BRAFAR: Bidirectional Refactoring, Alignment, Fault Localization, and Repair for Programming Assignments

Linna Xie
Nanjing University
Nanjing, Jiangsu, China
xieln@smail.nju.edu.cn

Chongmin Li
Nanjing University
Nanjing, Jiangsu, China
chongminli@foxmail.com

Yu Pei*
The Hong Kong Polytechnic University
Hong Kong, China
yupei@polyu.edu.hk

Tian Zhang*
Nanjing University
Nanjing, Jiangsu, China
ztluck@nju.edu.cn

Minxue Pan
Nanjing University
Nanjing, Jiangsu, China
mxp@nju.edu.cn

ABSTRACT

The problem of automated feedback generation for introductory programming assignments (IPAs) has attracted significant attention with the increasing demand for programming education. While existing approaches, like *Refactory*, that employ the *"block-by-block"* repair strategy have produced promising results, they suffer from two limitations. First, *Refactory* randomly applies refactoring and mutation operations to correct and buggy programs, respectively, to align their control-flow structures (CFSs), which, however, has a relatively low success rate and often complicates the original repairing tasks. Second, *Refactory* generates repairs for each basic block of the buggy program when its semantics differs from the counterpart in the correct program, which, however, ignores the different roles that basic blocks play in the programs and often produces unnecessary repairs. To overcome these limitations, we propose the Brafar approach to feedback generation for IPAs. The core innovation of Brafar lies in its novel bidirectional refactoring algorithm and coarse-to-fine fault localization. The former aligns the CFSs of buggy and correct programs by applying semantics-preserving refactoring operations to both programs in a guided manner, while the latter identifies basic blocks that truly need repairs based on the semantics of their enclosing statements and themselves. In our experimental evaluation on 1783 real-life incorrect student submissions from a publicly available dataset, Brafar significantly outperformed *Refactory* and Clara, generating correct repairs for more incorrect programs with smaller patch sizes in a shorter time.

CCS CONCEPTS

• **Social and professional topics → Computing education**; • **Theory of computation → Program analysis**.

KEYWORDS

Program Repair, Programming Education, Software Refactoring

1 INTRODUCTION

The demand for programming education has witnessed significant growth across various college-level subjects, ranging from computer science to psychology [34]. Numerous students are enrolling in offline programming classrooms and Massive Open Online Courses (MOOCs) [26], making it challenging for instructors to provide personalized feedback in a timely manner [23, 37]. Effective feedback is crucial for learning, but accomplishing this task in large-scale settings becomes impractical. Simply providing failed test cases or an instructor's solution as feedback is often insufficient [7, 33]. To address this issue, researchers have explored automated feedback generation approaches to offer more guided feedback, helping students understand and resolve coding issues effectively.

One line of the work utilizes error models, manually constructed [32] or learned from data [30, 31], to correct student programming errors. However, they still require manual effort or suffer from low repair rates. Another line of the work, including Clara [13], Sarfgen [37], and *Refactory* [16], leverages a corpus of correct programs and applies a "*block-by-block*" repair strategy, focusing on aligning the control flow and data flow between incorrect and reference correct programs separately, achieving impressive repair results. Simultaneously, they make different efforts to generate smaller patches. More recent techniques [4, 23] introduce new repair strategies but come with their limitations, such as *AssignmentMender*'s reliance on automated fault localization (AFL) and *Verifix*'s need to match function and loop structures (see §5 for details). Therefore, we adhere to the "*block-by-block*" repair strategy in our approach.

The "*block-by-block*" repair strategy is a promising approach that leverages the local structure of code. It takes a buggy program and a correct reference program with an identical control-flow structure as input, aiming to *generate repairs for each basic block of the buggy program by replacing expressions with those from corresponding blocks in the reference program.* This narrows the search space and preserves the buggy program's overall structure. However, despite its potential, two primary challenges persist: (1) *How to utilize a correct program to repair a buggy program with a different control-flow structure?* and (2) *How to minimize the number of unnecessary block repairs to aid student comprehension?*

Our Approach: In response to the first challenge, we introduce a novel *bidirectional refactoring* algorithm. This algorithm offers bidirectional refactoring guidance for two programs with different control-flow structures, aligning the control-flow structures without introducing any semantic changes. This avoids the risk of introducing new errors to the buggy program. To tackle the second challenge, we integrate a *coarse-to-fine fault localization* approach along with *statement matching* techniques into the "*block-by-block*"

*Corresponding authors.

Table 1: Comparison of BRAFAR against the most related feedback generation approaches.

Approach	Control-flow Repair	Minimal Control-flow Repair	Minimal Block Repair	Minimal Repair
CLARA [13]	✗	✗	✗	✗
SARFGEN [37]	✗	✗	✓	✓
Refactory [16]	✓	✗	✓	✗
BRAFAR	✓	✓	✓	✓

repair strategy. This combination optimizes the repair process by reducing unnecessary repairs and improving overall repair quality.

Implementation and Evaluation: We have implemented the above-mentioned approaches in a tool named BRAFAR[1] and conducted experiments to evaluate its performance. Experiments with 1783 real-world incorrect student programs from the *Refactory* [16] benchmark have indicated that BRAFAR achieves a *higher repair rate, smaller patch size, and less overfitting* when compared to the state-of-art tools *Refactory* and CLARA. At the same time, we designed additional experiments to individually evaluate the performance of *bidirectional refactoring algorithm* and the *integrated repair strategy with fault localization*. The experimental results demonstrate that both of these components play a crucial role in reducing unnecessary repairs and improving the repair quality. Additionally, we conducted a preliminary evaluation on the performance of ChatGPT [1] to emphasize the relevance and importance of our work in the current context of large language models (LLMs).

Contributions: We summarize our contributions below:

- We propose a general feedback generation framework called BRAFAR for introductory programming assignments (IPAs).
- We present a *bidirectional refactoring* algorithm (including a novel *control-flow matching* algorithm) to align two different control-flow structures with small modifications and without altering semantics, facilitating the generation of high-quality repairs.
- We integrate a *coarse-to-fine fault localization* approach along with *statement matching* techniques into the "*block-by-block*" repair strategy to reduce unnecessary repairs.
- We evaluate BRAFAR in a realistic setting and make the tool publicly available, which can facilitate future research.

2 MOTIVATION

To better clarify the motivation behind our approach, we conducted an exploratory study on why existing techniques that employ the "*block-by-block*" approach encounter the two mentioned challenges. Table 1 summarizes their strengths and weaknesses. Among them, CLARA [13] and SARFGEN [37] fail to provide feedback for buggy programs with unique control-flow or loop structures. They assume that there is always a correct program available with an identical control-flow or loop structure to the buggy program. *Refactory* [16] addresses this by using refactoring rules to generate multiple correct solution variants, but this can lead to efficiency and effectiveness concerns: *Refactory* just randomly refactors the correct codes without proper guidance, creating a vast search space and potentially infinite code variants, making it difficult to generate

[1]Bidirectional Refactoring, Alignment, Fault Localization, and Repair

```
1  def search(x, seq):
2    for i,elem in enumerate(seq):
3      if seq == False:
4        return 0
5      elif x <= elem:
6        return i
7      elif i == (len(seq)-1):
8        return i+1
9      else:
10       continue
```

(a) A buggy program.

```
def search(x, seq):
  for i,elem in enumerate(seq):
    if elem < x:
      continue
    if elem == x:
      return i
    elif elem > x:
      return i
  return len(seq)
```

(b) A reference program.

```
1  def search(x, seq):
2    for i,elem in enumerate(seq):
3      if seq == False:
4        return 0
5      if elem == x:
6        return i
7      elif elem > x:
8        return i
9    return len(seq)
10
11
```

(c) *Refactroy*'s repair result.

```
def search(x, seq):
  for i,elem in enumerate(seq):
    if seq == False:
      return 0
    elif x <= elem:
      return i
    elif i == (len(seq)-1):
      return i+1
    else:
      continue
  return len(seq)
```

(d) BRAFAR's repair result.

Figure 1: Motivation example of real student submissions.

a match in finite time, particularly for complex programs. When *Refactory* fails,it attempts *structure mutation*. Similar to code differencing problem [11, 35], it modifies the control-flow structure of the buggy program to match that of a reference program. However, without preserving semantic equivalence, this often complicates the original program to repair, resulting in unnecessary repairs.

More specifically, we discuss the motivation of our approach through a real example in Fig. 1, focusing on student attempts for the assignment `Sequential Search`, which outputs *how many numbers in a sorted number sequence* `seq` *are smaller than* `x`. To repair the buggy program (as depicted in Fig. 1(a)), *Refactory* initially tried to generate a matching correct variant with an identical control-flow structure but failed. Consequently, *Refactory* employed "*structure mutation*" to modify the control-flow structure of the buggy program to that of the reference program (as depicted in Fig. 1(b)). This involved removing the first "elif" statement (line 5), updating the second "elif" statement (line 7) to "if", and changing the "else" statement (line 9) to "elif". This significantly altered the semantics of the original buggy program, causing it to fail more test suites. As a result, *Refactory* had to put in more effort to repair the buggy program, leading to the need for additional, unnecessary repairs, as evident in Fig. 1(c). This example demonstrates that altering the control structure without preserving semantics can lead to unnecessary repairs and reduced quality. In an alternative approach, if we align the control-flow structures without introducing semantic changes, we would generate a better repair with a smaller patch size, as experimentally proven in Fig. 1(d). Simultaneously, reducing unnecessary block repairs after structure alignment remains a challenging task. While *Refactory* introduces *specification inference* and *block patch synthesis*, optimization is still needed. We propose integrating a coarse-to-fine fault localization approach and statement matching to generate more concise patches.

