Automated Feedback Generation for Programming Assignments through Diversification

Dongwook Choi
Department of Computer Science and Engineering
Sungkyunkwan University
Suwon, Republic of Korea
dwchoi95@skku.edu

Eunseok Lee*
College of Computing and Informatics
Sungkyunkwan University
Suwon, Republic of Korea
leees@skku.edu

***Abstract*—Immediate and personalized feedback on students' programming submissions is important for improving their programming skills. However, it is challenging for instructors to give personalized feedback to every student since each submission is written differently. To address this problem, the Automated Feedback Generation (AFG) technique has been proposed, which identifies faults from wrong program, generates patches, and provides feedback if the patches pass validation. AFG relies on correct programs from peers to find faults and generate patches. Therefore, having diverse correct programs is important for the performance of AFG. However, in small-scale programming courses or new online judge problems, might be a lack of diversity in correct programs. In this paper, we propose MENTORED, a new AFG for students' programming assignments through diversification. MENTORED generates new structures of programs through various combinations of programs to generate modifications optimized for wrong programs while solving the problem of dependency on correct programs. Additionally, MENTORED provides transparent feedback on the process of repairing wrong programs. We evaluate MENTORED on real student programming assignments and compare it with state-of-the-art AFG approaches. Our dataset includes real university introductory programming assignments and online judge problems. Experimental results show that MENTORED generates higher repair rates and more diverse program structures than other AFG approaches. Moreover, by providing a transparent sequence of repair processes, MENTORED is expected to improve students' programming skills and reduce instructors' manual effort in feedback generation. These results indicate that MENTORED can be a useful tool in programming education.**

***Index Terms*—Automated Program Repair, Automated Feedback Generation, Programming Assignments**

I. Introduction

Programming is not only a core tool for software development but also an essential learning tool that helps students apply theoretical knowledge to real-world problem-solving, fostering logical thinking and problem-solving skills. In recent years, the demand for coding skills has surged across various industries, highlighting the importance of programming education. As a result, educational institutions worldwide are strengthening programming education, with programming skills becoming a critical competency in diverse fields beyond just technical expertise. Programming assignments play a key role in this educational process, enabling students to apply their knowledge, debug errors, and solve problems. To maximize the educational impact, students need to receive personalized feedback in real-time while working on their programming assignments. However, since each student implements their programming assignments with different variable names and algorithms, providing personalized feedback requires significant time and effort, making it challenging for instructors to provide such feedback to every student. This issue is especially pronounced in environments like Massive Open Online Courses (MOOC), where hundreds of students participate simultaneously, making it practically impossible for instructors to provide personalized feedback to everyone. As a result, many students repeatedly make the same mistakes without understanding the errors in their code or knowing how to fix them. This situation can reduce students' motivation and sense of accomplishment, ultimately leading to a loss of interest in learning programming.

Automated Program Repair (APR) technique is a system that integrates various techniques to automatically identify faults in program code, generate patches to repair them, and verify the correctness of the repaired code. APR was primarily proposed for automatically repairing faults in Open Source Software (OSS), which is typically well-structured and written by experts, passing most test cases. Therefore, APR systems are highly effective when targeting such expert-level code. However, APR also holds great potential in programming education. When applied to student programming assignments, APR can automatically suggest fixes for wrong programs, providing personalized feedback. In a study by Yi et al. (2017) [1], APR tools [2]–[5] were applied to programming assignments written by students in an introductory programming course. Despite the relative simplicity of student code compared to OSS, the repair rate of the APR tool was low. This is because APR datasets primarily consist of expert-written code, focusing on single-location faults. In contrast, student programs are often poorly structured, contain multiple-location faults, and fail more than half of the test cases [1]. These characteristics limit the direct application of APR to programming assignments.

To overcome these limitations, Automated Feedback Generation (AFG) technique based on APR was proposed. AFG was developed to address APR's shortcomings, such as the multi-location fault issue, and to provide personalized feedback suitable for educational environments. Unlike APR, most AFG

techniques [6]–[19] do not use predefined templates [20] or repair models [21] that learn from fault, patch, and context information. Instead, AFG utilizes error models [12] that learn common mistakes made by students or compares the wrong program to peers' correct programs that have passed all test cases. The reason AFG tends to have a higher repair rate than APR is its reliance on correct programs. Since student submissions often contain many faults and fail more than half of the test cases, it is difficult to identify faults through test case-based fault localization [1]. However, by comparing the wrong program with the most structurally similar correct program, AFG can identify faults without relying on test cases. This method is unique to programming assignments where correct programs exist, highlighting a key distinction from APR. As a result, AFG outperforms APR, especially in handling poorly structured and fault-heavy student code. In essence, correct programs play a critical role in the effectiveness of AFG.

