem

over the rows from top-to-bottom and columns from left-to-right as seen in Figure 1 and uses previously computed values stored in dp. The DP recurrence can also be computed recursively. The recursive strategy is not preferred in practice because it does not scale well for large problem instances. In this work, we focus on the more common strategy of iteratively solving DP problems.

Iqbal and Alvi [19] argue that students have a tough time grasping the fundamentals of DP and need lot of practice and feedback. Our goal in this paper is to develop program analysis techniques to automatically generate feedback on iterative DP programs.

Our Approach. Most standard algorithmic techniques such as DP or divide-and-conquer prescribe certain templates for logical steps that should be instantiated to derive an implementation for the specific problem at hand. Our *key insight* is that these templates can be formally represented and their instantiations can be extracted statically from a program. We call it a *summary* of the program. The summaries do away with most of the syntactic and stylistic variations across different programs implementing the same algorithm and help in checking equivalence among them.

In this paper, we exploit this insight to automatically generate feedback for iterative DP assignments. We present a technique called *DPAssist*. Apart from student submissions, DPAssist takes *reference implementations* from the instructor. It then computes a precise summary of each program and checks equivalence of the summary of a student submission (i.e., *submission summary*) with the summary of each of the reference implementations (i.e., *reference summaries*). If the submission summary is not found to be equivalent to any reference summary, DPAssist generates feedback highlighting the differences between the submission summary and the *closest* reference summary. Thus, DPAssist *validates correct logical steps* in the student submission and generates *customized* feedback for the faulty steps, using a closest reference implementation. If the solution strategy used by the student is not represented in the reference implementations, the instructor may add another reference implementation and run DPAssist again.

To illustrate our approach, we give a faulty program, inspired by student submissions to the matrix path problem, in Figure 2. Let the program in Figure 1 be a reference implementation. DPAssist generates the following feedback —which is close to what could be generated manually— by comparing the two:

In the declaration step: (1) Types of A and D should be int[n][n]
In the update step:
(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 step: (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].

The first correction suggests that the submission use array sizes as int[n][n] instead of hardcoded value of int[101][101]. The DP update at line 20 misses some corner cases for which DPAssist generates corrections #2 and #3 above. The computation of output at line 22 should use the correct array bounds as indicated by correction #4.

These example programs use two-dimensional arrays, nested loops and the faulty submission even uses multiple procedures. Since they use different variable names and programming styles, it is nontrivial to use one to generate feedback for the other. The staged approach of DPAssist aids in overcoming these challenges. DPAssist first infers program summaries through static analysis based on backward substitution [13] and syntactic pattern matching specialized to iterative DP programs. The equivalence of two summaries is then checked using syntactic simplifications and satisfiability-modulo-theories (SMT) based constraint solving. We have developed a novel *counter-example guided feedback generation* technique to iteratively identify and fix *all* semantic differences between submission and reference summaries. The student submission in Figure 2 contains multiple faults. The feedback above identifies fixes for all the faults in it. In addition, we use SMT based simplification tactics to make feedback as *concise* as possible.

The idea of exploiting the common patterns in DP programs has been used by Pu et al. [36] but for synthesis of DP programs. Singh et al. [40] advocated that a complete list of semantic changes is a useful form of feedback to students. While they presented a repair technique for introductory programming assignments, we focus on the more sophisticated, but specialized, domain of DP. Gulwani et al. [14] used reference implementations to generate feedback for performance issues in introductory programming assignments. In contrast, our goal is to look for logical faults in DP programs. The program repair approaches (e.g., [25, 31, 22, 28, 27]) target developers and synthesize repairs for faults localized at the granularity of line numbers. In our opinion, the high-level feedback generated by DPAssist is more suitable to students. Unlike the program repair setting, we have the advantage of calling upon the instructor to provide reference implementations. This aids in giving complete feedback but then our technique must solve the challenging (and in general, undecidable) problem of checking program equivalence.

Implementation and Results. We have implemented DPAssist for C programs and evaluated it on 518 student submissions for two problems from CodeChef. Of these, 85 were tagged by the test-based online judge of CodeChef as wrong answers. On 79.5% of the total submissions, DPAssist generated correct feedback. In addition to faults in wrong answers, DPAssist also found many cases of faulty logic in the submissions accepted by CodeChef as *correct answers*. The static technique of DPAssist thus has a qualitative advantage over the test-based approach of online judges. The submissions come from 463 students from over 100 different institutes and are therefore representative of diverse backgrounds and coding styles. Even then, DPAssist required only one reference implementation for every 11 submissions. This shows that the staged approach of DPAssist successfully handles a multitude of stylistic

```
Declaration block
n ↦ int, m ↦ int[n][n], dp ↦ int[n][n], out ↦ int
Input block #1
true: n = any;
Input block #2
loop(i,0,n-1,+) {
  loop(j,0,i,+) {
    true: m[i][j] = any; } }
Init block
true: dp[0][0] = m[0][0]
Update block
loop(i,1,n-1,+) {
  loop(j,0,i,+) {
    (j == 0): dp[i][j] = dp[i-1][j] + m[i][j];
    (j != 0 && j == i): dp[i][j] = dp[i-1][j-1] + m[i][j];
    (j != 0 && j != i && dp[i-1][j] > dp[i-1][j-1]):
      dp[i][j] = dp[i-1][j] + m[i][j];
    (j != 0 && j != i && dp[i-1][j] ≤ dp[i-1][j-1]):
      dp[i][j] = dp[i-1][j-1] + m[i][j]; } }
Output block
true: out = _max(dp[n-1][0], dp[n-1][1..n-1]);
```

(a) Summary of reference implementation of Figure 1

```
Declaration block
n ↦ int, A ↦ int[101][101], D ↦ int[101][101], out ↦ int
Input block #1
true: n = any;
Input block #2
loop(i,0,n-1,+) {
  loop(j,0,i,+) {
    true: A[i][j] = any; } }
Init block
true: D[0][0] = A[0][0];
Update block
loop(i,1,n-1,+) {
  loop(j,0,i,+) {
    true: D[i][j] = A[i][j] + (D[i-1][j] > D[i-1][j-1] ?
      D[i-1][j] : D[i-1][j-1]);} }
Output block
true: out = _max(D[n-1][0], D[n-1][0..99]);
```

(b) Summary of faulty student submission of Figure 2

Figure 3: Program summaries computed by DPAssist

variations in submissions without requiring proportionately many reference implementations. DPAssist is fast even while checking a student submission against multiple reference implementations. It took on an average 5.3s to generate feedback for each submission.

Contributions. The contributions of this paper are follows:

- We motivate the problem of automatically generating feedback for programming assignments related to advanced algorithmic concepts, in particular, dynamic programming.
- We present DPAssist which contributes two novel techniques: (1) summary inference through a combination of pattern matching and static analysis which deals with stylistic variations effectively and (2) counter-example guided iterative feedback generation that generates a complete list of fixes for faults in a student submission. We believe that these techniques can be extended to other algorithmic classes as well.
- We report experimental results on 518 submissions to two algorithmic problems and show effectiveness of DPAssist.