3 THE BRAFAR TECHNIQUE

We design and build BRAFAR, a general automated feedback generation system for IPAs – *bidirectional refactoring, alignment, fault localization, and repair*. Fig. 2 presents an overview of our approach.

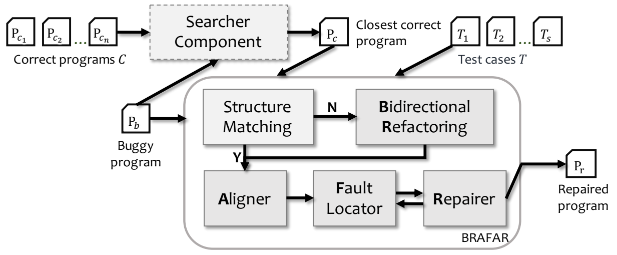


Figure 2: Overview of our approach.

It takes three inputs: a buggy program P_b, (one or more) correct programs C, and test cases T. Initially, we search for the closest correct program P_c as a repair reference. If P_b and P_c share different control-flow structures, we will use a novel *bidirectional refactoring* algorithm, described in Section 3.2, to align them. We then use three major components to generate feedback for P_b: (1) Aligner, described in Section 3.3.1, aligns the blocks and variables between P_b and P_c. (2) Fault Locator, described in Section 3.3.2, locates the suspicious basic block in P_b in a coarse-to-fine manner. And (3) Repairer, described in Section 3.3.3, takes the suspicious basic block and its corresponding basic block in P_c as input, and outputs block repairs. Notably, *fault localization* and *block repair* are repeatedly conducted until the generated program passes all test suites.

3.1 Searcher

Like existing techniques, our approach starts by identifying the closest correct program to the buggy program for repair. This is done using a coarse-to-fine method which considers control-flow structure similarities first and refines the selection based on overall syntax. First, we perform exact matching between correct programs and the buggy program *w.r.t* their control-flow structures (*Definition 3.1*). If no match is found, we rank the correct programs by control-flow sequence edit distance and select the top k (k=10). In the second step, we calculate the token-level edit distance between the buggy program and these top k programs to identify the closest one.

Definition 3.1 (Control-flow Structure). Given a program P, the *control-flow structure* (CFS) of P, is a representation derived from the abstract syntax tree (AST) that captures the program's control flow while discarding other details. It is constructed by identifying the control statements during AST traversal and creating corresponding nodes for each control statement. Table 2 shows three types of CFS nodes. One specific enhancement in the CFS construction is adding branch nodes to represent different branches of an If statement. The CFSs of P_b and P_c in Fig. 3 are shown in Fig. 4.

This two-step approach effectively narrows the search space and leverages the availability of existing correct solutions. However, when only a few correct programs are available, the similarity between the reference program and the buggy program cannot be guaranteed. In such cases, the search process becomes unnecessary, and *the feasibility of our tool highly relies on the robustness and adaptability of the bidirectional refactoring algorithm.* For example, even if the closest correct program P_c in Fig. 3(b) has a different CFS from the buggy problem P_b in Fig. 3(a), our approach remains effective. The following sections detail how P_c is used to repair P_b.

Table 2: The types and labels of *control-flow structure* nodes

Type	Description	Label
Method Entry	Nodes represent the entry of the methods under consideration. They are the root nodes of the *control-flow structures.*	"M" for "*MethodEntry*".
Control-flow Node	Nodes represent the control statements of the program.	"F" for *"ForStatement"*; "W" for *"WhileStatement"*; "If" for *"IfStatement"*.
Branch Node	Nodes represent the branches of the *IfStatement.*	"T" for "*ThenBranch*"; "E" for "*ElseBranch*".

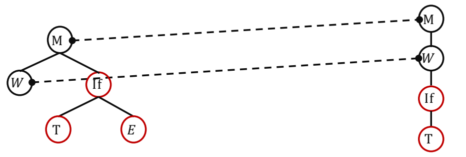
```
def search(x, seq):
  i = 0
  while i<len(seq) and x<seq[i]:
    i += 1
  if i==len(seq):
    seq += (x,)
  else:
    seq.insert(i, x)
  return seq
```

(a) A buggy program P_b.

```
def search(x, seq):
  i = 0
  while i<len(seq):
    if x <= seq[i]:
      return i
    i += 1
  return len(seq)
```

(b) A correct program P_c.

Figure 3: Real student submissions.



(a) The *control-flow structure* of P_b. (b) The *control-flow structure* of P_c.

Figure 4: The *control-flow structures* of $\mathrm{P_b}$ and $\mathrm{P_c}$ in Fig. 3 and the best legal match between them.

3.2 Bidirectional Refactoring

Our goal of *bidirectional refactoring* is to achieve *structure alignment* between the buggy program P_b and its reference program P_c without semantic changes. To maintain the overall semantics of the code, our algorithm carefully restricts the refactoring operations during the process, in contrast to *structure mutation*, which allows *additions, deletions, updates, or movements*. Specifically, we support the following semantic preserving refactoring operations:

A. Introduce new control-flow node Rules $R_{A_1} - R_{A_3}$ in Fig. 5 introduce new control-flow nodes without semantic change. Rule R_{A_1} introduces a new If statement. Rule R_{A_2} introduces a new While statement. Rule R_{A_3} introduces a new For statement.

B. Exchange branches of conditional The rule R_B in Fig. 5 states that the blocks in different branches of an If statement can be exchanged after the condition is negated.

C. Introduce new guard Rules R_{C_1} and R_{C_2} in Fig. 5 introduce new guards for block B without semantic change. R_{C_1}/R_{C_2} consists of two operations: (1) add a new if statement with the condition being *true/false*; (2) move block B to the *then/else* branch.

The refactoring rules themselves are straightforward. Our contribution extends beyond providing these rules; we focus on ***identifying the differences between CFSs to guide refactoring***. To

Category A. Introduce new control-flow node.

```
if True then              while True do        for i in range(0) do
| Empty Block B           | Break              | Empty Block B
```

(a) R_{A_1} (b) R_{A_2} (c) R_{A_3}

Category B. Exchange branches of conditional

```
if Condition C then       if not C then
| Block B1                | Block B2
else                      else
| Block B2                | Block B1
```

(a) (b)

Category C. Introduce new guard

```
                                       if False then
                  if True then         | pass
Block B           | Block B            else
                                       | Block B
```

(a) (b) R_{C_1} (c) R_{C_2}

Figure 5: List of refactoring rules.

achieve this, we propose a *bidirectional refactoring* algorithm with two steps: (1) determining the *best legal* match between the CFSs, and (2) generating a bidirectional edit script based on this match.

3.2.1 *Control-flow Matching Algorithm.* In the initial step of *bidirectional refactoring*, we introduce an innovative *control-flow matching* algorithm to establish the *best legal* match between the two CFSs. This step holds paramount importance, as the quality of the match significantly influences the overall success and safety of the entire process. *Ensuring the legality of the match is critical so that the effective application of the existing refactoring rules in Fig. 5 is sufficient for structure alignment.* Meanwhile, *the match length guarantees minimal modifications to the CFSs.*

Definition 3.2 (Control-flow Sequence). A *control-flow sequence* is a list L derived from a preorder traversal of a CFS, where each node is an element within the tree itself, preserving its hierarchical structure. The *head* L_n is defined as the first n nodes of L. For L = (M, W, If, T, E), L_0 = (), L_1 = (M), L_2 = (M, W), L_5=(M, W_1, If, T, E).

Definition 3.3 (isCompatible Function). Let `isCompatible(n_i, n_j)` represent whether control-flow structure nodes n_i and n_j are compatible. This function returns true if n_i and n_j satisfy one of the following conditions: (1) both are *Method Entry*; (2) both are *Branch Node* or (3) both are *Control-flow Node* with the same label.

