Search, Align, and Repair: Data-Driven Feedback Generation for Introductory Programming Exercises

Ke Wang
University of California, Davis
kbwang@ucdavis.edu

Rishabh Singh
Microsoft Research
risin@microsoft.com

Zhendong Su
University of California, Davis
su@cs.ucdavis.edu

Abstract

This paper introduces the "Search, Align, and Repair" data-driven program repair framework to automate feedback generation for introductory programming exercises. Distinct from existing techniques, our goal is to develop an efficient, fully automated, and problem-agnostic technique for large or MOOC-scale introductory programming courses. We leverage the large amount of available student submissions in such settings and develop new algorithms for identifying similar programs, aligning correct and incorrect programs, and repairing incorrect programs by finding minimal fixes. We have implemented our technique in the Sarfgen system and evaluated it on thousands of real student attempts from the Microsoft-DEV204.1x edX course and the Microsoft Code-Hunt platform. Our results show that Sarfgen can, within two seconds on average, generate concise, useful feedback for 89.7% of the incorrect student submissions. It has been integrated with the Microsoft-DEV204.1X edX class and deployed for production use.

1 Introduction

The unprecedented growth of technology and computing related jobs in recent years [25] has resulted in a surge in Computer Science enrollments at colleges and universities, and hundreds of thousands of learners worldwide signing up for Massive Open Online Courses (MOOCs). While larger classrooms and MOOCs have made education more accessible to a much more diverse and greater audience, several key challenges remain to ensure comparable education quality to that of traditional smaller classroom settings. This paper tackles one such challenge: *providing fully automated, personalized feedback to students for introductory programming exercises without requiring any instructor effort.* Even though introductory programming exercises require relatively small program size, relevant literature [7, 24] has shown that students still struggle and need effective tools to help them, further highlighting the need of automated feedback technology.

The problem of automated feedback generation for introductory programming courses has seen much recent interest — many systems have been developed using techniques from formal methods, programming languages, and machine learning. Most of these techniques model the problem as a *program repair* problem: repairing an incorrect student submission to make it functionally equivalent *w.r.t.* a given specification (*e.g.*, a reference solution or a set of test cases). Table 1 summarizes some of the major techniques in terms of their capabilities and requirements, and compares them with our proposed technique realized in the Sarfgen[1] system.

Approach	No Manual Effort	Minimal Repair	Complex Repairs	Data-Driven	Production Deployment
AutoGrader [24]	✗	✓	✗	✗	✗
CLARA [11]	✓	✗	✓	✓	✗
QLOSE [3]	✗	✗	✗	✗	✗
sk_p [22]	✓	✗	✓	✓	✗
REFAZER [23]	✓	✗	✗	✓	✗
CoderAssist [16]	✗	✗	✓	✓	✗
Sarfgen	✓	✓	✓	✓	✓

Table 1. Comparison of Sarfgen against the existing feedback generation approaches.

As summarized in the table, existing systems still face important challenges to be effective and practical. In particular, AutoGrader [24] requires a custom error model per programming problem, which demands manual effort from the instructor and her understanding of the system details. Moreover, its reliance on constraint solving to find repairs makes it expensive and unsuitable in interactive settings at the MOOC scale. Systems like CLARA [11] and sk_p [22] can generate repairs relatively quickly by using clustering and machine learning techniques on student data, but the generated repairs are often imprecise and not minimal, reducing the quality of the feedback. CLARA's use of Integer Linear Programming (ILP) for variable alignment during repair generation also hinders its scalability. Section 5 provides a detailed survey of related work.

To tackle the weaknesses of existing systems, we introduce "Search, Align, and Repair", a conceptual framework for program repair from a data-driven perspective. First, given an incorrect program, we search for similar reference solutions. Second, we align each statement in the incorrect program with a corresponding statement in the reference solutions to identify discrepancies for suggesting changes. Finally, we pinpoint minimal fixes to patch the incorrect program. Our key observation is that the diverse approaches and errors students make are captured in the abundant previous submissions because of the MOOC scale [5]. Thus, we aim at a fully automated, data-driven approach to generate *instant* (to be

PL'17, January 01–03, 2017, New York, NY, USA
2017.

[1] Search, Align, and Repair for Feedback GENeration

interactive), *minimal* (to be precise) and *semantic* (to allow complex repairs) fixes to incorrect student submissions. At the technical level, we need to address three key challenges:

Search: Given an incorrect student program, how to efficiently identify a set of closely-related candidate programs among all correct solutions?

Align: How to efficiently and precisely align each selected program with the incorrect program to compute a correction set that consists of expression- or statement-level discrepancies?

Repair: How to quickly identify a minimal set of fixes out of an entire correction set?

For the "Search" challenge, we identify *syntactically* most similar correct programs *w.r.t.* the incorrect program in a coarse-to-fine manner. First, we perform exact matching on the control flow structures of the incorrect and correct programs, and rank matched programs *w.r.t.* the tree edit distance between their abstract syntax trees (ASTs). Although introductory programming assignments often feature only lightweight small programs, the massive number of submissions in MOOCs still makes the standard tree edit distance computation too expensive for the setting. Our solution is based on a new tree embedding scheme for programs using numerical embedding vectors with a new distance metric in the Euclidean space. The new program embedding vectors (called *position-aware characteristic vectors*) efficiently capture the structural information of the program ASTs. Because the numerical distance metric is easier and faster to compute, program embedding allows us to scale to a large number of programs.

For the "Align" challenge, we propose a usage-based α-conversion to rename variables in a correct program using those from the incorrect program. In particular, we represent each variable with the new embedding vector based on its usage profile in the program, and compute the mapping between two sets of variables using the Euclidean distance metric. In the next phase, we split both programs into sequences of basic blocks and align each pair accordingly. Finally, we construct discrepancies by matching statements only from the aligned basic blocks.

For the final "Repair" challenge, we *dynamically* minimize the set of corrections needed to repair the incorrect program from the large set of possible corrections generated by the alignment step. We present some optimizations that gain significant speed-up over the enumerative search.

Our automated program repair technique offers several important benefits:

- **Fully Automated**: It does not require any manual effort during the complete repair process.
- **Minimal Repair**: It produces a *minimal* set of corrections (*i.e.*, any included correction is relevant and necessary) that can better help students.
- **Unrestricted Repair**: It supports both simple and complex repair modifications to the incorrect program without changing its control-flow structure.
- **Portable**: Unlike most previous approaches, it is independent of the programming exercise — it only needs a set of correct submissions for each exercise.

We have implemented our technique in the Sarfgen system and extensively evaluated it on thousands of programming submissions obtained from the Microsoft-DEV204.1x edX course and the CodeHunt platform [28]. Sarfgen is able to repair *89.7%* of the incorrect programs with *minimal* fixes in under two seconds on average. In addition, it has been integrated with the Microsoft-DEV204.1x course and deployed for production use. The feedback collected from online students demonstrates its practicality and usefulness.

This paper makes the following main contributions:

- We propose a high-level data-driven framework — search, align and repair — to fix programming submissions for introductory programming exercises.
- We present novel instantiations for different framework components. Specifically, we develop novel program embeddings and the associated distance metric to efficiently and precisely identify similar programs and compute program alignment.
- We present an extensive evaluation of Sarfgen on repairing thousands of student submissions on 17 different programming exercises from the Microsoft-DEV204-.1x edx course and the Microsoft CodeHunt platform.

2 Overview

This section gives an overview of our approach by introducing its key ideas and high-level steps.

```
X O X O X O X O
O X O X O X O X
X O X O X O X O
O X O X O X O X
X O X O X O X O
O X O X O X O X
X O X O X O X O
O X O X O X O X
```

Figure 1. Desired output for the chessboard exercise.

2.1 Example: The Chessboard printing problem

The Chessboard printing assignment, taken from the edX course of C# programming, requires students to print the pattern of chessboard using "X" and "O" characters to represent the squares as shown in Figure 1.