In large-scale courses like MOOC, the diversity of correct programs can be guaranteed, but in small-scale programming courses or with new online judge problems, there may be a shortage of correct programs. Refactory [6] generates diverse structures of patches by refactoring, but it still requires at least one correct program and cannot generate new algorithms or approaches. AssignmentMender [9] uses mutation to generate patches even when none exist, but random modifications can result in meaningless changes or new bugs. REFERENT [11] generates feedback without correct programs by learning from the modification history of online judge problems using Transformer-based deep learning, but it may apply incorrect algorithms due to the diversity in student assignment domains, and it only addresses single-location faults, making it unsuitable for real environments with multiple-location faults. PyDex [22] and PYFIXV [23] use Large Language Models (LLM) like Codex to repair wrong programs, but it is uncertain what data the LLM were trained on, and transparency in the repair process is limited, making it less ideal for students who need to understand the modifications. Additionally, fine-tuning these models is technically challenging and costly, placing a burden on instructors [24].

Genetic Programming (GP) is a probabilistic search method inspired by biological evolution to discover computer programs suited for specific tasks [25], [26]. GP-based techniques are particularly effective in problem-solving and program generation [2], [27]–[30]. GP can generate diverse feedback through various combinations of modifications such as crossover and mutation, making it excellent at creating new algorithms compared to refactoring [31]. Additionally, fitness evaluation of GP measures the quality of patches, while selection filters out less effective solutions, making it possible to provide effective feedback. This approach is beneficial both when correct programs are unavailable and when there are many correct programs, as GP efficiently narrows down the search space to propose suitable modifications for wrong programs, solving the problem of random modifications. Moreover, our proposed method can provide understandable feedback compared to deep learning by presenting the repair process transparently.

In this paper, we propose MENTORED, a GP-based AFG technique that generates diverse feedback. MENTORED utilizes GP to repair students' wrong programs and provides various information transparently used in the repair process as feedback. Various information provided as feedback is discussed in Section III-B.

The contribution of this paper is as follows:

- **Less dependence on correct programs.** MENTORED identifies fault locations by analyzing dynamic execution information of the program and generates patches through generations of GP iterations to provide feedback without structurally similar correct programs. This shows high performance with less dependence on correct programs compared to other AFG techniques.
- **Transparent feedback.** MENTORED uses various information of the program to find faults and measures the quality of patches from various perspectives to generate feedback. Unlike deep learning-based patch generation methods, it provides various information used in the repair process transparently, providing students and instructors with easy-to-understand and reliable feedback.
- **Diverse feedback.** MENTORED generates diverse feedback with a small amount by generating various combinations through genetic operations. In addition, due to multi-objective optimization and randomness, it can provide various feedback for the same wrong program. This helps improve programming skills by providing students with various perspectives and solutions.

II. Related Work

In recent years, research on Automated Feedback Generation (AFG) has been proposed to provide real-time personalized feedback on students' programming assignments based on AFG techniques.

AutoGrader [12] generates feedback on wrong programs using an error model that defines rules to correct common errors that students often make. AutoGrader corrects wrong programs based on the minimum difference with the correct program and provides specific feedback based on the difference. The error model may vary depending on the problem, and the correction is limited because students make various mistakes.

J Yi et al. [1] investigate the feasibility of automatically generating feedback for novice programming assignments using APR technology. They evaluated GenProg [2], AE [3], Angelix [4], and Prophet [5] as APR tools. The experimental results showed that only about 30% of the student programs were modified because most of the student programs failed in most of the tests. Therefore, they provided students with hints for partially passing the tests using APR tools. As a result, students improved their correctness rate by 84% by modifying their code with hints. These results suggest that APR technology is not suitable for student programs with many faults that fail most test cases.

CLARA [8] uses already submitted correct programs to cluster and correct new wrong programs. It corrects the wrong

program with the most similar but correct expression using the expressions of the correct programs in the cluster. The cluster is formed by programs with the same control flow, so there is a limitation that wrong programs with different control flows cannot be corrected.

SARFGEN [7] automatically generates data-based feedback through the three steps of Search, Align, and Repair in the process of repairing wrong programs. The system identifies and applies minimal modifications based on syntactic and semantic differences using a large number of student submissions on the scale of MOOCs. However, it cannot be applied when there is no correct program, such as in small courses or new problems in online judges. Also, it cannot generate effective feedback without a sufficient number of correct programs.