Definition 3.4 (Control-flow Match). The *control-flow match* is a set of node pairs that establishes a correspondence between the control-flow sequences of two programs. Formally, for control-flow sequences $X(x_1x_2...x_m)$ and $Y(y_1y_2...y_n)$, a control-flow match $\mathcal{M}$=$\{(x_{i_1}, y_{j_1}), ..., (x_{i_s}, y_{j_s})\}$, where $i_1, ..., i_s \in 1..m$ are different indices and $j_1, ..., j_s \in 1..n$ are different indices. $\forall(x, y) \in M$ satisfies the constraint of `isCompatible` function. A match is called the "longest control-flow match" if no other match has a greater length. To find this match to minimize control-flow modifications, a feasible solution is the longest common subsequence (LCS) algorithm [8]. LCS works as follows (let the function $Longest(Q)$ return the subset of set Q, where each element has the longest length):

LCS Algorithm Let $LCS(X_i, Y_j)$ function represent the set of *longest matches* between X_i and Y_j. *The problem can be broken down into smaller, simpler subproblems, with solutions saved for reuse.* To

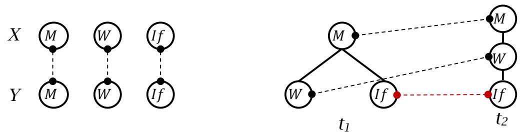


Figure 6: An example of the longest, but illegal, match.

find the LCS of X_i and Y_j, we compare x_i and y_j. If they cannot be matched, then $LCS(X_i, Y_j) = Longest\ (LCS(X_i, Y_{j-1}) \cup LCS(X_{i-1}, Y_j))$. If x_i and y_j can be matched, then each match $m \in LCS(X_{i-1}, Y_{j-1})$ is extended by the match pair (x_i, y_j), and $LCS(X_i, Y_j) = Longest\ (LCS(X_i, Y_{j-1}) \cup LCS(X_{i-1}, Y_j) \cup (LCS(X_{i-1}, Y_{j-1}) \oplus (x_i, y_j)))$. For sequences X ($M$, W_1, W_2) and Y (M, W_1), `isCompatible` (x_3, y_2) returns *true*, thus $LCS(X_3, Y_2) = Longest(LCS(X_3, Y_1) \cup LCS(X_2, Y_2) \cup (LCS(X_2, Y_1) \oplus (W_2, W_1))) = \{\{(M, M), (W_1, W_1)\}, \{(M, M), (W_2, W_1)\}\}$.

Definition 3.5 (Legal Control-flow Match). A control-flow match $\mathcal{M}$ is considered legal if it aligns with the existing refactoring rules in Fig. 5. In other words, a legal control-flow match identifies the differences in refactoring between two CFSs, providing a foundation for compliance with the predefined rules. Based on a legal match, we can identify two sets of refactoring operations $\mathcal{E}_1$ and $\mathcal{E}_2$, such that applying these operations to the CFSs $CFS(P_b)$ and $CFS(P_c)$ results in structurally equivalent $CFS(\mathcal{E}_1(P_b)) \equiv CFS(\mathcal{E}_2(P_c))$.

Considering the example in Fig. 6, the longest control-flow match $\{(t_1.M, t_2.M), (t_1.W, t_2.W), (t_1.If, t_2.If)\}$ between the two CFSs is deemed illegal. This is because the if node outside the loop of t_1 is matched with the if node inside the loop of t_2. Aligning the two CFSs based on this match would require moving $t_1.If$ to the children of $t_1.W$. However, *no existing refactoring rules in Fig. 5 permit such a movement, as it might alter the program's semantics.* To ensure the legality of the generated match in each subproblem, ***additional checks*** are essential. Thus, we provide four rules to check whether a matching pair (x_i, y_j) can be legally added to the given legal match $\mathcal{M}$, incorporating specific constraints of the CFS.

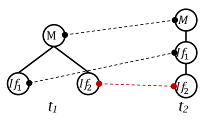
Rule1. If any CFS node $\mathbf{x_c}/\mathbf{y_c}$ in *ancestors*$(\mathbf{x_i})$/*ancestors*$(\mathbf{y_j})$ is matched in $\mathcal{M}$, the corresponding $\mathbf{y_d}/\mathbf{x_d}$ matched with $\mathbf{x_c}/\mathbf{y_c}$ in $\mathcal{M}$ should also be the ancestor of $\mathbf{y_j}/\mathbf{x_i}$. The ancestor nodes of node n are the nodes on the path from the root to node n.

As shown in Fig. 7(a), pair $(t_1.If_2, t_2.If_2)$ cannot be legally added to the $\mathcal{M}$ $(\{(t_1.M, t_2.M), (t_1.If_1, t_2.If_1)\})$, because $t_2.If_1$ is the ancestor of $t_2.If_2$ and $t_2.If_2$ is matched in $\mathcal{M}$ but the corresponding node $t_1.If_1$ is not the ancestor of $t_1.If_2$. (Nodes $t_1.If_1$ and $t_1.If_2$, which are siblings, are matched with parent-child nodes $t_2.If_1$ and $t_2.If_2$.)

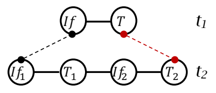
Rule2. If $\mathbf{x_i}$ and $\mathbf{y_j}$ are *Branch Nodes*, $\mathcal{M}$ should contain pair (*parent*$(\mathbf{x_i})$, *parent*$(\mathbf{y_j})$).

As shown in Fig. 7(b), pair $(t_1.T, t_2.T_2)$ cannot be legally added to $\mathcal{M}$ $(\{(t_1.If, t_2.If_1)\})$. It is because M does not contain pair $(t_1.If, t_2.If_2)$. (Node $t_1.T$ should be matched with $t_2.T_1$ since they both represent the then branch of matched if statements.)

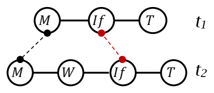
Rule3. If $\mathbf{x_i}/\mathbf{y_j}$ is a descendant of loop nodes, any *loop node* $\mathbf{x_c}/\mathbf{y_c}$ in *ancestors*$(\mathbf{x_i})$/*ancestors*$(\mathbf{y_j})$ should be matched in $\mathcal{M}$.



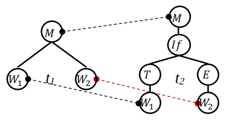
(a) Rule 1: Illegal match.



(b) Rule 2: Illegal match.



(c) Rule 3: Illegal match.



(d) Rule 4: Illegal match.

Figure 7: Illegal match between two CFSs.

As shown in Fig. 7(c), pair $(t_1.If, t_2.If)$ cannot be legally added to $\mathcal{M}$ ({($t_1.M$,$t_2.M$)}). It is illegal because $t_2.If$ is the descendant of a loop node $t_2.W$ while $t_1.If$ is not. No refactoring rule permits adding a `while` parent to $t_1.If$ to align their control-flow structures.

Rule4. If $\mathbf{x_i}/\mathbf{y_j}$ is a descendant of an unmatched *Else* node $\mathbf{n_e}$ (this scenario occurs when *parent*($\mathbf{n_e}$) – $\mathbf{n_{if}}$ is unmatched), the descendants of $\mathbf{n_{if}}$'s *then branch* should be unmatched in $\mathcal{M}$.

As shown in Fig. 7(d), pair $(t_1.W_2, t_2.W_2)$ cannot be legally added to $\mathcal{M}$ ({$(t_1.M, t_2.M)$, $(t_1.W_1, t_2.W_1)$}), because $t_2.W_2$ is a descendant of the unmatched else node $t_2.E$ but a descendant $t_2.W_1$ of $t_2.If$'s Then branch is matched in $\mathcal{M}$. (Nodes $t_1.W_1$ and $t_1.W_2$ of the same branch are matched with $t_1.W_1$ and $t_1.W_2$ of different branches.)

Definition 3.6 (Legal extension). Given a set of legal matches Q and a pair (x_i, y_j), the legal extension of Q by (x_i, y_j), denoted by $Q \odot (x_i, y_j)$, is defined as follows: For any match q in Q that can be legally extended by the pair (x_i, y_j), we extend match q with (x_i, y_j). Matches in Q that cannot be legally extended remain unchanged.

However, the introduction of additional checks during the LCS algorithm compromises the optimality of the global solution. This happens because dynamic programming relies on the principle of optimal substructure, where each subproblem's solution must be optimal and unaffected by other conditions. Therefore, we propose a novel *Extremely Long Legal Match* (ELLM) algorithm which is built upon *LCS algorithm* and maintains the optimal substructure property by changing the subproblem into finding a set of *Extremely Long Legal Control-flow Matches* between control-flow structures.

Definition 3.7 (Extremely Long Legal Match). A legal control-flow match $\mathcal{M}$ ({$(x_{i_1}, y_{j_1}), ..., (x_{i_s}, y_{j_s})$}) between two CFSs is considered extremely long only when it is not a subset of any other legal control-flow matches between the two CFSs.

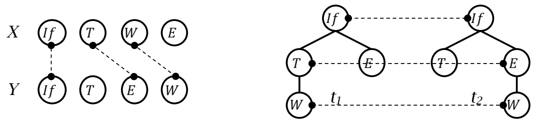


Figure 8: An example of a legal match.

ELLM Algorithm. Let *ELLM*(X_i, Y_j) represent the set of extremely long legal matches of X_i and Y_j. The optimal value function is given by the following (Let $E_b = ELLM(X_{i-1}, Y_j)$, $E_c = ELLM(X_i, Y_{j-1})$, $E_d = ELLM(X_{i-k}, Y_{j-k})$ ($k = min(\{k \in (1, min(i,j)) | \exists \mathcal{M} \in ELLM(X_{i-k}, Y_{j-k})$, $\mathcal{M}$ can be legally extended by the pair (x_i, y_j).}), and function $ExtremelyLong(Q)$ represent a subset of set Q, where $\forall q \in ExtremelyLong(Q)$ is not a subset of any other matches in Q.):

- If $i = 0$ or $j = 0$, $ELLM(X_i, Y_j) = \{\emptyset\}$.
- If $i, j > 0$ and $isCompatible(x_i, y_j)$ returns false, $ELLM(X_i, Y_j) = ExtremelyLong(E_b \cup E_c)$.
- If $i, j > 0$ and $isCompatible(x_i, y_j)$ returns true, $ELLM(X_i, Y_j) = ExtremelyLong(E_b \cup E_c \cup (E_d \odot (x_i, y_j)))$.