2. OVERVIEW

We now return to the example presented in the Introduction section and demonstrate important steps of DPAssist. Figure 3 gives the summaries computed by DPAssist for the reference implementation (Figure 1) and the student submission (Figure 2). Each summary consists of different blocks corresponding to different logical steps: *declaration, input, init, update* and *output* where init, update and output refer to the initialization of the DP arrays, their updation and the computation of the output from the DP arrays.

Summary Inference. We explain summary inference for the student submission. DPAssist identifies the input variables and output expressions by simple pattern matching which extracts arguments to the I/O library functions such as `scanf` and `printf`. The statements that match I/O patterns are labeled as *input/output statements* in Figure 2. This gives us the variable `n` and the array `A[101][101]` as the input variables. We use a fresh integer valued variable `out` to store the output produced by the program. These variables and their types are recorded in the declaration block in Figure 3(b).

We label an assignment statement as an *update statement* if the LHS is an array element and the RHS uses some element of the same array or a variable that is defined in terms of some element of that array previously. The array being assigned is a DP array. Identifying update statements can be tricky and requires global program analysis. In contrast, simple pattern matching suffices for labeling input/output statements. We shall label line 20 in Figure 2 as an update statement and identify the array variable `D` as the DP array provided we can infer that the RHS at line 20 uses some element of `D`. The RHS involves a call to procedure `max`. Through a static analysis, it infers that the RHS expression is equivalent to `A[i][j] + (D[i-1][j] > D[i-1][j-1] ? D[i-1][j] : D[i-1][j-1])`. Through pattern matching over this expression, it detects that `D` is used in both branches and labels line 20 in Figure 2 as an update statement. The declaration of the DP array `D[101][101]` is recorded in the declaration block. An assignment to `D` where the RHS is an expression over constants or input variables is labeled as an *init statement*. Line 17 in Figure 2 is an init statement. Following the above technique, we have also labeled the key statements of the reference implementation in Figure 1.

Having labeled the key statements, DPAssist proceeds to infer the summary blocks. DPAssist performs syntactic pattern matching to summarize the code which encloses each of the labeled statements. We denote an arbitrary value read as input by `any`. The input blocks in Figure 3(b) record the way the input variables `n` and `A` are initialized. The elements of `A` are initialized in a nested loop (lines 14–16) where the loop indices go over each of the array dimensions. The `loop` construct gives the loop index, the two bounds on it and the direction (+/-) of index updation. A "+" indicates that the index is incremented in each iteration by one. The assignment statements in the summary are guarded by a boolean constraint. The init block states that `D[0][0]` is initialized to `A[0][0]`. The update block is obtained by using the expression derived while analyzing the procedure call at line 20. In our experiments, we observed that there are certain repeating iterative patterns used in the submissions. The computation of a maximum over an array in lines 6–8 is one such example. We encode patterns to lift these to certain predefined functions which make it is easier to reason about the summaries later. The function `_max` takes the first and the last elements of a contiguous array segment and returns the maximum over them. We use it in the output block in Figure 3(b) to summarize the computation in lines 6–8.

Summary inference cleanly separates logical steps in separate summary blocks and does away with several syntactic and stylistic quirks of a program. First, we permit only the input/output variables, DP arrays and loop indices in the summary. We call them the *summary variables*. We eliminate the use of variables other than the summary variables from the summary. Second, we eliminate procedure calls (e.g., line 20) while summarizing their semantics precisely. Third, DPAssist maps different loop constructs such as `for`, `while` and `do-while` to the canonical `loop` construct. Fourth, the guarded statements within a summary block are interpreted as a *set* of statements. In other words, the ordering between them is compiled away. Thus, two code fragments which swap the `true` and `false` branches but perform the same computation in two programs would be represented by syntactically equivalent set of guarded statements. As a result, the summaries in Figure 3 are much easier to compare than the corresponding programs.

Feedback Generation. DPAssist first checks whether the two summaries in Figure 3 contain the same number of blocks of the same type and in the same order. It uses the type information in declaration blocks to infer type-compatible variable correspondences. If multiple correspondences are possible, it performs the subsequent analysis for each of them separately. For purposes of illustration, consider that the variables `n` from both summaries correspond to each other. Similarly, `A` corresponds to `m`, `D` corresponds to `dp` and `out` corresponds to `out`. But for this correspondence to hold, we require the array sizes to be same. This gives rise to correction #1 in the feedback presented in the Introduction section.

DPAssist then checks equivalence of each block from the submission summary with the corresponding block from the reference summary. By unifying the summary variables, we get syntactic equality of the input and init blocks with their counter-parts. In the case of output blocks, we identify correction #4.

The case of update blocks is more interesting. The iteration spaces of these blocks are equal. To check equivalence of loop bodies, DPAssist constructs a formula φ which asserts that in each iteration, if the two DP arrays are equal at the beginning then they are equal at the end of the iteration. We defer the more technical details to Section 3.2. It then checks validity of φ. In this example, the formula is not valid and the SMT solver returns a counter-example which assigns values to all the variables in the formula. When substituted in the loop bodies, the first guard in the reference summary is satisfied, and the ternary condition in the submission summary evaluates to false. The corresponding assignment statements use different DP array elements on the RHS. DPAssist therefore generates correction #2. DPAssist thus correctly discovers that the submission does not handle a corner case: When `j==0`, we are at the first column in the lower-triangular matrix (recall the problem statement from the Introduction) and we should use solution to the subproblem stored at `D[i-1][j]` whereas the submission incorrectly accesses `D[i-1][j-1]`. The student should add code to handle this case as suggested by DPAssist and guard line 20 with `j!=0`.

To check whether the two loop bodies become equivalent when the correction is applied, DPAssist updates the loop body in the submission summary as per the suggested fix and checks equivalence of loop bodies again. It then discovers a counter-example where in the reference, the second guard evaluates to true, and in the (updated) submission, the guard `j!=0` and the ternary condition evaluate to true. These result in different assignments to `D[i][j]` – the reference assigns `D[i-1][j-1]` whereas the submission assigns `D[i-1][j]`. So DPAssist generates correction #3. This is another corner case –of updating cells along the diagonal– missed by the faulty submission. When `(j!=0 && j!=i)`, the student should retain the assignment at line 20. On incorporating this feedback, DPAssist is able to verify that the student summary and the reference summary are equivalent. Thus, DPAssist does not stop after discovering some one fault in the student submission. It discovers faults in all summary blocks independently. As was seen for update blocks, within the same block, it tries to discover all possible faults by iteratively looking for counter-examples and applying fixes.

3. TECHNICAL DETAILS

3.1 Summary Inference

We first introduce the *language of summaries*. The summary of a DP program is a formal representation of the logical steps implemented in it. A summary is made up of a sequence of blocks. In Section 2, we discussed different types of summary blocks and the notion of summary variables. The declaration block gives C types of summary variables. For brevity, we have omitted type declarations of loop indices in Figure 3.

A summary block (other than a declaration block) is either a set of guarded assignments or a possibly nested loop. A guarded assignment is of the form `g: x = e` where `g` is a boolean expression called a guard, and `x` and `e` are type-compatible variable and expression respectively. We skip discussion on syntax of guards and expressions as these are standard. We do not permit pointer arithmetic in expressions. A loop is of the form `loop(a,b,c,d) { e }` where `a` is a loop index variable, `b` and `c` are the bounds on it, `d` is either "+" or "-" and `e` is another statement. The meaning of "+" was introduced earlier; "-" means that the loop index is decremented by one in each iteration. We permit tightly nested loops, that is, if a loop statement s_1 is nested inside a loop statement s_2 then there cannot be another statement between them.