The goal of this problem is to teach students the concept of conditional and looping constructs. On this problem, students struggled with many issues, such as loop bounds or conditional predicates. In addition, they also had trouble

```
1 static void Main(string[] args){
2     for(int i = 0;i < 8;i++){
3         String chess = "";
4         for (int j = 0;j < 8;j++){
5             if (j % 2 == 0)
6                 chess += "X";
7             else
8                 chess += "0";
9         }
10        Console.Write(chess+"\n");
11    }}
```

(a)

```
1 static void Main(string[] args){
2     int length = 8;
3     bool ch = true;
4
5     for (int i = 0; i < length; i++)
6     {
7         for (int j = 0; j < length; j++)
8         {
9             if (ch) Console.Write("X");
10            else Console.Write("O");
11
12            ch = !ch;
13        }
14
15        Console.Write("\n");
16    }}
```

(b)

```
1 static void Main(string[] args){
2     for (int i = 1; i < 9; i++){
3         for (int j = 1; j < 9; j++){
4             if (i % 2 > 0)
5                 if (j % 2 > 0)
6                     Console.Write("O");
7                 else
8                     Console.Write("X");
9             else
10                if (j % 2 > 0)
11                    Console.Write("X");
12                else
13                    Console.Write("O");
14        }
15        Console.WriteLine();
16    }}
```

(c)

Figure 2. Three different incorrect student implementations for the chessboard problem.

```
static void Main(string[] args){
    for (int i = 0; i < 8; i++){
        String query = string.Empty;
        for (int j = 0; j < 8; j++){
            if ((i + j) % 2 == 0)
                query += "X";
            else
                query += "O";
        }
        Console.WriteLine(query);
    }}
```

(a)

```
static void Main(string[] args){
    bool blackWhite = true;
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            if (blackWhite)
                Console.Write("X");
            else
                Console.Write("O");
            blackWhite = !blackWhite;
        }

        blackWhite = !blackWhite;
        Console.WriteLine("");
    }}
```

(b)

```
static void Main(string[] args){
    for (int i = 0; i < 8; i++){
        for (int j = 0; j < 8; j++){
            if (i % 2 == 0)
                if (j % 2 == 0)
                    Console.Write("X");
                else
                    Console.Write("O");
            else
                if (j % 2 == 0)
                    Console.Write("O");
                else
                    Console.Write("X");
        }
        Console.WriteLine();
    }}
```

(c)

Figure 3. The respective top-1 reference solutions found by Sarfgen.

The program requires **2** changes:

- In **j % 2 == 0** on **line 5**, change **j** to **(i+j)**.
- In **chess += 0** on **line 8**, replace **0** to **O**.

(a)

The program requires **1** change:

- At **line 14**, add **ch = !ch**.

(b)

The program requires **1** change:

- In **i % 2 > 0** on **line 4**, change **>** to **==**.

(c)

Figure 4. The feedback generated by Sarfgen on the three incorrect student submissions from Figure 2.

understanding the functional behavior obtained from combining looping and conditional constructs. For example, one common error we observed among student attempts was that they managed to alternate "X" and "O" for each separate row/column, but (mistakenly) repeated the same pattern for all rows/columns (rather than flipping for consecutive rows/columns).

One key challenge in providing feedback on programming submissions is that a given problem can be solved using many different algorithms. In fact, among all the correct student submissions, we found 337 different control-flow structures, indicating the fairly large solution space one needs to consider. It is worth mentioning that modifying incorrect programs to merely comply with the specification using a reference solution may be inadequate. For example, completely rewriting an incorrect program into a correct program will achieve this goal, however such a repair might lead a student to a completely different direction, and would not help her understand the problems with her own approach. Therefore, it is important to pinpoint minimal modifications in their particular implementation that addresses the root cause of the problem. In addition, efficiency is important especially in an interactive setting like MOOC where students expect instant feedback within few seconds and also regarding the deployment cost for thousands of students.

Given the three different incorrect student submissions shown in Figure 2, Sarfgen generates the feedback depicted in Figure 4 in under two seconds each. During these two seconds, Sarfgen searches over more than two thousand reference solutions, and selects the programs in Figure 3 for each incorrect submission to be compared against. For brevity, we only show the top-1 candidate program in the comparison set. Sarfgen then collects and minimizes the set of changes required to eliminate the errors. Finally, it produces the feedback which consists of the following information (highlighted in bold in Figure 4 for emphasis):

- The number of changes required to correct the errors in the program.
- The location where each error occurred denoted by the line number.
- The problematic expression/statement in the line.
- The problematic sub-expression/expression that needs to be corrected.
- The new value of sub-expression/expression.

Sarfgen is customizable in terms of the level of the feedback an instructor would like to provide to the students. The generated feedback could consist of different combinations of the five kinds of information, which enables a more personalized, adaptive, and dynamic tutoring workflow.

2.2 Overview of the Workflow

We now briefly present an overview of the workflow of the Sarfgen system. The three major components are outlined in Figure 5: (1) *Searcher*, (2) *Aligner*, and (3) *Repairer*.

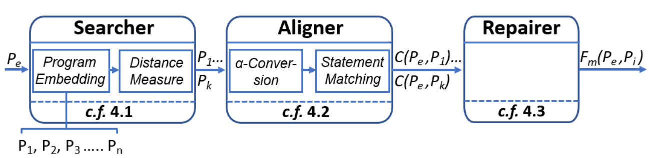


Figure 5. The overall workflow of the Sarfgen system.

(1) **Searcher**: The Searcher component takes as input an incorrect program P_e and searches for the top k closest solutions among all the correct programs $P_1, P_2, P_3, \ldots$ and P_n for the given exercise. The key challenge is to search a large number of programs in an efficient and precise manner. The Searcher component consists of:
   - **Program Embedder**: The Program Embedder converts P_e and $P_1, P_2, P_3, \ldots P_n$ into numerical vectors. We propose a new scheme of embedding programs that improves the original characteristic vector representation used previously in Jiang et al. [12].
   - **Distance Calculator**: Using program embeddings, the Distance Calculator computes the top k closest reference solutions in the Euclidean space. The advantage is such distance computations are much more scalable than the tree edit distance algorithm.

(2) **Aligner**: After computing a set of top k candidate programs for comparison, the Aligner component aligns each of the candidate programs $P_1, \ldots P_k$ *w.r.t.* P_e.
   - **α-conversion**: We propose a usage-set based α-conversion. Specifically, we profile each variable on its usage and summarize it into one single numeric vector via our new embedding scheme. Next, we minimize the distance between two sets of variables based on the Euclidean distance metric. This novel technique not only enables efficient computation but also achieves high accuracy.
   - **Statement Matching**: After the α-conversion, we align basic blocks, and in turn individual statements within each aligned basic blocks.

   Based on the alignment, we produce a set of syntactical discrepancies $C(P_e, P_k)$. This is an important step as misaligned programs would result in an imprecise set of corrections.

(3) **Repairer**: Given $C(P_e, P_k)$, the Repairer component produces a set of fixes $\mathcal{F}(P_e, P_k)$. Later it minimizes $\mathcal{F}(P_e, P_k)$ by removing the syntactic/semantic redundancies that are unrelated to the root cause of the error. We propose our minimization technique based on an optimized dynamic analysis to make the minimal repair computation efficient and scalable.

3 The Search, Align, and Repair Algorithm

This section presents our feedback generation algorithm. In particular, it describes the three key functional components: *Search*, *Align*, and *Repair* to cope with the challenges discussed in Section 1.

3.1 Search

To realize our goal of using correct programs to repair an incorrect program, the very first problem we need to solve is to identify a small subset of correct programs among thousands of submissions that are relevant to fixing the incorrect program in an efficient and precise manner. To start with, we perform exact matching between reference solutions and the incorrect program *w.r.t.* their control-flow structures.

Definition 3.1. (Control-Flow Structure) Given a program P, its control-flow structure, $CF(P)$, is a syntactic registration of how control statements in P are coordinated. For brevity, we denote $CF(P)$ to simply be a sequence of symbols (*e.g.* For_{start}, For_{start}, If_{start}, If_{end}, $Else_{start}$, $Else_{end}$, For_{end}, For_{end} for the program in Figure 2a).