Definition 3.8 (FindBestMatch Function). The function *findBestMatch* selects the best legal match among all extremely long legal matches between the two sequences. The best match is determined in two steps: (1) Ranking the matches based on the number of loop node mappings and selecting those with the maximum value, and (2) Considering the total number of node mappings in the selected matches. In our approach, loop node mappings are prioritized over If node mappings, ensuring that CFSs related to repetitive and iterative behavior are aligned as accurately as possible.

Control-flow Matching Algorithm. The algorithm is illustrated with a real example. Considering the *CFSs* of P_b and P_c in Fig. 4, the best match between them is shown in the same figure. It is generated through the following steps: (1) Firstly, the two *CFSs* are translated into *control-flow sequences* $X(M, W, If, T, E)$ and $Y(M, W, If, T)$. (2) Then, the *ELLM* algorithm is run on the two sequences, identifying a set of extremely long legal matches: $\{\{(M, M), (W, W)\}\}$, and (3) Finally the *FindBestMatch* function is employed to determine the best legal match, resulting in $\{(M, M), (W, W)\}$.

3.2.2 *Overall Algorithm.*

Algorithm 1 shows the overall bidirectional refactoring process (where the function *add*(*a*, *b*, *i*) or *move*(*a*, *b*, *i*) means introducing or moving node *a* to index *i* of the children list of node *b*): (1) First, we adopt the *control-flow matching* algorithm on the two CFSs t_1 and t_2 to get the original best legal match $\mathcal{M}$. Notably, if this match contains pairs where the "then" branch of an "If" statement in one CFS is matched with the "else" branch of another CFS (as shown in Fig. 8), we will use refactoring rule R_B to exchange the branches on certain "If" nodes in t_2. The above process will be repeatedly conducted until no requirement for branch changes (lines 1–7). (2) Next, we traverse the *control-flow sequence* X, generated by preorder traversal of CFS t_1. For each unmatched node x in $\mathcal{M}$, we will introduce a new control-flow node y_n into CFS t_2 using refactoring rule R_A and add the new pair (x, y_n) to $\mathcal{M}$ (lines 13–16). Especially, if a new branch node y_n is introduced, we will relocate its children accordingly, following the refactoring rule R_C (lines 17–21). (3) Similarly, we then traverse the *control-flow*

Algorithm 1: Bidirectional Refactoring Algorithm

```
   Input: Control-flow structures $t_1$ and $t_2$.
   Output: Refactored control-flow structures $t_1$ and $t_2$
1  do
2      $\mathcal{M} \leftarrow$ controlFlowMatch($t_1$, $t_2$), isBranchHasChanged $\leftarrow$ false;
3      foreach pair($x$, $y$) $\in \mathcal{M}$ do
4          if type($x$) is Branch-Node and label($x$) $\neq$ label($y$) then
5              exchangeBranch(parent($y$)) ;                    // Rule $R_B$
6              isBranchHasChanged $\leftarrow$ true;
7  while isBranchHasChanged is false;
8  $t_2$ $\leftarrow$Refactor ($t_1$, $t_2$, $\mathcal{M}$), $t_1$ $\leftarrow$Refactor ($t_2$, $t_1$, $\mathcal{M}$);
9  return Refactored control-flow structures $t_1$, $t_2$;
10 procedure Refactor($\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{M}$)
11     $L \leftarrow$ preorderTraversal($\mathcal{T}_1$);   ▷ Get control-flow sequence of $\mathcal{T}_1$;
12     foreach $x \in L$ do
13         if $x$ is matched in $\mathcal{M}$ then continue;
14         $y_p \leftarrow$ getMatchFromBy($\mathcal{M}$, parent($x$));
15         $y_n \leftarrow$ newNode(type($x$)), $\mathcal{M}$.add(pair($x$, $y_n$));
16         add($y_n$, $y_p$, positionInfer($x$))         ▷ Based on the index and
             context of $x$, infer where $y_n$ should be inserted ;   // Rule $R_A$
17         if type($x$) is Branch-Node then
18             $\mathcal{N} \leftarrow$ getMatchFromBy($\mathcal{M}$, children($x$));
19             move($\mathcal{N}$, $y_n$, 0) ;                          // Rule $R_C$
20         if label($y_n$) is ELSE and $\mathcal{N}$ is not None then
21             conditionalNegation($y_p$) ;                       // Rule $R_{C_2}$
22     return $\mathcal{T}_2$;
```

```python
def search(x, seq):
  i = 0
  while i<len(seq)
      and x<seq[i]:
    if True:
      pass
    i += 1
  if i==len(seq):
    seq += (x,)
  else:
    seq.insert(i, x)
  return seq
```

(a) P_{r_b}

```python
def search(x, seq):
  i = 0
  while i<len(seq):
    if x <= seq[i]:
      return i
    i += 1
  if True:
    pass
  else:
    pass
  return len(seq)
```

(b) P_{r_c}

```python
def search(x, seq):
  i = 0
  while i<len(seq):
    if x <= seq[i]:
      return i
    i += 1
  if i==len(seq):
    pass
  else:
    seq.insert(i, x)
  return len(seq)
```

(c) P_r

Figure 9: The refactored and repaired programs.

sequence Y, generated by preorder traversal of t_2, and repeat the above process to refactor CFS t_1 (line 8). The refactoring guidance for aligning the *CFSs* in Fig. 4 is as follows: (i) add(newIf, t_2.*M*, 1), (ii) add(newT, t_2.newIf, 0), (iii) add(newE, t_2.newIf, 1), (iv) add(newIf, t_1.W, 0) and (v) add(newT, t_1.newIf, 0). More intuitively, the refactored programs of P_b and P_c in Fig. 3 are shown in Fig. 9(a) and Fig. 9(b), which are generated by: introducing a new If conditional to P_b to get the refactored buggy program P_{r_b} and introducing a new If-else conditional to P_c to get the refactored correct program P_{r_c}.

3.2.3 Discussion. The time cost of *bidirectional refactoring* is primarily determined by the *control-flow matching* process. For two sequences of lengths n and m, compared to the dynamic programming approach (LCS) whose time complexity is $O(n \times m)$, the time cost of our ELLM algorithm grows exponentially in the worst-case scenario due to the additional checks. However, since real-world programs, especially student programs, typically do not have many CFS nodes per method, our algorithm is practical in most cases.

3.3 Program Repair

In this section, our goal is to utilize the reference correct program P_c to repair the buggy program P_b, given that they share identical control-flow structures. This will be achieved through three primary components: an aligner, a fault locator, and a repairer.

3.3.1 Aligner. Aligning a reference solution with the buggy program is crucial in generating accurate fixes, and in turn feedback, [37]. This process includes *block alignment* and *variable alignment*.

Definition 3.9 (Program Partitioning). The program P is divided into blocks from coarse to fine granularity. Initially, the function body of P is divided into *composite blocks* and *basic blocks* with each composite block bordered by basic blocks. Composite blocks are then systematically divided into smaller units. For example, the refactored buggy program P_{r_b} in Fig. 9(a) is divided into {basic block, `while` block, basic block(empty), `if` block, basic block}. Then, the `while` block and the `if` block are further subdivided into smaller units: the `while` block into {`while condition` (basic block), `while body`}, with continuous division occurring within the `while body`. The basic blocks of a program form a set $\{B_i\}_{i \in 1...n}$.

A. Block Alignment. Given a buggy program P_b and a reference solution P_c with the same CFS, we can find a 1-1 mapping of the blocks due to *CFS*(P_b)≡*CFS*(P_c). The criteria for dividing a program into blocks is according to its CFS (*Definition 3.9*). Programs with the same CFS share the same program partition criteria.

B. Variable Alignment. The goal of this stage is to map the variables of P_b to those of P_c. We apply *Enhanced Define/Use Analysis*(*EDUA*) to match variables with similar def/usage. This method is more reliable than the traditional *Define/Use Analysis* (*DUA*) proposed in [16], as it incorporates a new similarity calculation method.

Definition 3.10 (Variable Mapping). Let $Vars(P_b)$=$\{v_1, v_2, ..., v_m\}$, with a length of m, represent the variable set of the given buggy program P_b. Similarly, let $Vars(P_c)$=$\{u_1, u_2, ..., u_n\}$, with a length of n, be the variable set of the correct program P_c. Then, $\{v_{i_1} \mapsto u_{j_1}, ..., v_{i_s} \mapsto u_{j_s}\}$ is a mapping of variables, where $i_1, ..., i_s \in 1..m$ are different indices and $j_1, ..., j_s \in 1..n$ are different indices.

Definition 3.11 (Def/Usage Weight). Let function $\mathcal{W}(v_p, i)$ represent the def/usage weight of variable v in the i-th basic block in program P. If variable v is defined or used in the i-th basic block, and if the i-th basic block is an if/while condition or a for iterator, $\mathcal{W}(v, i) = \alpha$; if not, $\mathcal{W}(v, i) = 1$. (We consider the control statements have a higher weight and set α to 2 in our work.) Otherwise, if variable v is not defined or used in the i-th basic block, $\mathcal{W}(v, i) = 0$.