3.1.1 Building Blocks

Backward Substitution. It is natural for programmers to break a computation into smaller parts and use temporary variables to store intermediate results. From the perspective of summary inference, this complicates the structure of the program and obscures the underlying logical steps of the DP strategy. To eliminate the use of a temporary variable `x` at a program 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 the *summary variables* of the program. We denote this set by $\Sigma(l, \mathtt{x})$ and call Σ 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. For any run of the program, exactly one of the guards in $\Sigma(l, \mathtt{x})$ is satisfied at l. The substitution store Σ is lifted in a natural manner to expressions and statements. For instance, for an assignment statement $s \equiv \mathtt{x} = \mathtt{e}$, $\Sigma(l, s) = \{\mathtt{g_1} : \mathtt{x} = \mathtt{e_1}, \ldots, \mathtt{g_n} : \mathtt{x} = \mathtt{e_n}\}$ where $\{\mathtt{g_1} : \mathtt{e_1}, \ldots, \mathtt{g_n} : \mathtt{e_n}\} = \Sigma(l, \mathtt{e})$.

Gulwani and Juvekar [13] 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 call this *backward substitution*. We refer the reader to [13] for the details of the algorithm.

Syntactic Pattern Matching. We employ pattern matching for two main purposes: (1) for assigning labels (e.g., input, update, etc.) to program statements and (2) for extracting summary blocks.

We define a function updateLabelMap to assign a label to a statement s. Let l be the program location of s. updateLabelMap performs pattern matching over $\Sigma(l, s)$. In $\Sigma(l, s)$, the non-summary variables (or temporaries) in s are replaced by the guarded expressions from the substitution store. This makes the labeling part of DPAssist 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. In general, $\Sigma(l, s)$ may contain more than one statements (see the definition above). If $\Sigma(l, s) = \{s_1, \ldots, s_n\}$, updateLabelMap requires that all of $s_1, \ldots, s_n$ satisfy the same pattern and get the same label. We have already discussed the specific patterns that we match for different statement labels in Section 2.

Let genBlock be the function which infers summary blocks from

$$\begin{aligned}
\mathsf{uniqueDef}(s,B) &\equiv \forall s' \in B, \forall i,i' \in \mathsf{iter}(B) : h = \mathsf{defIn}(s,i) \wedge h' = \mathsf{defIn}(s',i') \implies (h \neq h') \vee (s = s' \wedge i = i') \\
\mathsf{allUseAfterDef}(s,B) &\equiv \forall s' \in B, \forall i,i' \in \mathsf{iter}(B) : h = \mathsf{defIn}(s,i) \wedge h \in \mathsf{useIn}(s',i') \implies (i < i') \vee (i = i' \wedge s <_{po} s') \\
\mathsf{useAfterAllDef}(s,B) &\equiv \forall s' \in B, \forall i,i' \in \mathsf{iter}(B), \forall h \in \mathsf{useIn}(s,i) : h = \mathsf{defIn}(s',i') \implies (i' < i) \vee (i' = i \wedge s' <_{po} s) \\
\mathsf{EqIter}(B_S,B_R,IC) &\equiv IC \wedge \mathsf{idxcorr}(B_S,B_R) \implies (\mathsf{iter}(B_R) \wedge \mathsf{guards}(B_R) \iff \mathsf{iter}(B_S) \wedge \mathsf{guards}(B_S)) \\
\mathsf{EqBody}(B_S,B_R,IC) &\equiv IC \wedge \mathsf{varcorr}(B_S,B_R) \implies (\mathsf{iter}(B_R) \wedge \mathsf{body}(B_R) \wedge \mathsf{body}(B_S) \implies \bigwedge_{\mathtt{v} \in \mathsf{updated}(B_R)} \mathtt{v} = \mathtt{v}')
\end{aligned}$$

Figure 4: Symbolic constraints

top-level statements in the program. A statement which is not enclosed within another statement is called a top-level statement. For example, the outer `for` loop starting at line 18 in Figure 2 is a top-level statement but the inner `for` loop starting at line 19 is not. Thus, DPAssist will generate a summary block for the outer loop (lines 18–20). A top-level statement can be an assignment statement, a conditional statement or a loop. The case of loops is most interesting. To summarize the loop header, our algorithm identifies the loop index variable `i`, expressions representing its initial value `iv` and final value `fv` and the increment to the loop index `d`. Let `e` be + is `d` is +1 and - if it is -1. DPAssist then generates `loop(i, iv, fv, e)` as the corresponding summary statement. We do the same for `while` and `do-while` loops using dataflow analysis [2].

Next, all labeled statements within the top-level statement are converted to their equivalent guarded statements in the summary using the substitution store such that the guards are pairwise disjoint. This removes `if-else` branching and sequencing between the program statement. If the top-level statement does not contain any labeled statement, no summary block is generated for it. If the resulting summary block does not meet the syntactic restrictions of our summary language, DPAssist returns $FAIL$.

Symbolic Constraints. Let B be a summary block and s a statement in it. Let $\mathsf{iter}(B)$ represent the iteration space defined by the loops in B. For example, `loop(i,0,n,+)` defines the space $0 \leq \mathtt{i} \leq \mathtt{n}$. $\mathsf{defIn}(s,i)$ returns the memory location that statement s defines in iteration $i \in \mathsf{iter}(B)$. For example, if s is the statement `a[i][j] = e`, and `i` and `j` are loop index variables then $\mathsf{defIn}(s,(0,0)) = \mathtt{a}[0][0]$. Similarly, $\mathsf{useIn}(s,i)$ returns the set of all memory locations that are used in s at iteration $i \in \mathsf{iter}(B)$. $<_{po}$ defines the *program order* among two statements in B.

The formula $\mathsf{uniqueDef}(s,B)$ asserts that no other statement in B writes to the memory locations defined by s and that s also does not write to the same location multiple times. Next, the formula $\mathsf{allUseAfterDef}(s,B)$ asserts that all uses of memory locations defined in s occur after s. We define $\mathsf{useAfterAllDef}(s,B)$ to assert that the definitions for all memory locations used in s occur before s. In other words, the location used in s is not redefined later. The other formulae in Figure 4 are explained in Section 3.2.

Pre-processing. In a pre-processing phase, we replace repeating iterative computations by the predefined functions (e.g., `_max`) directly in the source code of the program. We require the expressions in the program to be free of side-effects. This requirement is satisfied by applying a standard program transformation which introduces temporaries. Since we focus only on iterative DP programs, we do not permit recursive procedures. For presentation, we assume that all procedures are inlined. However, our implementation does work directly on procedural programs. We present the algorithm assuming there is only one DP array in the program. The extension to multiple DP arrays is straightforward.