Given the selected programs with the same control-flow structure, we search for similar programs using a syntactic approach (in contrast to a dynamic approach) for two reasons: (1) less overhead (*i.e.* faster) and (2) more fault-tolerant as runtime dynamic *traces* likely lead to greater differences if students made an error especially on control predicates. However, using the standard tree edit distance [27] as the

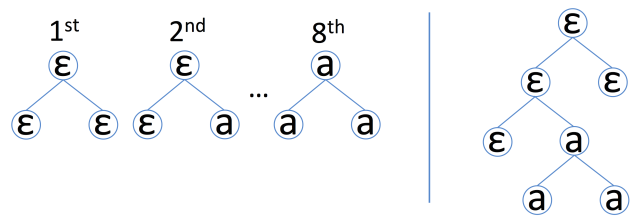


Figure 6. The eight 2-level atomic tree patterns for a label set of $\{a, \epsilon\}$ on the left. Using 2-level characteristic vector to embed the tree on the right: $\langle 1, 1, 0, 0, 0, 0, 0, 1\rangle$; and 2-level position-aware characteristic vector: $\langle\langle 4\rangle, \langle 3\rangle, ..., \langle 2\rangle\rangle$.

distance measure does not scale to a large number of programs. We propose a new method of embedding ASTs into numerical vectors, namely the position-aware characteristic vectors, with which Euclidean distance can be computed to represent the syntactic distance. Next, we briefly revisit Definitions 3.2 and 3.3 proposed in [12], upon which our new embedding is built.

Given a binary tree, we define a family of atomic tree patterns to capture structural information of a tree. They are parametrized by a parameter q, the height of the patterns.

Definition 3.2. (q-Level Atomic Tree Patterns) A q-level atomic pattern is a complete binary tree of height q. Given a label set L, including the empty label ϵ, there are at most $|L|^{2^q-1}$ distinct q-level atomic patterns.

Definition 3.3. (q-Level Characteristic Vector) Given a tree T, its q-level characteristic vector is $\langle b_1, ..., b_{|L|^{2^q-1}}\rangle$, where b_i is the number of occurrences of the i-th q-level atomic pattern in T.

Figure 6 depicts an example. Given the label set $\{a, \epsilon\}$, the tree on the right can be embedded into $\langle 1, 1, 0, 0, 0, 0, 0, 1\rangle$ using 2-level characteristic vectors. The benefit of such embedding schemes is the realization of gauging program similarity using *Hamming distance/Euclidean distance* metric which is much faster to compute. In order to further enhance the precision of the AST embeddings, we introduce a *position-aware characteristic vector* embedding which incorporates more structural information into the encoding vectors.

Definition 3.4. (q-Level Position-Aware Characteristic Vector) Given a tree T, its q-level position-aware characteristic vector is $\langle\langle b_1^1, ..., b_1^{n_1}\rangle, \langle b_2^1, ..., b_2^{n_2}\rangle, \langle b^L_{|L|^{2^q-1}}, ..., b_{|L|^{2^q-1}}^{n_{|L|^{2^q-1}}}\rangle\rangle$, where b_i^j is the height of the root node of the j-th occurrence of the i-th q-level atomic pattern in T.

Essentially we expand each element b_i defined in the q-level characteristic vector into a list of heights for each i-th q-level atomic pattern in T ($b_i = |\langle b_i^1, ..., b_i^{n_i}\rangle|$). Using a 2-level position-aware characteristic vector, the same tree in Figure 6 can be embedded into $\langle\langle 4\rangle, \langle 3\rangle, ..., \langle 2\rangle\rangle$. For brevity, we shorten the q-level position-aware characteristic vector to be $\langle h_{b_1}, ..., h_{b_{|L|^{2^q-1}}}\rangle$ where h_{b_i} represents the vector of heights of all i-th q-level atomic tree patterns. The distance between two q-level position-aware characteristic vector is:

$$\sqrt{\sum_{i=1}^{|L|^{2^q-1}} ||sort(h_{b_i}, \phi) - sort(h'_{b_i}, \phi)||_2^2} \tag{1}$$

$$\text{where } \phi = max(|h_{b_i}|, |h'_{b_i}|) \tag{2}$$

where $sort(h_{b_i}, \phi)$ means sorting h_{b_i} in descending order followed by padding zeros to the end if h_{b_i} happens to be the smaller vector of the two. The idea is to normalize h_{b_i} and h'_{b_i} prior to the distance calculation. $||...||_2^2$ denotes the square of the L2 norm. We are given the incorrect program (having m nodes in its AST) and ρ candidate solutions (assuming each has the same number of nodes n in their ASTs to simplify the calculation). Creating the embedding as well as computing the Euclidean distance on the resulting vectors has a worst-case complexity of $O(m + \rho * n + \rho * (|h_{b_1}|, ..., |h_{b_{|L|^{2^q-1}}}|))$. Because the position-aware characteristic vectors can be computed offline for the correct programs, we can further reduce the time complexity to $O(m + \rho * (|h_{b_1}|, ..., |h_{b_{|L|^{2^q-1}}}|))$. In comparison, the state-of-the art Zhang-Shasha tree edit distance algorithm [31] runs in $O(\rho * m^2 n^2)$. Our large-scale evaluation shows that this new program embeddings using position-aware characteristic vectors and the new distance measure not only leads to significantly faster search than tree edit distance on ASTs but also negligible precision loss.

3.2 Align

The goal of the align step is to compute Discrepancies (Definition 3.5). The rationale is that after a syntactically similar program is identified, it must be correctly aligned with the incorrect program such that the differences between the aligned statements can suggest potential corrections.

Definition 3.5. (Discrepancies) Given the incorrect program P_e and a correct program P_c, discrepancies, denoted by $C(P_e, P_c)$, is a list of pairs, (S_e,S_c), where S_e/S_c is a non-control statement[2] in P_e/P_c.

Aligning a reference solution with the incorrect program is a crucial step in generating accurate fixes, and in turn feedback. Figure 7 shows an example, in which the challenges that need to be addressed are: (1) renaming $s1$ in the correct solution to s — failing to do so will result in an incorrect let alone precise fix; (2) computing the correct alignment which leads to the minimum fix of changing *char s = 'O'* on line 3 in Figure 7a to *char s = 'X'*; and (3) solving the previous two tasks in a timely manner to ensure a good user experience. Our key idea for solving these challenges is to reduce the alignment problem to a distance computation problem,

[2] Hereinafter we denote non-control statement to include loop headers, branch conditions, *etc.*

specifically, of finding an alignment of two programs that minimizes their syntactic distances. We realize this idea in two-steps: variable-usage based α-conversion and two-level statement matching.

```
static void Main(string[] args){
    // change '0' to 'X'
    char s = '0';

    for (int i=0;i<8;i++){
        for (int j=0;j<8;j++){
            Console.Write(s);
            if (j < 7){
                if (s == 'X')
                    s = '0';
                else
                    s = 'X';
            }
        }

        Console.WriteLine();
}}
```

(a) An incorrect program.

```
static void Main(string[] args){
    string s1 = "X";
    string s2 = "";

    for (int i=0;i<8;i++){
        for (int j=0;j<8;j++){
            s2 = s2 + s1;
            if (j < 7){
                if (s1 == "X")
                    s = "0";
                else
                    s = "X";
            }}

        Console.WriteLine(s2);
        s2 = "";
}}
```

(b) A reference solution.

Figure 7. Highlighting the usage of variable s and $s1$ for aligning the two programs.

Variable-usage based α-Conversion The dynamic approach of comparing the runtime traces for each variable suffers from the same scalability issues mentioned in the search procedure. Instead, we present a syntactic approach — *variable-usage* based α-conversion. Our intuition is that how a variable is being used in a program serves as a good indicator of its identity. To this end, we represent each variable by profiling its usage in the program.

Definition 3.6. (Usage Set) Given a program P and the variable set $Vars(P)$, a usage set of a variable $v \in Vars(P)$ consists of all the non-control statements featuring v in P.