Definition 3.12 (Def/Usage Similarity). We compute the similarity for each pair of variables <v, u>, where v is defined in P_b and u is defined in P_c. Let the function $f_i(v, u)$ represent the extent to which v and u are defined/used similarly in the i-th basic block. If v and u are both defined or used in i-th basic block, and if the i-th basic block is an if/while condition or a for iterator, $f_i(v, u) = \alpha$; if not, $f_i(v, u) = 1$. Otherwise, if v and u are not both defined or used in the i-th basic block, $f_i(v, u) = 0$. The similarity between v and u is computed as follows (supposing there are n basic blocks):

$$Sim(v, u) = \frac{\sum_{i=1}^{n} f_i(v, u)}{max(\sum_{i=1}^{n} \mathcal{W}(v, i), \sum_{i=1}^{n} \mathcal{W}(u, i))} \tag{1}$$

Definition 3.13 (Enhanced Define/Use Analysis (EDUA)). $\mathcal{M}_{EDUA}$ stores a set of variable pairs matched using *EDUA*. *EDUA* follows a conservative approach, matching variables in multiple rounds to minimize erroneous matches. In the first round, parameter variables are matched in order. In the second round, variables v_b and v_c with similarity $Sim(v_b, V_c) = 1.0$ are matched. Subsequent rounds match variables with progressively relaxed similarity thresholds: ≥0.9, ≥0.8, ≥0.7, ≥0.6, or ≥0.5. For the programs in Fig. 9(a) and 9(b), the variable mapping would be: **`{x ↦ x, seq ↦ seq, i ↦ i}`**.

3.3.2 Fault Locator. To reduce unnecessary block repairs, we introduce a fault localization technique for identifying suspicious blocks. The core concept behind this approach involves inferring a specification for each block within the buggy program and conducting fault localization in a coarse-to-fine manner.

Definition 3.14 (Specification). Let B be a block in a program P, V be the set of variables defined or used in B and D be the set of possible values of V. A program state $\mathbb{S}$ during the execution of B is a mapping $\mathbb{S} : V \longrightarrow D$. The specification of a block B, denoted by $Spec(B)$, is defined as a set of input-output state pairs $\{\langle \mathbb{S}_{I_i}, \mathbb{S}_{O_i} \rangle\}_i \in 1, ..., r$, where i denotes the i-th variable v_i in V and $\mathbb{S}_{O_i}$ denotes the output value of v_i when given $\mathbb{S}_{I_i}$ as the input state of B. Specifically, if B is an if/while condition, the specification $Spec(B)$ will include the output state of this expression.

Definition 3.15 (Specification Inference). Let B_c be a block in the correct program P_c, and $\{\langle \mathbb{S}^c_{I_i}, \mathbb{S}^c_{O_i} \rangle\}_i \in 1, ..., r$ be a specification of B_c, and B_b be the corresponding block in the buggy program P_b, and M be a valid variable mapping. Each input-output state pair $\langle \mathbb{S}^b_{I_j}, \mathbb{S}^b_{O_j} \rangle$ of B_b should satisfy the following condition: if there exists a variable v^c_i in P_c which is mapped to v^b_j in M and the input state $\mathbb{S}^b_{I_j}$ equals $\mathbb{S}^c_{I_i}$, the output state $\mathbb{S}^b_{O_j}$ should be the same as $\mathbb{S}^c_{O_i}$.

A. Specification Construction. Run the refactored buggy program P_b on the failed test-suite T to collect *Specification* for each block of P_b. Run the refactored correct program P_c on the same test-suite T to collect *Specification* for each block of P_c. Here, the input state refers to the values of all variables before executing the block, and the output state refers to these values after executing the block.

B. Fault Localization. Similar to program partitioning, we employ a coarse-to-fine strategy to locate the suspicious basic block. This process is initiated by scanning the *composite blocks* and *basic blocks* within the function body of the buggy program. If the specifications of the current block fail to meet expectations, we consider it suspicious. If the located block is not a basic block, we continue refining our search until a suspicious basic block is identified. For example, for the test case `search(42, (-5, 1, 3, 5, 7, 10))`, we first identify the `while` block in Fig. 9(a) as suspicious, because the output of variable `i` in P_b is 0 while it is expected to be 6. Subsequently, we identify the `while condition` as suspicious, as it yields a `false` output when it is expected to be `true`.

3.3.3 Repairer. We focus on repairing only the suspicious basic block to guarantee minimum repair. The key idea is to generate block patches for B_b until B_b is semantically equivalent to its corresponding basic block B_c in the correct program P_c. This process includes two steps: (1) statement match and (2) patch generation.

A. Statement Match. We match the statements between B_b and B_c in rounds [35], each with progressively less strict match conditions. In the first round, we match statements that become identical after replacing variable names. In the second round, we match statements that become identical after swapping variable orders. In the third round, we match statements that contain matched variables. Statements matched in earlier rounds are "safer" matches and are excluded from subsequent rounds. Finally, the unmatched statements in B_b and B_c would be matched with empty statements.

B. Patch Generation. Given the statement match $\mathcal{M}(B_b, B_c)$, we generate block patches as follows: Given a pair (S_b, S_c),

- **Insertion Fix**: if S_b is $\emptyset$, meaning S_c is not matched to any statement in B_b, an insertion fix will be produced to insert S_c.
- **Deletion Fix**: if S_c is $\emptyset$, meaning S_b is not matched to any statement in B_c, a deletion fix will be produced to delete S_b.
- **Modification Fix**: if S_b and S_c are not $\emptyset$, a modification fix will be produced to turn the AST of S_b to that of S_c.

After performing insertion or modification operations, we will replace variable names in S_c with those in P_b based on the established variable mapping. If a variable v_c in $Vars(S_c)$ is not matched to any variable in P_b, we will insert a new variable initialization into the corresponding declaration block and add a new variable mapping pair. Note that statements of type *Variable Declaration* will not be deleted during this process. Unnecessary definitions will be removed after the entire program has been repaired.

3.3.4 Overall Repair Process. First, we align the blocks and variables between the buggy program P_b and its reference program P_c. Next, we systematically identify the suspicious basic blocks of P_b in a coarse-to-fine manner. Upon locating a suspicious basic block, the fault localization process concludes, and the identified block undergoes repair. It is crucial to note that block repair targets only a single block but does not guarantee the complete program repair. Therefore, we *repeatedly conduct* the *fault localization* and *block repair* steps until the repair program passes all the test suites. We generated the repaired program P_r in Fig. 9(c) for the program P_b.

3.4 Implementation

The Brafar technique has been developed into a tool with the same name. Given one or more reference programs and test cases, Brafar supports repairing compilable incorrect Python programs. The implementation of Brafar tool is composed of five components: (1) Searcher, (2) Bidirectional refactoring algorithm, (3) Aligner, (4) Fault localization algorithm, and (5) Repair algorithm.

4 EXPERIMENTAL EVALUATION

In this section, we conduct a series of experiments to evaluate the effectiveness and efficiency of Brafar. The evaluation aims to address the following research questions:

- **RQ1** How does the overall approach perform in terms of repairing incorrect programs, considering its overall effectiveness, efficiency, and repair quality?
- **RQ2** How does our *bidirectional refactoring* algorithm perform in terms of aligning control-flow structures?
- **RQ3** How does our *repair strategy* perform in fixing the incorrect programs with a matching correct reference program?

Besides, we have conducted a preliminary evaluation of ChatGPT [1] to discuss the significance of our work in the current LLM context.

Table 3: Overall results on five IPAs: The repair rate (RR), average repair time (Time), and relative patch size (RPS). The column "%Match" gives the percentage of incorrect programs for which correct programs with matching CFSs are found.

ID	Description	LOC	#Correct	#Incorrect	#Test	%Match	BRAFAR			Refactory			CLARA		
							RR	Time	RPS	RR	Time	RPS	RR	Time	RPS
1	Sequential search	26	768	575	11	80.0%	99.8%	0.38s	0.28	99.3%	1.43s	0.40	81.2%	0.36s	0.31
2	Unique dates/months	24	291	435	17	33.3%	94.0%	0.43s	0.31	78.4%	2.49s	0.47	41.8%	1.67s	0.33
3	Duplicate elimination	37	546	308	6	87.3%	100.0%	0.25s	0.29	97.4%	3.35s	0.32	88.6%	0.16s	0.44
4	Sorting tuples	34	419	357	6	52.9%	100.0%	0.33s	0.24	82.6%	5.95s	0.27	62.5%	0.31s	0.60
5	Top-k elements	18	418	108	5	80.6%	100.0%	0.28s	0.22	85.1%	10.58s	0.26	88.0%	0.22s	0.48
Overall		27	2442	1783	45	64.5%	98.5%	0.35s	0.28	89.7%	3.38s	0.37	69.5%	0.49s	0.41

4.1 Experiment Setup

Benchmark Setup. We evaluate BRAFAR on a large dataset from *Refactory* [16]. This dataset consists of 2442 correct submissions and 1783 incorrect student programs from 5 different introductory programming assignments, along with reference programs and instructor-designed test suites. *We consider repairs correct only if they pass all the test suites.* Table 3 gives the fundamental information about the dataset, with more details available in *Refactory*.