3.1.2 Algorithm

Algorithm 1 presents our algorithm COMPUTESUMMARY to compute the summary of an iterative DP program P. Line 2 initializes the substitution store Σ, the labeling map L for statements and the summary S. COMPUTESUMMARY calls identifySummVars (line 3) which returns the set of summary variables SV in the program by pattern matching. Initially, all array variables are treated as potential DP arrays. The algorithm then goes over each statement s in the program. It calls updateSubStore (line 5) which returns an updated substitution store after analyzing s. Next, the algorithm calls updateLabelMap (see Section 3.1.1) at line 6 to obtain the updated labeling map after analyzing s.

Once the labeling is computed, the algorithm proceeds to obtain summary blocks from the program. It goes over each top-level statement t in the program (line 8) in the program order and calls genBlock (line 9) to transform t into a single summary block through syntactic pattern matching (see Section 3.1.1). Note that the ordering of the summary blocks in S is same as the ordering between the respective top-level statements in the program.

The summary block obtained at line 9 may contain statements with different labels. This happens, for example, if the program performs initialization and updation of the DP array within the same loop. For such blocks, isHeterogenous returns true (line 10). The algorithm then proceeds to obtain a sequence of homogeneous blocks for the heterogeneous block (line 11). A block is homogeneous if all the labeled statements within it have the same label.

The procedure getHomogeneousBlocks goes over each input and init statement s in B (line 19). It checks whether $\mathsf{uniqueDef}(s,B)$ (defined in Figure 4) is valid to assert that there are no other definitions of the locations defined at s (line 20). Next, it checks the validity of $\mathsf{allUseAfterDef}(s,B)$ (line 21) to assert that any use of the locations defined in s happens after s. If these checks succeed, we can safely move the input/init statement outside B and place it *before* the rest of B in the final sequence of homogeneous blocks. The sequence of blocks thus obtained is semantically equivalent to the original heterogeneous block since the transformation preserves all *dependences* [10, 16, 37]. Next, the procedure goes over each output statement s in B (line 23). It checks the validity of $\mathsf{useAfterAllDef}(s,B)$ to assert that all variables used in the output statement s are defined before the output statement. In this case, we can safely move the output statement outside B and place it *after* the rest of B. Again, this transformation preserves the dependences in the block. When all the validity checks are successful, the algorithm calls splitBlocks (line 26) that applies the transformations described above and returns a sequence of homogeneous blocks. The function splitBlocks duplicates the loop header of B to create a new block B' and moves the appropriate statements to B'. The input and init statements are placed in distinct blocks and the block with input statements comes before the block with init statements.

We note that getHomogeneousBlocks is a powerful procedure that leverages dependence analysis to separate out different logical steps interleaved within the same block. For example, if a student submission performs init and update in the same loop but a reference implementation performs them in separate loops, COMPUTESUMMARY computes comparable summaries for the two.

Algorithm 1: Algorithm ComputeSummary

```
Input: A program $P$ implementing an iterative DP strategy
Output: The summary $S$ of $P$ and the set of residual statements of $P$
1  begin
2      $\Sigma \leftarrow \emptyset, L \leftarrow \emptyset, S \leftarrow \emptyset$
3      $SV \leftarrow \mathsf{identifySummVars}(P)$
4      foreach statement $s \in P$ do
5          $\Sigma \leftarrow \mathsf{updateSubStore}(\Sigma, s, SV, P)$
6          $L \leftarrow \mathsf{updateLabelMap}(L, \Sigma, s, SV)$
7      end
8      foreach top-level statement $t \in P$ in the program order do
9          $B_t \leftarrow \mathsf{genBlock}(t, \Sigma, L, P)$
10         if $\mathsf{isHeterogeneous}(B_t)$ then
11             $B_h \leftarrow \mathsf{getHomogeneousBlocks}(B_t)$
12             $S \leftarrow \mathsf{append}(B_h, S)$
13         else  $S \leftarrow \mathsf{append}(B_l, S)$ end
14     end
15     if $\mathsf{BlocksInOrder}(S)$ then  return $\langle S, \mathsf{getResidual}(P, S, \Sigma)\rangle$
16 end
17 Procedure $\mathsf{getHomogeneousBlocks}(B : block)$
18 begin
19     foreach statement $s \in \mathsf{Input}(B) \cup \mathsf{Init}(B)$ do
20         if $\mathsf{uniqueDef}(s, B)$ is not valid then  return FAIL
21         if $\mathsf{allUseAfterDef}(s, B)$ is not valid then  return FAIL
22     end
23     foreach statement $s \in \mathsf{Output}(B)$ do
24         if $\mathsf{useAfterAllDef}(s, B)$ is not valid then  return FAIL
25     end
26     return $\mathsf{splitBlocks}(B)$
27 end
```

Finally, the algorithm checks if the blocks are in the required order. The function BlocksInOrder takes a sequence of blocks, obtains the labels for them and returns true if the labels match the regular expression "Declaration Input* Init* Update* Output*". If not, the algorithm fails. Otherwise, it returns the blocks obtained as the summary and calls getResidual to obtain the set of residual statements. The function getResidual returns the set of program statements that are not summarized in S. These are the statements in the program that did not match any of the statement patterns and also were not analyzed as part of the backward substitution procedure. These may arise if the submission performs extraneous computation not required in the DP solution. The residual statements are shown to the student as part of feedback. Recall that identifySummVars returns all arrays as potential DP arrays (line 3). The arrays for which ComputeSummary does not discover an update statement are treated as auxiliary variables. The statements operating over them are added to the residual statements. If any block in S refers to them, the algorithm returns $FAIL$.

Our algorithm satisfies several properties. The backward substitution procedure computes the substitution store for the given program. The labeling procedure computes precise statement labels. Finally, the dependence analysis ensures that the splitting of heterogeneous blocks into homogeneous blocks is sound. These ensure that whenever ComputeSummary succeeds (i.e., does not return $FAIL$), it computes a sound and precise summary of the given program modulo the residual statements.

3.2 Feedback Generation

Our objective is to generate feedback for a student submission by identifying the semantic differences between it and the closest reference implementation. Program equivalence is an undecidable problem and in practice, the key difficulties are in identifying variable and control correspondences between programs [29]. DPAssist therefore uses program summaries. Their simple format makes it easy to identify variable and control correspondences.

To identify *variable correspondence*, first all type-compatible maps are generated from the declaration blocks of two summaries. If we cannot obtain any bijective map, we discard the candidate reference summary. DPAssist retains only those maps that are consistent with the order in which the input blocks in the two summaries initialize the input variables. If there are multiple variable correspondences that remain, DPAssist generates feedback for the summaries wrt each variable correspondence map ρ separately. If two variables are mapped to each other but their types are not equal then DPAssist generates a feedback which suggests that the type of the variable from the student summary be changed to that of the corresponding variable from the reference summary. The *control correspondence* is based on the order and types of summary blocks. Because our summary inference algorithm already separates the logical steps into distinct summary blocks, we check equivalence of a pair of related blocks from the two summaries. This reduces the problem of checking program equivalence to the more amenable problem of checking equivalence of summary blocks.

3.2.1 Building Blocks

Identifying Closest Reference Summary. Different students may follow different high-level solution strategies for the same problem. We rely on the instructor to provide reference implementations that represent different high-level strategies.