We then collect the usage set for each variable in P_e/P_c to form $\mathcal{U}(P_e)/\mathcal{U}(P_c)$. Now the goal is to find a one-to-one mapping between $Vars(P_e)$ and $Vars(P_c)$ which minimizes the distance between $\mathcal{U}(P_e)$ and $\mathcal{U}(P_c)$. Note that if the number of variables between the two programs $Vars(P_e)$ and $Vars(P_c)$ are different before alignment, new (existing) variables will be added (deleted) during the alignment step.

$$\textit{α-conversion} \;=\; \underset{Vars(P_c)\leftrightarrow Vars(P_e)}{\arg\min}\; \Delta(\mathcal{U}(P_c), \mathcal{U}(P_e)) \tag{3}$$

We can now compute the tree-edit distance between statements in any two usage sets in $\mathcal{U}(P_e)$ and $\mathcal{U}(P_c)$, and then find the matching that adds up to the smallest distance between $\mathcal{U}(P_e)$ and $\mathcal{U}(P_c)$. However, the total number of usage sets in $\mathcal{U}(P_e)$ and $\mathcal{U}(P_c)$ (denoted by the level of usage set) and the number of statements in each usage set (denoted by the level of statement) will lead this approach to a combinatorial explosion that does not scale in practice. Instead, we rely on the new program embeddings to represent each usage set with only one single position-aware characteristic vector. Using H_{v_i}/H'_{v_i} to denote the vector for usage set of $v_i \in Vars(P_e)/\ v'_i \in Vars(P_c)$, we instantiate Equation 3 into

$$\textit{α-conversion} \;=\; \underset{v_i\leftrightarrow v'_i}{\arg\min} \sum_{i=1}^{\omega} ||H_{v_i} - H'_{v_i}||_2 \tag{4}$$

$$\text{where } \omega \;=\; min(|Vars(P_c)|, |Vars(P_e)|) \tag{5}$$

The benefits of this instantiation are: (1) it only focuses on the combination at the level of usage set, and therefore eliminates a vast majority of combinations at the level of statement; and (2) it can quickly compute the Euclidean distance between two usage sets. Note the computed mapping between $Vars(P_c)$ and $Vars(P_e)$ in Equation 4 does not necessarily lead to the correct α-conversion. To deal with the phenomenon that variables may need to be left unmatched if their usages are too dissimilar, we apply the Tukey's method [29] to remove statistical outliers based on the Euclidean distance. After the α-conversion, we turn P_c into $P_{\alpha c}$.

Two-Level Statement Matching We leverage program structure to perform statement matching at two levels. At the first level, we fragment P_e and $P_{\alpha c}$ into a collection of basic blocks according to their control-flow structure, and then align each pair in order. This alignment will result in a 1-1 mapping of the basic blocks due to $CF(P_e) \equiv CF(P_{\alpha c})$. Next, we match statements/expressions only within aligned basic blocks. In particular, we pick the matching that minimizes the total syntactic distance (tree edit distance) among all pairs of matched statements within each aligned basic blocks. Figure 8 depicts the result of aligning programs in Figure 7.

3.3 Repair

Given $C(P_e, P_{\alpha c})$, we generate $\mathcal{F}(P_e, P_{\alpha c})$ as follows:

- **Insertion Fix**: Given a pair (S_e,S_c), if $S_e = \emptyset$ meaning S_c is not aligned to any statement in P_e, an insertion operation will be produced to add S_c.
- **Deletion Fix**: Given a pair (S_e,S_c), if $S_c = \emptyset$ meaning S_e is not aligned to any statement in P_c, a deletion operation will be produced to delete S_e.
- **Modification Fix**: Given a pair (S_e,S_c), if $S_c \neq \emptyset$ and $S_e \neq \emptyset$, a modification operation will be produced consisting the set of standard tree operations to turn the AST of S_e to that of S_c.

The computed set of potential fixes $\mathcal{F}(P_e, P_{\alpha c})$ typically contains a large set of redundancies that are unrelated to the root cause of the error in P_e. Such redundancies can be classified into two categories:

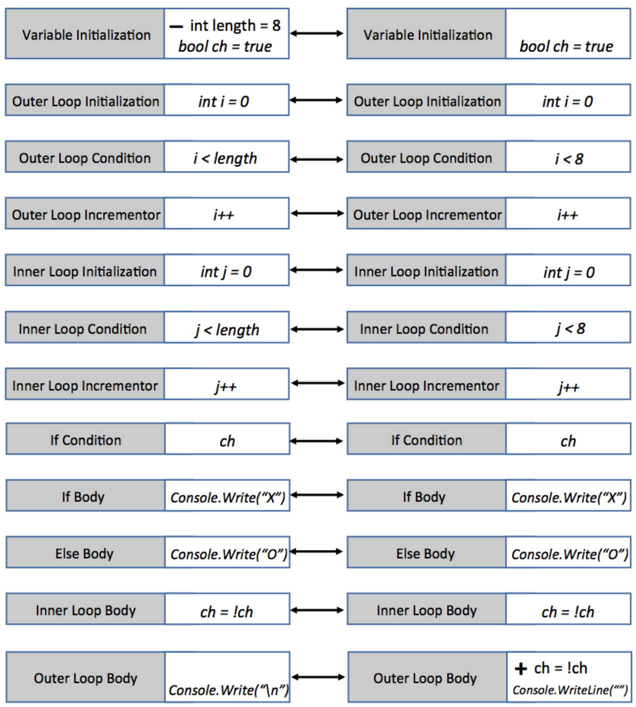


Figure 8. Aligning basic blocks for programs in Figure 2b and 3b after α-conversion (aligned blocks are connected by arrows); matching statements within each pair of aligned basic blocks (italicized statements denotes matching; statements annotated by +/- denotes insertion/deletion.).

- **Syntactic Differences:** P_e and $P_{\alpha c}$ may use different expressions in method calls, parameters, constants, *etc.* Figure 9a highlights such syntactic differences.
- **Semantic Differences:** More importantly, $\mathcal{F}(P_e, P_{\alpha c})$ may also contain semantically-equivalent differences such as different conditional statements yet logically equivalent *if* statement; different initialization and conditional expressions that evaluate to the same number of loop iterations; or expressions computing similar values composed of different variables (not variable names). Figure 9b highlights such semantic differences.

The Repair component is responsible for filtering out all those benign differences that are mixed in $\mathcal{F}(P_e, P_{\alpha c})$, in other words, picking $\mathcal{F}_m(P_e, P_{\alpha c})$ ($\mathcal{F}_m(P_e, P_{\alpha c}) \subseteq \mathcal{F}(P_e, P_{\alpha c})$) that only fix the issues in P_e while leaving the correct parts unchanged (Definition 3.7). A direct brute-force approach is to exhaustively incorporate each subset of $\mathcal{F}(P_e, P_{\alpha c})$ into the original incorrect program, and test the resultant program by dynamic execution. This approach yields $2^{|\mathcal{F}(P_e, P_{\alpha c})|} - 1$ number of trials in total. As the number of operations in $\mathcal{F}(P_e, P_{\alpha c})$ increases, the search space become intractable. Instead, we apply several optimization techniques to make the procedure more efficient and scalable (*cf.* Section 4).

Definition 3.7. (Minimality) Given an incorrect program P and a set of possible changes U, a set of changes $F \subset U$ to correct P is defined to be minimal if there does not exist F' *s.t.* $|F'| < |F|$ and F' fixes P. Correctly fixing P is *w.r.t.* a given test suite, *i.e.* the fixed program should pass all tests.

3.4 The Repair Algorithm in Sarfgen

We now present the complete repair algorithm in Sarfgen. The system first matches the control flow (Line 3-6); then selects top k candidates for repair (Line 7-12); Line 16-19 denotes the search, align and repair procedure, whereas Line 23 generates the feedback.

Algorithm 1: Sarfgen's feedback generation.