Baseline Feedback Generation Techniques. We selected *Refactory*[2] [16] and CLARA[3] [13] as baselines for their relevance as publicly accessible Python feedback generators. While previous research [16] evaluated them on the same dataset, we have re-executed *Refactory* and CLARA under identical conditions for fair comparison. Note that during the reproduction of CLARA, we made some necessary code modifications to adapt to the dataset. Comparisons with other recent approaches like *AssignmentMender* [23] and FAPR [25] were not feasible due to their lack of public availability. Additionally, *Verifix* [4], designed for C programs, was excluded from our comparison since BRAFAR is specifically tailored for Python programs.

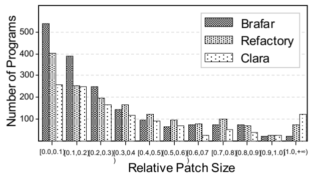
Environment. All the experiments were performed on a desktop equipped with an Intel® Core^{TM} i7-10700 CPU, 32GB RAM, running Ubuntu 22.04. During the experiment, all the tools were set to operate in single-thread mode, with a one-minute timeout for repairing each incorrect program. Specifically, CLARA requires an offline phase where it clusters the correct programs, for which we allocate an additional five-minute timeout per assignment.

4.2 RQ1: Overall Performance

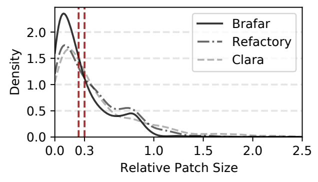
Table 3 presents the overall performance comparison among BRAFAR, *Refactory*, and CLARA. CLARA successfully generates correct repairs for 69.5% of 1783 incorrect programs, with an average time of 0.49 seconds. *Refactory* achieves correct repairs for 89.7% of incorrect submissions, averaging 3.38 seconds per repair. In contrast, BRAFAR can generate correct repairs for 1756 programs in total (≈98.5%) within an average time of 0.35 seconds. This indicates that BRAFAR outperforms *Refactory* and CLARA, as it can repair a greater number of incorrect submissions while requiring less time.

4.2.1 Repair Rate. The low repair rate of CLARA is mainly due to its inability to repair incorrect programs with unique looping structures, accounting for about 67.0% (364 out of 543) of the failures.

[2]https://github.com/githubhuyang/refactory
[3]https://github.com/iradicek/clara



(a) Histogram of relative patch size.



(b) KDE plot of relative patch size.

Figure 10: Relative Patch Size Comparison.

For the remaining failures in CLARA, 143 are due to unsupported Python features, 8 are attributed to numeric precision errors in the ILP solver and 28 result from misjudging that a repair is unnecessary. In comparison, BRAFAR and *Refactory* achieve higher repair rates due to their structure alignment phases. Between them, BRAFAR failed to generate a repair in only 27 cases. The only reason for BRAFAR's failure is *the invocation of an unknown method in the reference programs.* Using these correct solutions to repair the incorrect programs risks introducing an unknown method invocation into the incorrect program, rendering it non-compilable. *Refactory* also encounters this failure reason, but it fails to generate repairs in more cases. This is because *Refactory* struggles to generate repairs without valid variable mappings, and nearly half of its failures occur due to exceptions or timeouts during the repair process.

4.2.2 Repair Size. For further evaluating generated patches, we use the Zhang-Shasha [15] tree-edit-distance algorithm to calculate the patch size and normalize the patch size by dividing it by the size of the original buggy program, using the Relative Patch Size (*RPS*) metric. The formula for *RPS* is $RPS = Dist(AST_r, AST_b)/Size(AST_b)$ which is widely used in existing techniques. To ensure a fair comparison, we recalculated CLARA's patch size (repair cost) to align with BRAFAR and *Refactory*. As shown in Table 3, repairs generated by BRAFAR have a smaller average *RPS* compared to those generated by *Refactory* and CLARA. *This indicates that our repairs are smaller and, therefore, more likely to be easier for students to comprehend.* Fig. 10 shows a more detailed *RPS* comparison. Fig. 10(a) compares the distribution of *RPS* and Fig. 10(b) plots the *Kernel Density Estimate* (KDE) of *RPS* for BRAFAR, *Refactory* and CLARA. From Fig. 10(a) and 10(b), we observe that (1) 67% of all repairs generated by BRAFAR have $RPS < 0.3$, 53% have $RPS < 0.2$ and 31% have $RPS < 0.1$; and (2) BRAFAR produces a higher number or density of *RPS* values compared to when *RPS* is less than *Refactory* and CLARA when *RPS* is less than 0.3 and 0.25 (x-axis), respectively. That is, *a larger proportion of repairs generated by Brafar have a small RPS.*

4.2.3 Repair Quality. To evaluate the quality of repairs generated by Brafar and compare them with those produced by *Refactory* and Clara, we randomly and proportionally selected 100 buggy programs for which both Brafar and *Refactory* were able to generate repairs. Note that we excluded Clara from consideration due to its relatively poor repair performance. We then manually inspected the repairs produced by these tools. In total, we inspected 148 buggy methods, as each buggy submission for the second assignment required an average of 3 methods to be repaired.

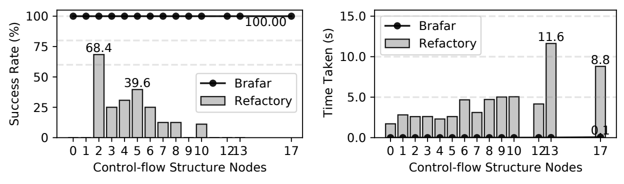
Among the 148 methods, Brafar generated: (a) *minimal* repairs that involve only necessary and natural modifications that programmers would make in 109 cases, (b) *small* repairs that involve some unnecessary modifications in 22 cases, (c) *distant* repairs that are correct but deviate somewhat or entirely from the students' ideas in 7 cases, and (d) *free* repairs that fix incorrect methods which have almost empty bodies in 9 cases. Given that most, or 88.5% to be precise, of the generated repairs are minimal and small, not causing the repaired programs to deviate from the students' ideas and therefore being easier to comprehend, the quality of the generated repairs is generally high. Meanwhile, the significant numbers of *distant* repairs suggest future directions to improve the current repair quality. On the one hand, to reduce the number of large repairs, we should utilize both the semantic and the syntactic information about the buggy methods when generating repairs. On the other hand, to better align the free repairs with the implementations of the other existing methods, the repair generation process needs to incorporate semantic and syntactic information across methods.

Brafar compared favorably with *Refactory* in terms of the quality of their generated repairs. Among the 148 buggy methods, the two tools generated the same repairs in 90 cases; In 19 cases, they generated different repairs, but the repairs were of comparable quality, being *minimal*, *small*, *distant*, or *free*. In one case, repair from *Refactory* was considered better; In the remaining 38 cases, repairs from Brafar were considered better. Within these 38 cases, Brafar generated *minimal* and *small* repairs in 32 and 6 cases, respectively, while *Refactory* generated *distant* repairs in 19 cases. Besides, the quality difference between the repairs generated by the two tools was more striking on buggy methods that demand more complex repairs. In particular, 35 of the buggy methods went through the *structure alignment* process, where Brafar produced *minimal* and *small* repairs in 16 and 13 cases, respectively, while *Refactory* produced *distant* repairs in 26 cases.

In comparison, Clara failed to generate repairs in 53 cases, with 49 of these failures due to structure mismatch issues. For the remaining repaired cases, Brafar and Clara produced repairs of comparable quality in 65 cases. Brafar produced better repairs in 28 cases, while Clara produced better repairs in only 2 cases.

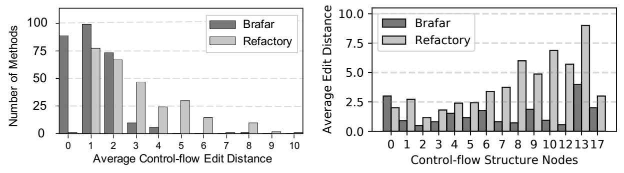
Overall, our manual analysis confirms that Brafar can produce high-quality repairs for buggy programming assignments and that compared to the repairs generated by *Refactory* and Clara, the repairs from Brafar often have comparable or better quality.

4.2.4 Sampling Rate. As *Refactory* [16] reported, with a sampling rate of 0%, i.e., using only the master solution as the input correct program, *Refactory* achieved over 90% repair rate while Clara



(a) Average success rate per method. (b) Average time taken per method.

Figure 11: Bidirectional Refactoring vs. Refactoring.



(a) Distribution of average control-flow edit distances in incorrect methods. (b) Average control-flow edit distance with the increase of control-flow nodes.

Figure 12: Bidirectional Refactoring vs. Structure Mutation.

archived less than 40% repair accuracy. We evaluated the effectiveness of Brafar under the same condition, and the tool managed to successfully repair 1772, i.e., 99.4%, of the 1783 cases with a sampling rate of 0%. Meanwhile, the different sampling rates did not significantly affect the average time taken by Brafar to repair a buggy student program. Therefore, we conclude that the effectiveness of both Brafar and *Refactory* in repairing buggy programming assignments is *insensitive* to the number of input correct submissions. In contrast, Clara relies on diverse correct programs. Moreover, Brafar is more effective than *Refactory* when the input correct programs are reduced to the minimum.