DPAssist performs a lightweight structural comparison of a submission summary with reference summaries. It checks whether they contain the same number of blocks of the same kind and in the same order. Within each block, it checks whether a `loop` construct is used and if yes, do they have the same nesting depth and whether the directions (+/-) in the corresponding loops are the same. The submission summary is checked for equivalence with every reference summary that meets these requirements. We design a simple function score which assigns a score that is same as the number of corrections suggested in the feedback. A reference summary with the smallest feedback score is thereby *closest* to the submission summary and the corresponding feedback is shown to the student.

Symbolic Constraints. Let B_S and B_R be two summary blocks from submission and reference summaries respectively whose equivalence we want to check. We discuss the general case when these blocks contain nested loops. Note that due to the structural comparison explained above, these will have to have the same depth and directions. DPAssist checks the equivalence of iteration spaces and loop bodies separately. We identify the correspondence between the loop index variables at the same depth in B_S and B_R. This together with the variable correspondence ρ between the summaries gives us correspondence between array elements and expressions appearing in B_S and B_R. We rename the variables and array elements of B_S to primed versions of the matching ones from B_R. That is, $\mathtt{y[j]}$ in B_S is renamed to $\mathtt{x}'[\mathtt{i}']$ if $\mathtt{y}$ corresponds to $\mathtt{x}$ from B_R and $\mathtt{j}$ corresponds to $\mathtt{i}$ from B_R. Next, we replace array elements by fresh scalar variables. If we have $\mathtt{x[i]}$ in B_R and we replace it with a scalar variable $\mathtt{v}$ then in B_S, we replace $\mathtt{x}'[\mathtt{i}']$ by $\mathtt{v}'$. In order to identify matching expressions under commutative operations, we force a unique ordering among variables when they appear as operands of commutative operations. Thus, our renaming can successfully match $\mathtt{x[i+j]}$ and $\mathtt{x}'[\mathtt{j}' + \mathtt{i}']$. This simple step allows us to abstract away some more stylistic differences. If there is a variable that is updated in B_S but the corresponding one is not updated in B_R or vice versa, our algorithm discards the reference summary as unsuitable. The instructor may impose certain constraints over inputs (e.g., the cells of a matrix contain only positive numbers). DPAssist takes the constraints IC on the input variables as an input.

Algorithm 2: Algorithm GenFeedback

```
Input: Submission summary $S$, reference summary $R$, variable correspondence map $\rho$, constraints on input variables $IC$ and threshold value thresh
Output: Feedback $F$ and associated score $s$
$F \leftarrow$ "", $s \leftarrow 0$
foreach $i \in \{1, ..., \mathsf{numBlocks}(S)\}$ // Iterate over all blocks in the submission summary
do
    Let $B_S$ and $B_R$ be the $i^{th}$ blocks in $S$ and $R$ respectively
    if the loop headers of $B_S$ and $B_R$ are not syntactically equal and $\mathsf{EqIter}(B_S, B_R, IC)$ is not valid // Check equivalence of loop headers
    then
        $\langle L, U\rangle \leftarrow \mathsf{getDifference}(B_S, B_R)$
        foreach $(i, l) \in L$ do
            $f \leftarrow$ "In" + $\mathsf{label}(B_S)$ + "step, lower bound of" + $\mathsf{getAsSubExpr}(i)$ + "should be" + $\mathsf{getAsSubExpr}(l)$
            $F.\mathsf{append}(f)$, $s \leftarrow s + \mathsf{score}(f)$
        end
        foreach $(i, u) \in U$ do
            $f \leftarrow$ "In" + $\mathsf{label}(B_S)$ + "step, upper bound of" + $\mathsf{getAsSubExpr}(i)$ + "should be" + $\mathsf{getAsSubExpr}(u)$
            $F.\mathsf{append}(f)$, $s \leftarrow s + \mathsf{score}(f)$
        end
    end
    $F_b \leftarrow$ "", $s_b \leftarrow 0$
    foreach $i \in \{0, ..., \mathsf{thresh}\}$ // Counter-example guided feedback generation loop
    do
        $\alpha \leftarrow \mathsf{checkValid}(\mathsf{EqBody}(B_S, B_R, IC))$ // Check equivalence of loop bodies
        if $\alpha = \emptyset$ then break
        Let $\phi_S = g_S : s_S$ and $\phi_R = g_R : s_R$ be statements in $B_S$ and $B_R$ such that $\alpha \models g_S$ and $\alpha \models g_R$
        if $g_S \iff g_R$ is valid then
            $f \leftarrow$ "In" + $\mathsf{label}(B_S)$ + "step, under" + $\mathsf{getAsSubExpr}(g_s)$ + ", compute" + $\mathsf{getAsSubExpr}(s_R)$+"instead of" + $\mathsf{getAsSubExpr}(s_S)$
            Rewrite $\phi_S$ to $g_S : \mathsf{translate}(s_R)$
        else
            $f \leftarrow$ "In" + $\mathsf{label}(B_S)$ + "step, under" + $\mathsf{getAsSubExpr}(g_s \wedge g_R)$ + ", compute" + $\mathsf{getAsSubExpr}(s_R)$+
                "instead of" + $\mathsf{getAsSubExpr}(s_S)$
            Split $\phi_S$ to two statements $\phi_{S1} = g_S \wedge \mathsf{translate}(g_R) : \mathsf{translate}(s_R)$ and $\phi_{S2} = g_S \wedge \neg\mathsf{translate}(g_R) : s_S$
        end
        $F_b.\mathsf{append}(f)$, $s_b \leftarrow s_b + \mathsf{score}(f)$
    end
    if $i = \mathsf{thresh}$ or $\mathsf{isNotMemSafe}(\mathsf{B_S})$ then $F_b \leftarrow$ "In" + $\mathsf{label}(B_S)$ + "compute" + $B_R$ + "instead of" + $B_S$; $s_b \leftarrow \mathsf{score}(B_R)$ end
    $F.\mathsf{append}(F_b)$, $s \leftarrow s + s_b$
end
return $\langle F, s\rangle$
```

We check equivalence of iteration spaces by checking syntactic equality of the loop headers under the variable correspondences. Syntactic equality can be too strong a check. For example, suppose B_R is loop(i,1,n,+){*true*: s} and B_S is loop(i′,0,n,+){i′ > 0: s′}. B_R executes s for $1 \leq \mathtt{i} \leq \mathtt{n}$ and B_S executes s′ also for $1 \leq \mathtt{i}' \leq \mathtt{n}$. The syntactic check will end up reporting that B_S executes one additional iteration when i′ is 0. To avoid reporting such cases, we create a symbolic constraint. Let $\mathsf{idxcorr}(B_S, B_R)$ assert equality between the loop index variables at the same depths from the two blocks and correspondence between input variables. For a block B, $\mathsf{iter}(B)$ encodes the ranges of each loop index variable and $\mathsf{guards}(B)$ gives the disjunction of all guards present in the loop body of block B. The formula $\mathsf{EqIter}(B_S, B_R, IC)$ from Figure 4 asserts equivalence of the iteration spaces.