/* P_e: an incorrect program; P_s: all correct solutions; ℓ: feedback level */

1 **function** FeedbackGeneration(P_e, P_s, ℓ)
2 **begin**
/* identify P_{cs} that have the same control flow of P_e among all solutions P_s */
3 $P_{cs} \leftarrow \emptyset$
4 **for** $P \in P_s$ **do**
5 **if** $CF(P) = CF(P_e)$ **then**
6 $P_{cs} \leftarrow \{P\} \cup P_{cs}$
/* collect P_{dis} that consist of k most similar programs with P_e. */
7 $P_{dis} \leftarrow \emptyset$
8 **for** $P \in P_{cs}$ **do**
9 **if** $|P_{dis}| < k$ ||
10 $\mathcal{D}(P, P_e) < \mathcal{D}(P_{kth} \in P_{dis}, P_e)$ **then**
11 $P_{dis} \leftarrow P_{dis} \setminus \{P_{kth}\}$
12 $P_{dis} \leftarrow \{P\} \cup P_{dis}$
13 $n \leftarrow \infty$
14 $\mathcal{F}_m(P_e) \leftarrow$ null // minimum set of fix for P_e
15 **for** $P_c \in P_{dis}$ **do**
16 $P_{\alpha c} \leftarrow \alpha$-Conversion($P_e$, P_c)
17 $C(P_e, P_{\alpha c}) \leftarrow$ Discrepencies(P_e, $P_{\alpha c}$)
18 $\mathcal{F}(P_e, P_{\alpha c}) \leftarrow$ Fixes($C(P_e, P_{\alpha c})$)
19 $\mathcal{F}_m(P_e, P_{\alpha c}) \leftarrow$ Minimization($\mathcal{F}(P_e, P_{\alpha c})$)
20 **if** $|\mathcal{F}_m(P_e, P_{\alpha c})| < n$ **then**
21 $\mathcal{F}_m(P_e) \leftarrow \mathcal{F}_m(P_e, P_{\alpha c})$
22 $n \leftarrow |\mathcal{F}_m(P_e, P_{\alpha c})|$
/* translate fixes into feedback message. */
23 f = Translating($\mathcal{F}_m(P_e)$, ℓ)
24 **return** f

```
static void Main(string[] args){
    for(int i = 0;i < 8;i++){
        String chess = "";
        for (int j = 0;j < 8;j++){
            if (j % 2 == 0)
                chess += "X";
            else
                chess += "0";
        }
        Console.Write(chess+"\n");
    }}
```

```
static void Main(string[] args){
    for (int i = 0; i < 8; i++){
        String query = string.Empty;
        for (int j = 0; j < 8; j++){
            if ((i + j) % 2 == 0)
                query += "X";
            else
                query += "O";
        }
        Console.WriteLine(query);
    }}
```

(a) Syntactic differences between programs in Figures 2a and 3a.

```
static void Main(string[] args){
    for (int i = 1; i < 9; i++){
        for (int j = 1; j < 9; j++){
            if (i % 2 > 0)
                if (j % 2 > 0)
                    Console.Write("O");
                else
                    Console.Write("X");
            else
                if (j % 2 > 0)
                    Console.Write("X");
                else
                    Console.Write("O");
        }
        Console.WriteLine();
    }}
```

```
static void Main(string[] args){
    for (int i = 0; i < 8; i++){
        for (int j = 0; j < 8; j++){
            if (i % 2 == 0)
                if (j % 2 == 0)
                    Console.Write("X");
                else
                    Console.Write("O");
            else
                if (j % 2 == 0)
                    Console.Write("O");
                else
                    Console.Write("X");
        }
        Console.WriteLine();
    }}
```

(b) Semantic differences between programs in Figures 2c and 3c.

Figure 9. Syntactic and semantic differences.

4 Implementation and Evaluation

We briefly describe some of the implementation details of Sarfgen, and then report the experimental results on benchmark problems. We also conduct an in-depth analysis for measuring the usefulness of the different techniques and framework parameters, and conclude with an empirical comparison with the CLARA tool [11].

4.1 Implementation

Sarfgen is implemented in C#. We use the Microsoft Roslyn compiler framework for parsing ASTs and dynamic execution. We keep 5 syntactically most similar programs as the reference solutions. For the printing problem from Microsoft-DEV204.1X, we convert the console operation to string operation using the *StringBuilder* class. As for all the program embeddings, we use 1-level characteristic vectors only due to the suitable dimensionality. In other words, levels greater than one will yield characteristic vectors of excessively high dimensions (*i.e.* over millions for any realistic programming language such as C/C++/C#) that do not provide good efficiency/precision tradeoffs. All experiments are conducted on a Dell XPS 8500 with a 3rd Generation Intel Core®™ i7-3770 processor and 16GB RAM.

4.2 Results

We evaluate Sarfgen on the chessboard printing problem from Microsoft-DEV204.1X as well as 16 out of the 24 problems (the other seven are rarely attempted by students) on the CodeHunt education platform (ignoring the submissions that are syntactically incorrect).[3] Table 2 shows the results. Overall, Sarfgen generates feedback based on minimum fixes for 4,311 submissions out of 4,806 incorrect programs in total (≈ 90%) within two seconds on average. Our evaluation results validate the assumption we made since all the minimal fixes modify up to three lines only. Another interesting finding is that Sarfgen performed better on CodeHunt programming submissions than on Microsoft-DEV204.1x's assignment despite the fewer number of correct programs. After further investigation, we conclude that the most likely cause for this is the printing nature of Microsoft-DEV204.1x's exercise placing little constraint on the control-flow structure. In an extreme, we find students writing straight-line programs even in several different ways, and therefore those programs are more diverse and difficult for Sarfgen to precisely find the closest solutions. On the other hand, CodeHunt programs are functional and more structured. Even though there are fewer reference implementations, Sarfgen is capable of repairing more incorrect submissions.

4.3 In-depth Analysis

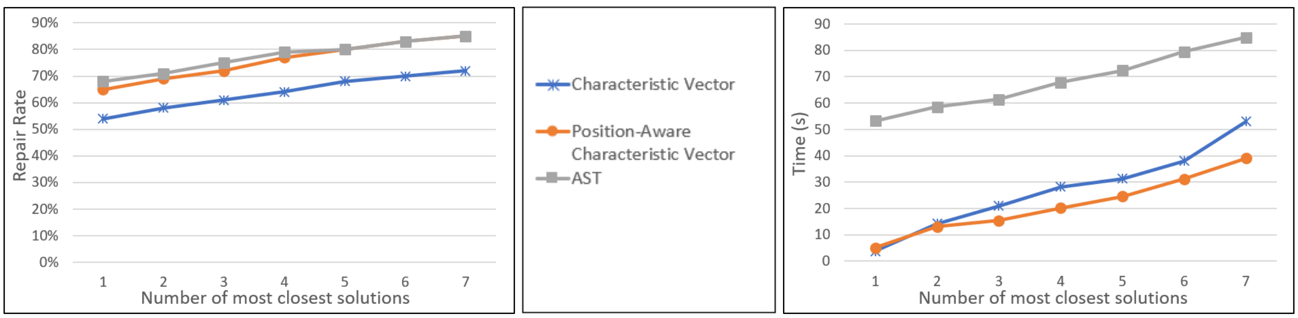
We now present a few in-depth experiments to better understand the contributions of different techniques for the search, align, and repair phases. First, we investigate the effect of different program embeddings adopted in the search procedure. Then, we investigate the usefulness of our α-conversion mechanism compared against other variable alignment approaches. For these experiments, we use two key metrics: (i) *performance* (*i.e.* how long does Sarfgen take to generate feedback), and (2) *capability* (*i.e.* how many incorrect programs are repaired with minimum fixes). To keep our experiments feasible, we focus on three problems that have the most reference solutions (*i.e.* Printing, MaximumDifference and FibonacciSum).

Program Embeddings As the number k of top-k closest programs used for the reference solutions increases, we compare the precision of our program embeddings using position-aware characteristic vector against (1) program embeddings using the characteristic vector and (2) using the original AST representation. We adopt a cut-off point of $\mathcal{F}_m(P_e, P_{\alpha c})$ to be three. Otherwise, if we let Sarfgen traverse the power set of $\mathcal{F}(P_e, P_{\alpha c})$, the capability criterion will be redundant. In all settings, Sarfgen adopts the usage-set based α-conversion via position-aware characteristic vectors. Also, Sarfgen runs without any optimization techniques. Figure 10 shows the results. The embeddings using

[3]Please refer to https://kbwang.bitbucket.io/misc/pldi18-paper93-supplemental-text.pdf for a brief description of each benchmark problem.