4.3 RQ2: Bidirectional Refactoring Evaluation

The higher repair rates achieved by Brafar and *Refactory* are due to their *structure alignment* algorithms. As shown in Table 3, only 64.5% of the incorrect programs have a matching correct program with identical CFS. To repair the remaining 633 programs, both Brafar and *Refactory* initially perform *structure alignment*. However, 233 programs were repaired by both tools without further alignment by separately finding matching correct methods and combining them as references. Excluding these 233 incorrect programs and 61 incorrect programs where *Refactory* encountered exceptions, this section focuses on evaluating the *structure alignment* abilities of Brafar and *Refactory* in terms of their effectiveness, efficiency, and CFS edit distance for the remaining 339 incorrect programs, which include a total of 387 methods requiring *structure alignment*.

To help these incorrect programs find correct programs with matching structures, *Refactory* attempts to use a set of refactoring rules to generate new correct variants. However, *Refactory* faces effectiveness and efficiency concerns during this process, as mentioned in the previous section (§2). For the effectiveness concern, *Refactory* only helps 94 of 339 incorrect programs (27.7%) find a new matching correct program. For the efficiency concern, the average time taken (the deviation values are removed) during the online refactoring process is around 3.0s per method. To further analyze the performance, we refer to Fig. 11: (1) As the complexity of the method's control structure increases, *Refactory*'s approach of simply refactoring correct programs without any guidance proves to be insufficient and unsuccessful for structure alignment. (2) The average time taken by *Refactory*'s refactoring process increases exponentially. In contrast, after applying the *bidirectional refactoring* algorithm, Brafar achieves successful structure alignment for all the incorrect programs (100%) in a short amount of time ($\approx$ 0.0s).

The remaining 245 incorrect programs (containing 277 incorrect methods) that do not have a CFG match with correct programs undergo *structure mutation* during *Refactory*'s structure alignment phase. We focused on the CFS modifications made by both *Refactory* and Brafar for these programs and gave the comparison in Fig. 12: (1) Fig. 12(a) illustrates that *most of the CFS modification made by* Brafar *does not exceed 2 operations and the number falls faster than Refactory*. (2) Fig. 12(b) shows that the CFS modification made by *Refactory* tends to be large as the CFS nodes increase. In contrast, Brafar exhibits a more consistent behavior. *We conclude that, in most cases, our bidirectonal refactoring algorithm achieves effective structure alignment with a small CFS modification size.*

Moreover, as discussed in Section 2, *Refactory*'s *structure mutation*, which cannot preserve the program's semantics, might ultimately require more effort to repair the incorrect programs. We simply compared the number of test cases passed by the incorrect programs before and after *structure mutation* and found that 63.7% (156/245) of the mutated programs generated by *Refactory* failed more test cases. Focusing on the repair results for these incorrect programs that undergo *structure mutation*, *Refactory* successfully repairs 93.9% (230/245) of them, with an average RPS reaching up to 0.68. In contrast, Brafar achieves a higher repair rate of 99.2% and generates repairs with a smaller average RPS of 0.49 for these incorrect programs. The smaller RPS result is made possible by our *bidirectional refactoring* algorithm and our integrated repair strategy. To separately evaluate how the *bidirectional refactoring* algorithm helps reduce RPS, we created *Refactory+* by replacing *structure mutation*'s results with this algorithm's results. The mixed tool's average RPS for these incorrect programs is 0.55. *This demonstrates that* Brafar *can achieve structure alignment while preserving semantic consistency, which ultimately helps generate smaller repairs.*

4.4 RQ3: Repair Strategy Evaluation

The smaller average relative patch size (RPS) by Brafar is mainly attributable to its integrated repair strategy, which incorporates a coarse-to-fine fault localization approach. To compare our repair strategy with that of *Refactory*, we focus on these incorrect programs (totaling 1150) that have a matching correct submission with an identical CFS, thereby obviating the need for the *structure alignment* process. In summary, *Refactory* generated 1119 repairs with an average RPS of 0.26, while Brafar generated 1150 repairs with a smaller average RPS of 0.22. Specifically, for assignment 1, the average RPS is 0.25 in Brafar and 0.31 in *Refactory*. For assignment 2, the average RPS is 0.15 in Brafar and 0.20 in *Refactory*. For assignment 3, the average RPS is 0.25 in Brafar and 0.27 in *Refactory*. For assignment 4, the average RPS is 0.16 in Brafar and 0.20 in *Refactory*. For assignment 5, the average RPS is 0.16 in Brafar and 0.18 in *Refactory*. *This demonstrates that our integrated repair strategy can help generate smaller repairs than Refactory.*

Table 4: Repairing results of ChatGPT 4.0 in repair rate (RR) and relative patch size (RPS) for the three types of queries.

ID	Type I		Type II		Type III		Brafar	
	RR_1	RPS_1	RR_2	RPS_2	RR_3	RPS_3	RR	RPS
1	57.9%	0.39	89.0%	0.51	91.7%	0.56	99.8%	0.28
2	65.7%	0.51	79.8%	0.59	94.7%	0.71	94.0%	0.31
3	62.3%	0.33	78.6%	0.40	94.5%	0.55	100.0%	0.29
4	37.8%	0.22	88.8%	0.40	65.8%	0.96	100.0%	0.24
5	84.3%	0.39	89.8%	0.50	89.8%	0.50	100.0%	0.22
Overall	58.2%	0.39	85.0%	0.49	87.6%	0.66	98.5%	0.28

4.5 ChatGPT Evaluation

Recent advancements in large language models (LLMs) have led to their widespread use in programming tasks. To gain insight into how Brafar stacks up against LLMs in repairing IPAs, we conducted a preliminary experiment where OpenAI ChatGPT 4.0 was applied to repair the buggy Python programs sourced from the same dataset as described earlier in this section. In the experiment, we configured ChatGPT 4.0 to utilize the default values for all its hyperparameters, e.g., temperature, max_length, and top_p.

In the experiment, we designed three types of queries to assess ChatGPT's effectiveness under different conditions: one without any additional information, one with test cases, and one with a reference program. To standardize the procedure, queries of Type I are in the form "Repair the following incorrect code P_b with minimal modifications.", where P_b is the buggy student code; queries of Type II are in the form "Repair the following incorrect code P_b with minimal modifications along with the test cases T", where T is a set of test cases; and queries of type III are in the form "Repair the following incorrect code P_b with minimal modifications along with the reference correct code P_c", where P_c is the reference program provided by the instructors. The repair results of ChatGPT for the five IPAs using each query type are shown in Table 4, alongside the results achieved by Brafar for easy reference.

We acknowledge that our experiment with ChatGPT, compared with those done in related work [42], was rudimentary and insufficient for quantitative analysis, e.g., we only used the default values for ChatGPT's hyperparameters, and we did not really interact with ChatGPT so it has a chance to refine its answers. Nevertheless, based on the results, we make the following two observations about ChatGPT's effectiveness in repairing IPAs: (1) First, ChatGPT faced challenges in fixing the programming assignments when no additional information was available but showed a significant improvement in repair rates when provided with test cases and reference implementations. This suggests that even higher repair rates might be achievable with more comprehensive input. (2) Second, properly encoding additional information or requirements into the prompts is crucial for ChatGPT to understand and utilize them effectively. Manual inspection revealed that in 47.2% (737/1562) of the successfully repaired cases with the third type of queries, ChatGPT simply returned the exact reference programs as the repair results.

These results are considered less valuable, indicating that ChatGPT has limited capability to comprehend and utilize contextual information effectively. Meanwhile, while ChatGPT was instructed to generate repairs "with minimal modifications", the repairs it produced were significantly larger than those returned by Brafar, showing that ChatGPT's capability to comprehend and fulfill the requirements without extra help is also limited.

We believe that our work remains meaningful despite the rapid development of LLMs and their successful applications in programming tasks. The reason is that, given a buggy student program, a group of tests, and a non-empty set of correct implementations as the reference as the input, there are solutions to derive high-quality repairs to the buggy program step by step, and Brafar is such a systematic and effective solution. In contrast, while LLMs are extremely powerful, they are probabilistic in nature, and there is a long way to go before they can fully understand and utilize the input data and constantly produce high-quality repairs accordingly.

4.6 Threats to Validity

In this section, we outline possible threats to the validity of our experimental findings.

Construct Validity. Threats to construct validity of our findings in the experimental evaluation concern whether the measurements accurately capture the studied phenomena. We classify a repair as *correct* if it passes all input tests. While this approach may mistakenly accept incorrect repairs if the input tests miss some aspects of the program specification, it is reasonable in the context of programming assignment grading since it mimics how instructors determine program correctness in practice, and it is consistent with how other researchers in this area assess correctness [13, 16].

Internal Validity. Threats to internal validity concern whether the experiments controlled for possible confounders. One obvious threat comes from possible mistakes in our implementation of Brafar. To mitigate this threat, we reviewed Brafar's implementation and the experimental scripts to ensure their correctness before conducting the final experiments.

External Validity. Threats to external validity concern whether the experimental findings generalize to other contexts. We evaluated Brafar on a dataset of buggy programming assignments from a related work [16], which contains a large number of correct and incorrect implementations for five different introductory programming assignments and constitutes a good representative for introductory programming assignments in practice.