The formula $\mathsf{EqBody}(B_S, B_R, IC)$ in Figure 4 encodes our equivalence check for loop bodies whose aim is to establish that if the DP arrays of the submission and reference summaries are equal at the beginning of an iteration, they remain equal at the end. The precondition is captured in $\mathsf{varcorr}(B_S, B_R)$ which asserts equality of all scalar variables substituted for the DP array elements used in the loop body. In addition, $\mathsf{varcorr}(B_S, B_R)$ also encodes equality of summary variables and loop index variables. The postcondition we want to establish is: for any variable v updated in B_R, $\mathtt{v} = \mathtt{v}'$. Recall that due to the renaming of variables as explained above, v′ is the variable from B_S that corresponds to v. The relation between the values at the beginning and end of an iteration of a block B is given by $\mathsf{body}(B)$. For a block B, $\mathsf{body}(B)$ represents the conjunction of guarded assignments in B. A guarded assignment g: x = e is encoded as a constraint $\mathtt{g} \implies \mathtt{x} = \mathtt{e}$ where the assignment is converted to an equality constraint. If the variable x is also used on the RHS then we rename the LHS occurrence to a fresh variable, say $\mathtt{x}_0$, in the equality constraint above.

3.2.2 Algorithm

We now focus on generating feedback for a pair of summaries which satisfy the structural constraints and for which the variable renaming transformations described in Section 3.2.1 are already applied. Algorithm 2 presents our algorithm GenFeedback to generate feedback for a submission summary S wrt a reference summary R given a variable correspondence map ρ, input constraints IC and a positive constant thresh. thresh is used for controlling the complexity of feedback as explained later. The declaration blocks are handled earlier to generate variable correspondences and renaming. The input summaries to GenFeedback do not contain them. The output of GenFeedback is a feedback string comprising possibly multiple corrections and a score. The score is not displayed to the student but is used for identifying feedback wrt the closest reference summary as described in Section 3.2.1.

The feedback string F and the score s are initialized in line 1. The algorithm then goes over a pair of corresponding blocks (B_S, B_R) from submission and reference summaries respectively (line 2). If the blocks contain loops, the algorithm checks equivalence of the loop iteration spaces in line 5. If they are not equivalent, it calls getDifference (line 7) which outputs two sets L and U which are sets of pairs of the form (i, b) where i is a loop index variable in B_S and b is the correct lower/upper bound for the corresponding loop index variable as specified in B_R. L gives the set of lower bounds and U gives the set of upper bounds. Then, the algorithm goes over each pair in L and U (lines 8 and 12) and generates feedback text (lines 9 and 13) describing the correct lower/upper bound of each loop index variable. In the feedback text, the function $\mathsf{label}(B_S)$ refers to the type of the summary block (e.g., init, update, etc.). The algorithm renames the summary variables to the actual variables in the student submission using the function getAsSubExpr. The generated feedback f is accumulated in F (line 14). If the blocks do not contain loops, the algorithm skips lines 5–16.

At line 17, it initializes F_b and s_b to an empty string and 0 to track block-level feedback and score. The loop at lines 18–31 is a *counter-example guided iterative feedback generation* algorithm. In each iteration, it checks the equivalence of the loop bodies of B_S and B_R by checking validity of EqBody (line 20). The specialization of EqBody (Figure 4) to blocks which do not contain loops but only guarded assignments is straightforward. The function checkValid returns a counter-example α if its argument is not valid. If α is empty, the loop bodies are equivalent and the algorithm exits from the loop (line 21).

The algorithm uses the counter-example α to localize the fault in B_S and the matching statement in B_R. Let $\phi_S = g_S : s_S$ and $\phi_R = g_R : s_R$ be the statements from B_S and B_R such that $\alpha \models g_S$ and $\alpha \models g_R$ (line 22). There are two possibilities: (1) the guards are equivalent and the assignment statements disagree or (2) the guards themselves disagree. In the former case, the algorithm states that under the guard g_S, the submission should compute s_R instead of s_S (line 24). In the latter case, it states that under the guard $g_S \wedge g_R$, the submission should compute s_R instead of s_S (line 27). The summaries can be manipulated easily. GenFeedback rewrites the faulty statement ϕ_S to $g_S : \mathsf{translate}(s_R)$ in the former case (line 25). The function translate translates an assignment or a guard from B_R to use the corresponding variables of B_S. In the latter case, at line 28, ϕ_S is split into two statements in B_S: (1) $\phi_{S1} = g_S \wedge \mathsf{translate}(g_R) : \mathsf{translate}(s_R)$ and (2) $\phi_{S2} = g_S \wedge \neg\mathsf{translate}(g_R) : s_S$. The statement ϕ_{S1} performs the computation that B_S would do under the original guard g_S and the guard suggested in the feedback. The statement ϕ_{S2} continues to behave like ϕ_S but under $g_s \wedge \neg\mathsf{translate}(g_R)$. The feedback and score are accumulated at line 30 and the loop continues.

The counter-example guided feedback generation loop terminates when no more counter-examples can be found (line 21) and thus, progressively finds *all* semantic differences between the submission and reference summaries. Each iteration eliminates the semantic difference between a pair of statements from B_S and B_R that gave rise to the counter-example α 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. The constant thresh provided by the instructor controls this. The function isNotMemSafe checks whether the accesses to array indices within a loop are all within the declared size of the array. If the threshold has been reached or the submission summary B_S here is not memory safe, the algorithm discards the feedback generated so far in the loop and simply returns the entire reference summary block B_R (line 32). The feedback and scores for the two blocks are updated at line 33. Finally, the algorithm returns the feedback F and the score s across all blocks.

Table 1: Results of summary and feedback generation

	SUMTRIAN			MGCRNK		
	Total	AC	WA	Total	AC	WA
#Submissions	364	300	64	154	133	21
#Summarized	341	291	50	119	101	18
#Feedback Obtained	306	264	42	106	89	17
#References		23			14	

Due to the explicit verification of summary equivalence, our algorithm only generates correct feedback. Each iteration of the feedback generation loop increases the size of guards present in the submission summary. To limit this, our algorithm uses *simplification tactics* present in the SMT solvers to simplify the guards.

4. IMPLEMENTATION AND EVALUATION

We have implemented DPAssist using the Clang front-end of LLVM [24] to obtain ASTs and CFGs of programs. DPAssist performs pattern matching on the ASTs and structural analysis over the CFGs to identify summary blocks, as described in Section 3.1. For summary inference, common library functions such as memset are modeled using summary language constructs. The feedback generation module then parses the summaries obtained using the Antlr3 framework [32], and compares them against reference implementations using Z3 [8] for SMT solving. The module implements the feedback generation algorithm described in Section 3.2.

4.1 Experimental Setup

To assess the performance and effectiveness of DPAssist, we created a benchmark using programs submitted to CodeChef. The programs were submitted as solutions to the following two DP problems: (1) SUMTRIAN, described in the Introduction section and (2) MGCRNK[2], which asks students to 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, moves to cells to the right or below are permitted.