Benchmark	Average (LOC)	Median (LOC)	Correct Attempts	InCorrect Attempts	Generated Feedback on Minimum Fixes	Average Time Time (in s)	Median Time Time (in s)
Divisibility	4	3	609	398	379 (%95.2)	0.84	0.89
ArrayIndexing	3	3	524	449	421 (%93.8)	0.82	0.79
StringCount	6	4	803	603	562 (%93.2)	1.38	1.13
Average	8	7	551	465	419 (%90.1)	1.16	1.08
ParenthesisDepth	18	15	505	315	277 (%88.0)	2.25	2.32
Reversal	14	11	623	398	374 (%94.0)	2.10	1.77
LCM	15	10	692	114	97(%85.1)	2.07	2.14
MaximumDifference	11	8	928	172	155 (%90.1)	1.40	1.78
BinaryDigits	10	7	518	371	343 (%92.4)	2.29	2.01
Filter	12	8	582	88	71 (%80.7)	1.64	1.72
FibonacciSum	14	12	899	131	114 (%87.0)	2.78	2.45
K-thLargest	11	9	708	241	208 (%86.3)	1.51	1.16
SetDifference	16	13	789	284	236 (%83.1)	1.68	1.19
Combinations	14	10	638	68	56 (%82.4)	2.13	1.94
MostOnes	29	33	748	394	328 (%87.9)	2.72	2.15
ArrayMapping	16	14	671	315	271 (%86.0)	1.46	1.8
Printing	24	21	2281	742	526 (%70.9)	3.16	2.77

Table 2. Experimental results on the benchmark problems obtained from CodeHunt and edX course.



(a) Comparison on capability.

(b) Comparison on performance.

Figure 10. Position-Aware Characteristic Vector vs. Characteristic Vector.

(a) Comparison on capability.

(b) Comparison on performance.

Figure 11. Compare different α-conversion techniques based on the same set of reference solutions.

position-aware characteristic vectors are almost as precise as ASTs and achieves exactly the same accuracy as ASTs when the number of closest solutions equals five. In addition, the position-aware characteristic vector embedding consistently outperforms the characteristic vector embedding by more than 10% in terms of capability. Moreover, although position-aware characteristic vector uses higher dimensions to embed ASTs, it is generally faster due to the higher accuracy in selected reference solutions, and in turn lowering the computational cost spent on reducing $\mathcal{F}(P_e, P_{\alpha c})$ to $\mathcal{F}_m(P_e, P_{\alpha c})$. When SARFGEN uses fewer reference solutions, the two embeddings do not display noticeable differences since the speedup offered by the later is insignificant. Both embedding schemes are significantly faster than the AST representation.

***α*-Conversion** We next evaluate the precision and efficiency of the α-conversion in the align step. Given the same set of candidate solutions (identified by AST representation in the previous step), we compare our usage-based α-conversion via position-aware characteristic vector against (1) the same usage-based α-conversion via standard tree-edit operations and (2) a dynamic trace-based α-conversion via sequence alignment. We adopt the same configurations as in the previous experiment which is to set the cut-off point of $\mathcal{F}_m(P_e, P_{\alpha c})$ to be three and perform minimization without optimizations. As shown in Figure 11, our usage-set based α-conversion via position-aware characteristic vectors outperforms that via standard tree-edit operations by more than an order of magnitude at the expense of little precision loss measured by capability. In the meanwhile, dynamic variable-trace based sequence alignment displays the best performance at a considerable cost of capability mainly due to its poor tolerance against erroneous variable traces.

Optimizations One of the main performance bottlenecks in the repair phase is dynamic execution, where the largest performance hit comes from the traversal of power sets of $\mathcal{F}(P_e, P_{\alpha c})$. To tackle this challenge, we design and realize three key techniques that help prune the search space substantially.

- *Reachability-based Pruning:* We leverage the reachability of a fix to examine its applicability before it is attempted.[4] In particular, if a fix location is never reached on any execution path *w.r.t.* any of the provided inputs, it is safe to exclude it from the minimization procedure.
- *Co-ordinated Changes:* Certain corrections are coordinated, *i.e.* they should only be included/excluded simultaneously due to their co-occurrence nature. For example, when operations in $\mathcal{F}(P_e, P_{\alpha c})$ introduces a new variable ($v \in Vars(P_{\alpha c}) \wedge v \notin Vars(P_e)$) or delete existing variable ($v \notin Vars(P_{\alpha c}) \wedge v \in Vars(P_e)$), we bundle the operations that deletes/inserts an expression composed of Var together to be co-existing members that can either all be included or excluded in $\mathcal{F}_m(P_e, P_{\alpha c})$. The intuition is if a new/existing variable is inserted/deleted, it's illegitimate to separate the cascading expressions that are not sensible on its own, *i.e.* expressions composed of a deleted variable will cause compilation to fail.
- *Reusing Dynamic Executions:* The exhaustive dynamic executions on the power set of $\mathcal{F}(P_e, P_{\alpha c})$ can be used to recycle the computations to detect the syntactic/semantic redundancy. For example, while exhausting all subsets of one operation, even if we did not discover $\mathcal{F}_m(P_e, P_{\alpha c})$, we can detect the correction sets that are functionally redundant (*i.e.* do not change the semantics of incorrect program), and consequently remove them from the future computations. The more the number of iterations required to find $\mathcal{F}_m(P_e, P_{\alpha c})$, the larger the set of redundancies that can be discovered and eliminated resulting in a mutual beneficial relationship.

[4]Assuming test cases provided by the instructors are thorough and cover all corner cases.

According to our evaluation, these optimizations are able to gain approximately one order of magnitude speedup.

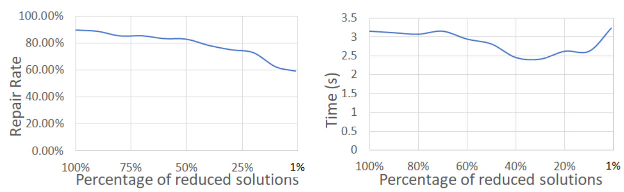
Minimization Effectiveness In Table 3, we show the effectiveness of SARFGEN's repair component by comparing the number of fixes pre- and post-minimization procedure.

Programming Problems	Number of Fixes Before Minimization	Number of Fixes After Minimization
Divisibility	2.1	1.6
ArrayIndexing	1.8	1.2
StringCount	3.5	1.9
Average	3.8	2.1
ParenthesisDepth	8.3	2.8
Reversal	6.5	2.3
LCM	8.2	3.1
MaximumDifference	5.5	2.3
BinaryDigits	6.4	2.1
Filter	7.9	3.5
FibonacciSum	7.6	3.1
K-thLargest	6.8	2.6
SetDifference	10.1	3.6
Combinations	9.7	3.6
MostOnes	10.8	4.2
ArrayMapping	9.5	3.2
Printing	12.7	3.5

Table 3. Evaluating SARFGEN's repair component.