5 RELATED WORK

The problem of automated feedback generation has seen much recent interest over the last years [4–6, 9, 10, 12–14, 16–18, 22, 23, 28–33, 36, 37, 41, 43]. It appears to be related to automated program repair (APR). However, compared with professional software, student programs have different characteristics [41], leading to the poor performance of traditional APR tools [7, 19–21, 24, 27, 38–40].

Our work builds upon prior feedback generation techniques which apply the *"block-by-block"* repair strategy. Among them, Clara [13] clusters the correct submissions by dynamic analysis and uses them to repair buggy programs with matching *loop* structures. Sarfgen [37] proposes a new method of embedding ASTs into numerical vectors to accelerate the process of searching for the closest correct solution. They fail to generate patches for buggy programs with unique control-flow/loop structures. *Refactory* [16] tries to address this problem by using refactoring rules to create multiple correct variants, but as noted earlier, it has limited effectiveness.

The state-of-the-art techniques for repairing student programs written in OCaml include FixML [22] and the newly released work Cafe [33]. While FixML is limited to fixing multi-location errors, Cafe leverages a context-aware matching algorithm to repair incorrect programs by using multiple partially matching reference programs to repair methods separately. However, in our approach, context-sensitive interprocedural analysis is not under discussion.

Newly released techniques for automated feedback generation take different repair strategies but introduce new limitations. For instance, *AssignmentMender* [23] uses *partial repair* to generate concise feedback by reusing fine-grained code snippets from submissions. However, its performance heavily relies on existing automated fault localization (AFL) techniques [3] and is limited in generating complex fixes, such as repairing multiple statements. On the other hand, Verifix [4] aims to generate verifiably correct program repairs as student feedback. But Verifix fails to generate repairs for incorrect programs with unique loop structures.

While the techniques mentioned above and our approach focus on code-level feedback, Fan et al. [12] open up a new avenue for exploration. They abstracted programs into "concept graphs" to provide concept-level feedback. These concept graphs contain expressions translated into natural language to enhance readability, making them more suitable as hints for students.

Large-scale language models (LLMs) like Copilot [2] and ChatGPT [1] have proven effective for code-related tasks. This paper evaluates ChatGPT on our dataset. While it performs well in some cases, careful evaluation and verification of its suggestions are crucial, especially for complex scenarios. Effective prompting, which enhances ChatGPT's performance, relies on traditional program analysis, underscoring the continued relevance of our work.

6 CONCLUSION

We present Brafar, a general feedback generation framework for programming assignments. To tackle two challenges, Brafar incorporates a *bidirectional refactoring* algorithm for control-flow repairs and employs a coarse-to-fine fault localization approach to reduce unnecessary repairs. Our evaluation results on real student submissions indicate Brafar can achieve a better repair success rate in a shorter time with a smaller repair size than existing approaches.

DATA AVAILABILITY

The tools/datasets implemented/analyzed during the current study are available at https://github.com/LinnaX7/brafar-python.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (No. 62232014 and No. 62372227). Yu Pei's work is supported in part by the Hong Kong Polytechnic University Fund under Grants P0051205 and P0051074.

REFERENCES

[1] 2023. *ChatGPT.* https://openai.com/blog/chatgpt/

[2] 2023. *Microsoft Copilot.* https://github.com/features/copilot/

[3] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 88–99. https://doi.org/10.1109/ASE.2009.25

[4] Umair Z. Ahmed, Zhiyu Fan, Jooyong Yi, Omar I. Al-Bataineh, and Abhik Roychoudhury. 2022. Verifix: Verified Repair of Programming Assignments. *ACM Trans. Softw. Eng. Methodol.* 31, 4, Article 74 (jul 2022), 31 pages. https://doi.org/10.1145/3510418

[5] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation error repair: for the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training* (Gothenburg, Sweden) *(ICSE-SEET '18)*. Association for Computing Machinery, New York, NY, USA, 78–87. https://doi.org/10.1145/3183377.3183383

[6] Umair Z. Ahmed, Nisheeth Srivastava, Renuka Sindhgatta, and Amey Karkare. 2020. Characterizing the Pedagogical Benefits of Adaptive Feedback for Compilation Errors by Novice Programmers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training* (Seoul, South Korea) *(ICSE-SEET '20)*. Association for Computing Machinery, New York, NY, USA, 139–150. https://doi.org/10.1145/3377814.3381703

[7] Andrea Arcuri. 2008. On the automation of fixing software bugs. In *Companion of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE Companion '08)*. Association for Computing Machinery, New York, NY, USA, 1003–1006. https://doi.org/10.1145/1370175.1370223

[8] L. Bergroth, H. Hakonen, and T. Raita. 2000. A survey of longest common subsequence algorithms. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*. 39–48. https://doi.org/10.1109/SPIRE.2000.878178

[9] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantitative Objectives. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 383–401. https://doi.org/10.1007/978-3-319-41540-6_21

[10] 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*. 785–790. https://doi.org/10.1109/ICDM.2014.142

[11] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

[12] Zhiyu Fan, Shin Hwei Tan, and Abhik Roychoudhury. 2023. Concept-Based Automated Grading of CS-1 Programming Assignments. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 199–210. https://doi.org/10.1145/3597926.3598049

[13] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. *SIGPLAN Not.* 53, 4, 465–480. https://doi.org/10.1145/3296979.3192387

[14] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Neural attribution for semantic bug-localization in student programs. , Article 1064 (2019), 11 pages.

[15] Tim Handerson. 2018. *Zhang-Shasha: Tree edit distance in Python.* https://github.com/timtadh/zhang-shashan

[16] Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-Factoring Based Program Repair Applied to Programming Assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 388–398. https://doi.org/10.1109/ASE.2019.00044

[17] 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* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 739–750. https://doi.org/10.1145/2950290.2950363

[18] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[19] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 593–604. https://doi.org/10.1145/3106237.3106309

[20] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 213–224. https://doi.org/10.1109/SANER.2016.76

[21] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[22] Junho Lee, Dowon Song, Sunbeom So, and Hakjoo Oh. 2018. Automatic diagnosis and correction of logical errors for functional programming assignments. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 158 (oct 2018), 30 pages. https://doi.org/10.1145/3276528

[23] Leping Li, Hui Liu, Kejun Li, Yanjie Jiang, and Rui Sun. 2023. Generating Concise Patches for Newly Released Programming Assignments. *IEEE Transactions on Software Engineering* 49, 1 (2023), 450–467. https://doi.org/10.1109/TSE.2022.3153522

[24] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. *SIGPLAN Not.* 51, 1, 298–312. https://doi.org/10.1145/2914770.2837617

[25] Yunlong Lu, Na Meng, and Wenxin Li. 2021. FAPR: Fast and Accurate Program Repair for Introductory Programming Courses. https://doi.org/10.48550/ARXIV.2107.06550

[26] Ken Masters. 2011. A Brief Guide To Understanding MOOCs. *The Internet Journal of Medical Education* 1, 2, 2. https://doi.org/10.5580/1f21

[27] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 691–701. https://doi.org/10.1145/2884781.2884807

[28] Benjamin Paassen, Barbara Hammer, Thomas W. Price, Tiffany Barnes, Sebastian Gross, and Niels Pinkwart. 2018. The Continuous Hint Factory - Providing Hints in Vast and Sparsely Populated Edit Distance Spaces. *Journal of Educational Data Mining* 10, 1 (Jun. 2018), 1–35. https://doi.org/10.5281/zenodo.3554697

[29] David M. Perry, Dohyeong Kim, Roopsha Samanta, and Xiangyu Zhang. 2019. SemCluster: clustering of imperative programming assignments based on quantitative semantic features. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 860–873. https://doi.org/10.1145/3314221.3314629

[30] 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* (Amsterdam, Netherlands) *(SPLASH Companion 2016)*. Association for Computing Machinery, New York, NY, USA, 39–40. https://doi.org/10.1145/2984043.2989222

[31] 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 *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 404–415. https://doi.org/10.1109/ICSE.2017.44

[32] 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* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2491956.2462195

[33] Dowon Song, Woosuk Lee, and Hakjoo Oh. 2021. Context-aware and data-driven feedback generation for programming assignments. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 328–340. https://doi.org/10.1145/3468264.3468598

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

[35] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 483–494. https://doi.org/10.1145/3180155.3180206

[36] Ke Wang, Rishabh Singh, and Zhendong Su. 2017. Dynamic Neural Program Embedding for Program Repair. *CoRR* abs/1711.07163 (2017). arXiv:1711.07163 http://arxiv.org/abs/1711.07163

[37] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, align, and repair: data-driven feedback generation for introductory programming exercises. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 481–495. https://doi.org/10.1145/3192366.3192384

[38] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 356–366. https://doi.org/10.1109/ASE.2013.6693094

[39] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. 364–374. https://doi.org/10.

1109/ICSE.2009.5070536

[40] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 416–426. https://doi.org/10.1109/ICSE.2017.45

[41] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 740–751. https://doi.org/10.1145/3106237.3106262

[42] Jialu Zhang, José Pablo Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2024. PyDex: Repairing Bugs in Introductory Python Assignments using LLMs. , 25 pages. https://doi.org/10.1145/3649850

[43] Kurtis Zimmerman and Chandan R. Rupakheti. 2015. An Automated Framework for Recommending Program Elements to Novices (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 283–288. https://doi.org/10.1109/ASE.2015.54

Received 2024-04-12; accepted 2024-07-03