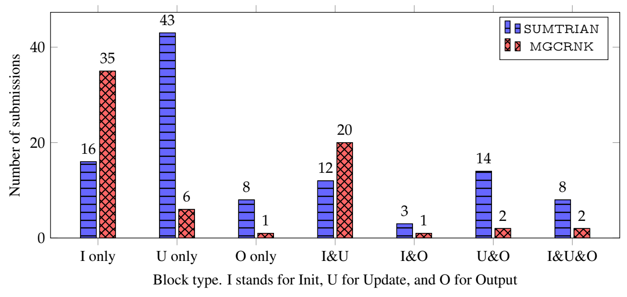
We selected submissions to these problems that implemented an iterative DP strategy in C. We filtered out submissions that used pointer arithmetic and solutions that had loop carried dependencies on scalars other than the loop index variables. We picked the latest submissions from individual users. CodeChef tags submissions as "WA" (wrong) or "AC" (correct) using test cases. For several users, the latest submissions were tagged WA. We believe they represent their best efforts, and are in clear need of feedback. For automating testing on CodeChef, the solutions had an outermost loop to iterate over test cases – we identified and removed this loop automatically.

We gathered 518 submissions in all, with 364 from SUMTRIAN and 154 from MGCRNK, together from 463 students representing over 100 institutions. The submissions employ a wide range of coding idioms and many possible solution approaches, both correct and incorrect. Thus, this real-world benchmark used for our experimental evaluation is large, diverse and challenging. We wrote reference implementations for evaluation, as described in RQ3.

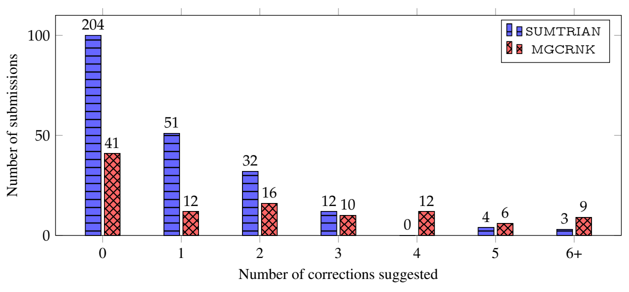
4.2 Results

(RQ1 - Effectiveness). Table 1 shows DPAssist's effectiveness on our benchmark. We ran our experiments on an Intel Xeon E5-1620 3.60 GHz machine with 8 cores and 24GB RAM. DPAssist runs only on a single core. DPAssist generated correct summary *as well as* feedback on 79.5% submissions across both problems and WA/AC categories in 5.3s on average. Of all the submissions,

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



(a)



(b)

Figure 5: Count of submissions by (a) the types of summary blocks that required corrections, and (b) the total number of corrections suggested by DPAssist across all blocks: zero corrections means the program was verified to be correct.

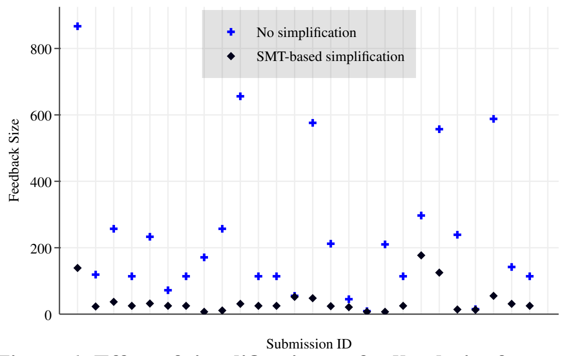


Figure 6: Effect of simplification on feedback size for MGCRNK

DPAssist provided appropriate corrections as feedback for 69.4% of all WA submissions and 81.5% of all AC submissions in 1.9s on average. It successfully computed summaries on 88.9% of the submissions in 3.4s on average. The remaining submissions either had a non-tightly nested loop within a block, or had statements that could not be assigned labels by our technique.

For some submissions, DPAssist generated correct summaries but could not provide feedback because: (1) the student either used an esoteric solution strategy or an arbitrarily wrong strategy and we did not have corresponding references or (2) the submission had constructs which the SMT solver could not handle. We assess the feedback quality shortly. DPAssist identified residual statements in 12 cases and in all but one, they were indeed spurious.

(RQ2 - Benefits to Students). A desirable goal for an automated feedback generation technique like DPAssist is to *customize* the feedback to the student's submission to the extent possible. The feedback provided should not force changing every submission to conform to a single solution strategy. To test whether DPAssist is able to provide feedback customized to individual submissions, we manually identified the closest reference implementation for each submission. For all submissions, DPAssist selected the same reference that we had identified manually as being the closest.

Students may make *multiple and even inter-related errors* of diverse nature in the same program. We evaluated how DPAssist works in such cases. For 62 submissions, DPAssist provided feedback to correct faults in multiple summary blocks (logical steps) (See Figure 5(a)), and it provided more than one correction to 104 submissions as feedback (See Figure 5(b)). The results show that DPAssist provides feedback to correct a wide range of errors, even if they occur *simultaneously*, and *localizes feedback* to address incorrect components of the solution, while *validating correct components*. The form of corrections suggested included fixing incorrect loop headers, initialization mistakes including missing and spurious initialization, missing cases in the DP recurrence, errors in expressions and guards, incorrect dimensions, etc. Studying the feedback text from our benchmarks revealed, for instance, that many students struggled with getting the corner cases such as the ones illustrated in Section 2 correct. For some submissions tagged "WA" by CodeChef, we were able to validate all components. On investigation, we found that the program logic was correct, as validated by DPAssist - the bugs were localized to output formatting, or in custom input/output functions.

Failing to identify faults in student submissions hurts students since they may not realize their mistakes. For AC submissions, DPAssist proved equivalence with a reference in 237 cases. Out of the rest, DPAssist identified spurious initializations in 30 cases. DPAssist was able to provide more interesting feedback asking users to rectify *faulty logic* that lead to idempotent computations, or code filling in entries in the DP table beyond what the problem asked for in 48 cases. Importantly, DPAssist detected an *out-of-bounds array access* in 3 cases, and suggested appropriate corrections as feedback. Thus, the static technique of DPAssist has a qualitative advantage over the test-based approach of online judges. DPAssist forced 28 submissions to conform to existing reference implementations even though they were already correct. We call these suggestions as spurious feedback and we observed them in 28 cases (~8% of all AC submissions).

Many students, especially beginners, write programs with convoluted conditional control flow, and unnecessarily complex expressions. Moreover, our counter-example guided feedback generation algorithm may generate complex guards. To present *clear and concise feedback* even in the face of these challenges, we simplify guards using the SMT solver. We use Z3's tactics to remove redundant clauses, evaluate sub-expressions to Boolean constants, and simplify systems of inequalities. Figure 6 quantifies the impact of the simplifications on feedback size in the case of MGCRNK. We measure the effectiveness of the simplification by disabling it, and using the sum of AST sizes (#nodes in the AST) of the guards in our feedback text as *feedback size*. The figure excludes cases where simplification had no impact on feedback size. Simplifications ensured that the feedback size was at most 177, and 24 on average. Without simplification, the maximum feedback size was 867. Simplification, where applicable, reduced feedback size by 57.8% (resp. 83.6%) on average for SUMTRIAN (resp. MGCRNK).

(RQ3 - Benefits to Instructors). DPAssist relieves the instructor from the *burden of manually triaging hundreds of submissions*. The instructor can use DPAssist to generate feedback for the submissions in an automated fashion by providing just a few (tens of) reference implementations. Table 1 shows that every reference implementation could provide feedback for 13 (resp. 7) submissions on average in the case of SUMTRIAN (resp. MGCRNK).