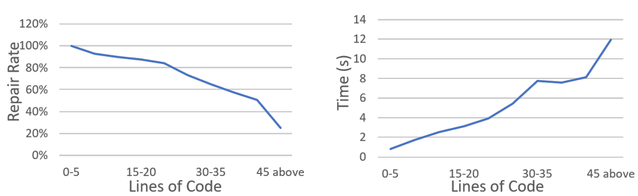
4.4 Reliance on Data

We conducted a further experiment to understand the degree to which SARFGEN relies on the correct programming submissions to have a reasonable utility. Initially, we use all the correct programs from all the programming problems, then we gradually down-sample them to observe the effects this may have on SARFGEN's capability and performance. Figure 12a/12b depicts the capability/performance change as the number of correct solutions is reduced from 100% to 1% under the standard configuration. In terms of capability, SARFGEN maintains almost the same power as the number of correct programs drops to half of the total. Even using only 1% of the total correct programs, SARFGEN still manages to produce feedback based on minimal fixes for almost 60% of the incorrect programs in total. The reason for this phenomenon is that the vast majority of students generally adopt an algorithm that their peers have correctly adopted. So even though the correct programs are down-sampled, there still exist some solutions of common patterns that can help a large portion of students who also attempt to solve the problem in a common way. Consequently, those students will not be affected. On the other hand, Sarfgen will understandably have more difficulties to deal with corner-case programming submissions. However, because the number of such programs is small, it generally does not have a severe impact. As for performance, the changes are twofold. On one hand, with fewer correct solutions, Sarfgen performs less computation. On the other hand, Sarfgen generally spends more time reducing $\mathcal{F}(P_e, P_{\alpha c})$ to $\mathcal{F}_m(P_e, P_{\alpha c})$ since the reference solutions become more dissimilar due to down-sampling. Since Sarfgen is able to find precise reference solutions for most of the incorrect programs when the number of reference solutions is not too low, *i.e.* above 1%, the final outcome is that Sarfgen becomes slightly faster.



(a) Capability trend. **(b)** Performance trend.

Figure 12. Capability and performance changes as the number of correct solutions decrease.



(a) Capability trend. **(b)** Performance trend.

Figure 13. Capability and performance changes as the size of program grows.

4.5 Scalability with Program Size

To understand how our system scales as the size of program increases, we conducted another experiment in which we partitioned the incorrect programs into ten groups according to size of the program, *i.e.* the number of lines in code. In terms of performance (Figure 13a), Sarfgen manages to generate the repairs within ten seconds in almost all cases. Regarding capability (Figure 13b), Sarfgen generates minimal fixes for close to 90% of incorrect programs when their size is under 20 lines of code and still over 50% when the size is less than 40 lines of code.

4.6 Comparison against CLARA

In this experiment, we conduct an empirical comparison against CLARA [11] using the same benchmark set. Since CLARA works on C programs, we followed the following procedure. First, we convert our C# programs into C programs that CLARA supports. In fact, we only converted the *Console* operation into *printf* for the Printing problem, as a result we have 488 out of 742 programs from edX that still compile, but only 395 out of 4,311 programs from CodeHunt since they generally contain more complex data structures. In total, we have 883 programs as the benchmark set for both systems to compare. Both systems use exactly the same set of correct programs in different languages. Second, because we experienced issues when invoking the provided clustering API[5] to cluster correct programs, we instead run CLARA on each correct program separately to repair the incorrect programs and select the fixes that are minimal and fastest (prioritize minimality over performance when necessary). On the other hand, Sarfgen is set up with standard configuration following Algorithm 1.

The results are shown in Table 4. As the solution set is down-sampled from 100% to 1%, Sarfgen generates consistently more minimal fixes than CLARA (Table 4a). For the results on performance shown in Table 4b, CLARA outperforms Sarfgen marginally on the programs of small size, *i.e.*, fewer than 15 lines of code, whereas CLARA scales significantly worse than Sarfgen when the size of programs grows, *i.e.*, slower by more than one order of magnitude when dealing with program of more than 25 lines. Regarding the performance comparison, in reality, CLARA usually compares an incorrect program with hundreds of reference solutions according to [11] to pick the smaller fixes, therefore the performance measured is a significant under-estimation. Furthermore, CLARA shows better performance in part due to the less work it undertakes since it does not guarantee minimal repairs.

4.7 User Study

The latest version of Sarfgen has been deployed onto the Microsoft-DEV204.1x course website on edX. Here we report the feedback users have submitted.

The goal of this study is to measure the usefulness of Sarfgen after being deployed to integrate with the C# edX course[6]. We study two research questions: 1) Can Sarfgen help the learning efficiency of the students? and 2) Do students consider the feedback generated by Sarfgen to be helpful? To answer the first question we randomly selected 200 hundred students based on the edX Id and evenly separated them into two groups (*i.e.* 100 students per group): pre-deployment (Group A) and post-deployment (Group B).

[5] The match command provided in the Example section at https://github.com/iradicek/clara produces unsound result
[6] http://mslexcodegrader.azurewebsites.net/

Percentage of Solutions	Number of Minimum Fixes (SARFGEN)	Number of Minimum Fixes (CLARA)
100%	688	573
80%	661	540
60%	647	492
40%	603	426
20%	539	349
1%	466	271

(a) Comparison on capability.

LOC (Total # of Programs)	Average Time Taken (SARFGEN)	Average Time Taken (CLARA)
0-5 (122)	1.14s	0.84s
5-10 (227)	1.38s	1.71s
10-15 (269)	2.69s	2.25s
15-20 (137)	3.02s	9.71s
20-25 (91)	3.81s	20.84s
25 or more (37)	5.26s	40.39s

(b) Comparison on performance.

Table 4. SARFGEN vs. CLARA on the same dataset.

Then we compare the interaction history across the two groups of students using two metrics: i) the number of submission iterations and ii) the length of time until they submitted a correct program. Results show that 96% of students in Group B (vs. 37% of students in Group A) completed their assignment within next two attempts. In addition 97% of post-deployment students (vs. 40% of pre-deployment students) finished within 30 minutes. This preliminary evidence supports that SARFGEN provides useful feedback. We also added a couple of qualitative metrics in terms of a student rating and comments about the provided feedback. The latest overall user rating on a 1-5 scale is 3.74, showing overall positive feedback.

5 Related Work

This section describes several strands of related work from the areas of automated feedback generation, automated program repair, fault localization and automated debugging.

5.1 Automated Feedback Generation

Recent years have seen the emergence of automated feedback generation for programming assignments as a new, active research topic. We briefly review the recent techniques.

AutoGrader [24] proposed a program synthesis based automated feedback generation for programming exercises. The idea is to take a reference solution and an error model consisting of potential corrections to errors student might make, and search for the minimum number of corrections using a SAT-based program synthesis technique. In contrast, SARFGEN advances the technology in the following aspects: (1) SARFGEN completely eliminates the manual effort involved in the process of constructing the error model; and (2) SARFGEN can perform more complex program repairs such as adding, deleting, swapping statements, *etc.*

CLARA [11] is arguably the most similar work to ours. Their approach is to cluster the correct programs and select a canonical program from each cluster to form the reference solution set. Given an incorrect student solution, CLARA runs a trace-based repair procedure *w.r.t.* each program in the solution set, and then selects the fix consisting of the minimum changes. Despite the seeming similarity, SARFGEN is fundamentally different from CLARA. At a conceptual level, CLARA assumes for every incorrect student program, there is a correct program whose execution traces/internal states only differs because of the presence of the error. Even though the program repairer generally enjoys the luxury of abundant data in this setting, there are a considerable amount of incorrect programs which yield new (partially) correct execution traces. Since the trace-based repair procedure does not distinguish a benign difference from a fix, it will introduce semantic redundancies which likely will have a negative impact on student's learning experience. As we have presented in our evaluation, CLARA scales poorly with increasing program size, and does not generate minimal repairs on the benchmark programs.

sk_p [22] was recently proposed to use deep learning techniques for program repair. Inspired by the skipgram model, a popular model used in natural language processing [20, 21], sk_p treats a program as a collection of code fragments, consisting of a pair of statements with a hole in the middle, and learns to generate the statement based on the local context. Replacing the original statement with the generated statement, one can infer the generated statement contains the fix if the resulting program is correct. However, sk_p suffers from low capability results, as the system only perform syntactic analysis. Another issue with the deep learning based approaches is low reusability. Significant efforts are needed to retrain new models to be applied across new problems.

QLOSE [3] is another recent work for automatically repairing incorrect solutions to programming assignments. The major contribution of this work is the idea of measuring the program distance not only syntactically but also semantically, *i.e.*, preserving program behavior regardless of syntactic changes. One way to achieve this is by monitoring runtime execution. However, the repair changes to an incorrect program is based on a pre-defined template corresponding to a linear combination of constants and all program variables in scope at the program location. As we have shown, more complex modifications are necessary for real-world benchmarks.