Refactory [6] generates feedback by finding correct programs with the same control flow structure as the wrong program by applying refactoring rules to correct programs. Refactory allows feedback to be generated even when there are few correct programs by creating new structures of correct programs through refactoring. However, it may cause misunderstandings to novice programmers as feedback by inserting unnecessary code such as "if (True): pass" to achieve an equivalent control flow structure.

AssignmentMender [9] generates feedback using wrong programs as well as correct programs, unlike existing AFG approaches. AssignmentMender is divided into three repair methods: mutation, reference, and last resort. Mutation-based repair corrects faults through simple editions such as Operator Replacement, Variable Name Replacement, and Statement Deletion. Reference-based repair uses static analysis and graph-based matching algorithms to find and apply patches for faults. In the last resort, it randomly selects and replaces a bug-free block to correct it. AssignmentMender can generate new patches due to mutation, but it can also create new bugs.

PyFixV [23] is based on OpenAI's Codex model and generates feedback by receiving a student's code with syntax errors and integrating the corrected program with natural language descriptions. PyFixV also solves the verification mechanism of the patch through the Codex model, which makes it difficult to generate consistent quality feedback because it is difficult to guarantee 100% accuracy unlike the existing verification mechanism. In addition, it can be burdensome for instructors to use it continuously because it costs too much [24].

PyDex [22] is used to automatically correct students' code errors in Python programming assignments. This proposed approach can handle both syntactic and semantic errors using Large Language Model on Code (LLMC). PyDex corrects errors using multiple modal prompts, iterative queries, test case-based minority sample selection, and program chunking. Specifically, the system corrects the wrong program by integrating various inputs such as the student's buggy code, assignment description, test cases, and provides the optimal solution that satisfies the test cases among multiple candidates. LLMC can generate more code than necessary and can change even the correct parts of the program, which may lead to excessive patches. This can interfere with students' understanding or learning of the corrected code.

III. Background and Example

In this section, we describe GP-based repair approaches such as GenProg and ARJA, which are the background of MENTORED, and provide examples of the feedback generated by MENTORED.

A. Genetic Programming based Repair

Genetic Programming (GP) is a probabilistic search method inspired by biological evolution to discover computer programs suitable for specific tasks [25], [26]. GP transforms existing programs using computational analogs of biological mutation and crossover to create new programs. This method has been applied to solve various software engineering problems and has proven its effectiveness in software bug repair. GenProg [2] utilizes GP to repair bugs in OSS by reusing existing code without generating new statements, which significantly reduces the search space during the repair process. By applying mutations only to the parts of the code executed by failed test cases, GenProg narrows the search space compared to repairing the entire program. Whild this increases the speed and efficiency of repair, but it may limit the diversity of possible repaires. ARJA [27] proposes automatic program repair of Java programs through multi-objective GP. ARJA uses lower-granularity patch representation [32] to subdivide patches into likely-buggy locations, operation types, and potential fix ingredients, allowing GP to explore the search space more effectively with minimal changes. Additionally, by optimizing multiple objective functions using NSGA-II [33], ARJA increases repair rates and reduces task time. This approach allows for more precise identification of faulty locations and proposes effective repairs with minimal changes.

Our proposed MENTORED is based on the existing approaches of GenProg and ARJA, but has the following key differences and challenges:

- **Quality and diversity of candidate patches**: GenProg and ARJA use OSS written by experts as datasets, which allows them to secure candidate patches of relatively high quality and diversity. In contrast, MENTORED uses programs written by novice students as its datasets, leading to lower quality and a lack of diversity in the candidate patches.
- **Complexity of fixes**: APR techniques like GenProg and ARJA primarily generate patches for single faults, resulting in relatively simple fixes (one-hunk changes). In contrast, AFG techniques like MENTORED typically require more complex fixes (multiple-chunk changes).

Thus, it is highly challenging to identify the correct patch that requires complex fixes from candidate patches with low quality and diversity.