We first wrote an implementation of a recurrence relation solving the problem. We then added more references that had variations such as 0 vs. 1 based array indexing, column vs. row based traversal of the DP array, and direction of the loop nests. On encountering a submission that none of the existing references could validate, we studied the solution and implemented its strategy as a new reference, and iterated this procedure. While we wrote them ourselves for the purposes of the experimental evaluation, the reference implementations may be written by the instructor/TAs, or reused from previous offerings of a course.

DPAssist allows the instructor to obtain *a bird's eye view of the multitude of solution strategies and student difficulties at scale*. For example, DPAssist can be used to find the most popular strategy used by student submissions. The instructor can also use the statistics aggregated from DPAssist's output (e.g., shown in Figure 5) to tailor future recitation sessions to explain, say, lesser used, but pedagogically interesting strategies or common errors.

Limitations. DPAssist cannot generate sound summaries for programs that have loop carried dependencies over scalar variables apart from the loop index variables, and programs that use auxiliary arrays. Our approach requires a separate reference implementation for each solution strategy to the DP problem. Variations in strategy include using the input array in-place as the DP array, update loops that differ in direction and stride, and strategies that either split or fuse loops. We inherit the limitations of SMT solvers in reasoning about non-linear constraints and program expressions that are ill-defined mathematically, such as division by 0. We did not observe non-linear constraints in our experiments. Our approach cannot suggest feedback for errors in custom input/output functions, output formatting, typecasting etc. Our approach may provide spurious feedback to correct submissions enforcing stylistic conformance. Although our experiments show that this is rare, future work is to design heuristics to filter these out. Finally, our implementation currently handles only a frequently used subset of C syntactic constructs and library functions.

Threats to Validity. There may be faults in our implementation that might have affected our results. To address this threat, we manually checked the summaries and feedback obtained, and did not encounter any error. Our evaluation used reference implementations we wrote. To mitigate the threat of incorrect reference implementations, we scrutinized each carefully. Our results may not generalize to a different set of reference implementations. Threats to external validity arise because our results may not generalize to other student submissions and problems. We mitigated this threat by drawing upon submissions from more than 400 students from different institutions. 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.

5. RELATED WORK

Automated Feedback Generation. The idea of comparing instructor provided reference implementations with student submissions appears in [1]. It uses graph representation and transformations for comparison of Fortran programs. Xu and Chee [43] use richer graph representations for object-oriented programs. Rivers and Koedinger [38] use edit distance as a metric to compare graphs and generate feedback, while Gross et al. [12] cluster student solutions by structural similarity, and use a correct solution that is "close" to the buggy solution to provide feedback. In contrast, DPAssist performs more powerful abstractions in terms of summaries and uses symbolic reasoning but for the restricted domain of DP.

Alur et al. [3] develop a technique to automatically grade DFA constructions. Our work targets the algorithmically challenging class of DP assignments and generates corrections using reference implementations, whereas they search over a pre-defined set of corrections over automata. Singh et al. [40] 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. [14] address the orthogonal issue of providing feedback to address performance issues, while recent work by Srikant and Aggarwal [41] uses machine learning to automatically grade programs by determining the closeness of the program to the correct program using code features. The approach of [41] is aimed at helping companies assess prospective employees, and does not provide feedback on incorrect solutions.

Automated Program Repair. Genetic programming and mutations have been used to automatically generate repairs that make failing test cases pass and do not break any passing test case [4, 25, 9]. These approaches are not directly applicable in our setting as the search space of mutants is very large. Further, GenProg [25] uses only statements from the program to generate repairs, and therefore, relies on redundancy present in other parts of the code for fixing faults. This condition is not met in our setting. Long and Rinard [27] proposed an approach that generates candidate repairs by using a set of transformation schemas as a first step. The candidates are parameterized by an abstract condition that is synthesized next. Although the patch generated by this approach could contain multiple lines, it applies to exactly one program location. Repairs that modify multiple locations are not in their search space.

Recent approaches that use tests to infer specifications and propose repairs include SemFix [31], MintHint [22] and DirectFix [28]. These approaches use synthesis [20], symbolic execution [7] and partial MaxSAT [8] respectively. In contrast, we use reference implementations and our technique uses a combination of pattern matching, static analysis and SMT solving.

Konighopher et. al. [23] present an approach for automated error localization and correction of imperative programs using reference implementations. They use model-based diagnosis to localize the faulty component and use template-based approach for suggesting repairs. Their fault model only considers the right hand side of assignment statements as replaceable components, whereas our approach can even find missing statements in the program. There exist many approaches that rely on program specifications for repair, including contracts [34, 42], LTL [21], assertions [39] and pre-post conditions [11, 26, 15]. In our setting, writing reference implementations is much easier than writing such specifications.

All these approaches can be applied to larger classes of programs than DPAssist which targets only iterative DP programs.

Program Equivalence. Automated program equivalence between a program and its optimized version has been studied in translation validation [35, 30, 5]. Partush and Yahav [33] design an abstract interpretation based technique to check equivalence of a program and its patched version. In comparison, our technique permits equivalence check between programs written by *different individuals* independently but only for iterative DP programs.

6. CONCLUSIONS AND FUTURE WORK

We presented and evaluated DPAssist, an automated technique to generate feedback for student submissions implementing iterative DP algorithms, using a set of reference implementations. We want to extend the summary inference and counter-example guided iterative feedback generation algorithms of DPAssist to other algorithmic techniques such as greedy and divide-and-conquer.

7. REFERENCES

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

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

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

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

[5] C. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck. Tvoc: A translation validator for optimizing compilers. In *Proceedings of the 17th International Conference on Computer Aided Verification*, CAV'05, pages 291–295, Berlin, Heidelberg, 2005. Springer-Verlag.

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

[7] 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.

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

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

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3), July 1987.

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

[12] S. Gross, X. Zhu, B. Hammer, and N. Pinkwart. Cluster based feedback provision strategies in intelligent tutoring systems. Proceedings of the 11th international conference on Intelligent Tutoring Systems, pages 699–700, 2012.

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

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

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

[16] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1), Jan. 1990.

[17] 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.

[18] 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.

[19] M. A. Iqbal and A. Alvi. The magic of dynamic programming. *Innovations 2004: World Innovations in Engineering Education and Research*, pages 409–418, 2004.

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

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

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

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

[24] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[25] 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.

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

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

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

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

[30] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM.

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

[32] T. Parr and K. Fisher. Ll(*): The foundation of the ANTLR parser generator. In *PLDI*, pages 425–436, 2011.

[33] N. Partush and E. Yahav. Abstract semantic differencing via speculative correlation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '14, pages 811–828, New York, NY, USA, 2014. ACM.

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

[35] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 151–166, London, UK, UK, 1998. Springer-Verlag.

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

[37] W. Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8), Aug. 1992.

[38] K. Rivers and K. Koedinger. Automatic generation of programming feedback: A data-driven approach. In *AIED*, pages 4:50–4:59, 2013.

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

[40] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming

assignments. In *PLDI*, pages 15–26, 2013.

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

[42] 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.

[43] 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.