REFAZER [23] is another approach applicable in the domain of repairing program assignments. The idea is to learn a syntactic transformation pattern from examples of statement/expression instances before and after the modification.

Despite the impressive results, this approach also suffers from similar issues as QLOSE, *i.e.*, there are many incorrect programs that require changes more complex than simple syntactic changes.

CoderAssist [16] presents a new methodology for generating verified feedback for student programming exercises. The approach is to first cluster the student submissions according to their solution strategies and ask the instructor to identify a correct submission in each cluster (or add one if none exists). In the next phase, each submission in a cluster is verified against the instructor-validated submission in the same cluster. Despite the benefit of generating verified feedback, there are several weaknesses. As mentioned, CoderAssist requires manual effort from the instructor. More importantly, the quality of the generated feedback relies on how similar the provided solution is to the incorrect submissions in the same cluster. In contrast, SARFGEN searches through all possible solutions automatically and uses those that it considers to be the most similar to repair the incorrect program. In addition, CoderAssist targets dynamic programming assignments only. Its utility and scalability would need require further validation on other problems.

There is also work that addresses other learning aspects in the MOOC setting. For example, Gulwani *et al.* [10] proposed an approach to help students write more efficient algorithms to solve a problem. Its goal is to teach students about the performance aspects of a computing algorithm other than its functional correctness. However, this approach only works with correct student submissions, *i.e.*, it cannot repair incorrect programs. Kim *et al.* [17] is another interesting piece of work focusing on explaining the root cause of a bug in students' programs by comparing their execution traces. This approach works by first matching the assignment statement symbolically and then propagating to match predicates by aligning the control dependencies of the matched assignment statements. The key difference is that our work can automatically repair the student's code while Kim *et al.* [17] can only illustrate the cause of a bug.

5.2 Automated Program Repair

Gopinath *et al.* [8] propose a SAT-based approach to generate repairs for buggy programs. The idea is to encode the specification constraint on the buggy program into a SAT constraint, whose solutions lead to fixes. Könighofer and Bloem [18] present an approach based on automated error localization and correction. They localize faulty components with model-based diagnosis and then produce corrections based on SMT reasoning. They only take into account the right hand side (RHS) of the assignment statements as replaceable components. Prophet [19] learns a probabilistic, application-independent model of correct code by generalizing a set of successful human patches. There is also work [13, 26] that models the problem of program repair as a game. The two actors are the environment that provides the inputs and a system that provides correct values for the buggy expressions, so ultimately the specification is satisfied. These approaches use simple corrections (*e.g.*, correcting the RHS sides of expressions) since they aim to repair large programs with arbitrary errors. Another line of approaches use program mutation [4], or genetic programming [1, 6] for automated program repair. The idea is to repeatedly mutate statements ranked by their suspiciousness until the program is fixed. In comparison our approach is more efficient in pinpointing the error and fixes as those mutation-based approaches face extremely large search space of mutants (10^{12}).

5.3 Automated Debugging and Fault localization

Test cases reduction techniques like Delta Debugging [30] and QuickXplain [15] can complement our approach by ranking the likely fixes prior to dynamic analysis. The hope is to expedite the minimization loop and ultimately speed up performance. A major research direction of fault localization [2, 9] is to compare faulty and successful executions. Jose and Majumdar [14] propose an approach for error localization from a MAX-SAT aspect. However, such approaches suffer from their limited capability in producing fixes.

6 Conclusion

We have presented the "Search, Align, and Repair" data-driven framework for generating feedback on introductory programming assignments. It leverages the large number of available student solutions to generate instant, minimal, and semantic fixes to incorrect student submissions without any instructor effort. We introduce a new program representation mechanism using position-aware characteristic vectors that are able to capture rich structural properties of the program AST. These program embeddings allow for efficient algorithms for searching similar correct programs and aligning two programs to compute syntactic discrepancies, which are then used to compute a minimal set of fixes. We have implemented our approach in the SARFGEN system and extensively evaluated it on thousands of real student submissions. Our results show that SARFGEN is effective and improves existing systems *w.r.t.* automation, capability, and scalability. Since SARFGEN is also language-agnostic, we are actively instantiating the framework to support other languages such as Python. SARFGEN has also been integrated with the Microsoft-DEV204.1X edX course and the early feedback obtained from online students demonstrates its practicality and usefulness.

References

[1] Andrea Arcuri. 2008. On the Automation of Fixing Software Bugs. In *Companion of the 30th International Conference on Software Engineering*. ACM, New York, NY, USA, 1003–1006.

[2] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. 2003. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 97–105.

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

[4] Vidroha Debroy and W Eric Wong. 2010. Using mutation to automatically suggest fixes for faulty programs. In *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 65–74.

[5] Anna Drummond, Yanxin Lu, Swarat Chaudhuri, Christopher Jermaine, Joe Warren, and Scott Rixner. 2014. Learning to grade student programs in a massive open online course. In *2014 IEEE International Conference on Data Mining*. IEEE, 785–790.

[6] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A Genetic Programming Approach to Automated Software Repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*. ACM, New York, NY, USA, 947–954.

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

[8] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-based Program Repair Using SAT. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software*. Springer-Verlag, Berlin, Heidelberg, 173–188.

[9] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. 2006. Error explanation with distance metrics. *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004. Proceedings* 8, 3 (2006), 229–247.

[10] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2014. Feedback generation for performance problems in introductory programming assignments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 41–51.

[11] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2016. Automated Clustering and Program Repair for Introductory Programming Assignments. *arXiv preprint arXiv:1603.03165* (2016).

[12] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 96–105.

[13] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. 2005. Program Repair as a Game. In *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings*, Kousha Etessami and Sriram K. Rajamani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 226–238.

[14] Manu Jose and Rupak Majumdar. 2011. Cause Clue Clauses: Error Localization Using Maximum Satisfiability. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 437–446.

[15] Ulrich Junker. 2004. QUICKXPLAIN: Preferred Explanations and Relaxations for Over-constrained Problems. In *Proceedings of the 19th National Conference on Artifical Intelligence*. AAAI Press, 167–172.

[16] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-supervised Verified Feedback Generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, USA, 739–750.

[17] Dohyeong Kim, Yonghwi Kwon, Peng Liu, I. Luk Kim, David Mitchel Perry, Xiangyu Zhang, and Gustavo Rodriguez-Rivera. 2016. Apex: Automatic Programming Assignment Error Explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 311–327.

[18] Robert Könighofer and Roderick Bloem. 2011. Automated error localization and correction for imperative programs. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. IEEE, 91–100.

[19] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 298–312.

[20] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[21] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global Vectors for Word Representation.. In *EMNLP*, Vol. 14. 1532–1543.

[22] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. Sk_P: A Neural Program Corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. ACM, New York, NY, USA, 39–40.

[23] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 404–415.

[24] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 15–26.

[25] Taylor Soper. 2014. Analysis: the exploding demand for computer science education, and why America needs to keep up. Geekwire. (2014). http://www.geekwire.com/2014/analysis-examining-computer-science-education-explosion

[26] Stefan Staber, Barbara Jobstmann, and Roderick Bloem. 2005. Finding and Fixing Faults. In *Correct Hardware Design and Verification Methods: 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005. Proceedings*, Dominique Borrione and Wolfgang Paul (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 35–49.

[27] Kuo-Chung Tai. 1979. The tree-to-tree correction problem. *J. ACM* 26, 3 (1979), 422–433.

[28] Nikolai Tillmann, Jonathan de Halleux, Tao Xie, and Judith Bishop. 2014. Code hunt: gamifying teaching and learning of computer science at scale. In *Learning@Scale*. 221–222.

[29] John W Tukey. 1977. Exploratory data analysis. *Addison-Wesley Series in Behavioral Science: Quantitative Methods, Reading, Mass.: Addison-Wesley, 1977* (1977).

[30] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.

[31] Kaizhong Zhang and Dennis Shasha. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing* 18, 6 (1989), 1245–1262.