B. Feedbacks

The feedbacks that MENTORED generates to support students in improving their programming skills help students

understand their errors and acquire correct programming skills. The feedbacks we provide are as follows:

1) **Failed Test cases**: Provides input and expected output of failed test cases, and the output of the student program. This helps students to see how their program produces incorrect output for a given input and how it differs from the expected output.
2) **Execution Traces**: Provides execution traces of both wrong and correct programs for each failed test cases. This helps students visually understand where errors occur during execution.
3) **Variable Value Sequence**: Displays the sequence of variable value changes in the execution of both wrong and correct programs. This helps students to identify logic errors more clearly by tracking variable changes.
4) **Fault locations & Patches**: Provides the exact fault locations in the wrong program and the patches applied to fix them. This helps students pinpoint the errors and learn specific ways to repair them.
5) **History of Fixes**: Provides the modification history of the wrong program over multiple GP iterations. This helps students to track the order of fixes and improve their debugging skills.

These feedback elements are designed to help students deeply understand their programming faults and effectively debug and modify their code. The feedback provided through MENTORED is expected to offer educational value that goes beyond simply pointing out errors, enabling students to genuinely improve their programming skills. While MENTORED provides various information as feedback, for comparison with other AFG techniques, it is assumed that only the patch is provided. In other words, feedback and patch are used synonymously.

IV. Approach

A. Overview

An overview of the MENTORED, automated feedback generation using Genetic Programming (GP) is presented in Algorithm 1. MENTORED inputs wrong programs P_W, correct programs P_C, a set of test cases T and maximum number of population size pop_size. And outputs the correctly fixed program as solutions S. The working process of the algorithm is briefly explained as follows:

First, in the initialization step, the solution set S for P_W is assigned as an empty set (line 1). Then, the population P of the GP is initialized with P_W and P_C (line 2). The process repeats the steps of $selection$, $swtVariable$, $faultLocalization$, $programFix$, $validation$, and $fitness$ until the criterion is satisfied, generating S for P_W (lines 3–15). The criterion is either to generate a solution p_f that passes the $validation$ for all P_W or to repeat the process for the maximum number of generations. In the $selection$ step, pairs of p_w and a reference program p_r are selected up to the maximum population size pop_size (line 4, Section IV-B). Note that p_r can be either p_c or p_w. In the $swtVariable$ step, the variable names in p_r are switched to match those in p_w since student programs may use different variable names (line 5, Section IV-C). In the $faultLocalization$ step, the suspicion of faults is calculated line-by-line to locate the program's fault (line 6, Section IV-D). In the $programFix$ step, a fixed program p_f is generated by fixing the fault in p_w with reference to p_r based on the suspicion (line 7, Section IV-E). If p_f passes all tests in T during $validation$, it is added to S (lines 8–10). Next, $fitness$ scores of p_f and p_w are compared, and if p_f has a better score, it is added to P (lines 11–13, Section IV-B). Finally, the solution set S for P_W is returned (line 16).

Algorithm 1: MENTORED

Input: $P_W - Wrong\ programs$
$P_C - Correct\ programs$
$T - Set\ of\ test\ cases$
$pop\_size - Population\ size$
Output: $S - Solutions$

```
1  S ← ∅;
2  P ← P_W ⋃ P_C ;                    /* Population */
3  repeat
4      foreach ⟨p_w, p_r⟩ ∈ selection(P, pop_size) do
5          p_r ← swtVariable(p_w, p_r, T);
6          susp ← faultLocalization(p_w, p_r, T);
7          p_f ← programFix(p_w, p_r, susp, T);
8          if validation(p_f, T) passes all T then
9              S ← S ⋃ p_f;
10         end
11         if fitness(p_w) < fitness(p_f) then
12             P ← P ⋃ p_f;
13         end
14     end
15 until criterion is satisfied;
16 return S
```

B. Selection

In general GP, $selection$ pairs up to $pop\_size/2$ from P, as the goal is typically to find a single optimal solution. However, in our case, the goal is to generate solutions for every P_W. Therefore, we select p_w from the entire P in the amount of pop_size. Then, through $selection$, pair p_w with p_r to generate patches. Our $selection$ method uses elite selection[1] and NSGA-III (Non-dominated Sorting Genetic Algorithm) [34]. Since P can grow infinitely, as mentioned in line 12 of Algorithm 1, we need to sample the best solutions to reduce the search space. Thus, through elite selection, we sample pop_size programs from P_C and S. If P_C and S do not exist, we perform random selection from P in the amount of pop_size. Then, using $fitness\ Evaluation$ (Section IV-F), we calculate the scores of the samples. Based on these scores, NSGA-III selects the optimal solution p_r from the samples.

[1]**Elite selection** is one of the commonly used evolutionary strategies where the top individuals from the population are selected and copied as they are, helping to find optimal solutions faster.

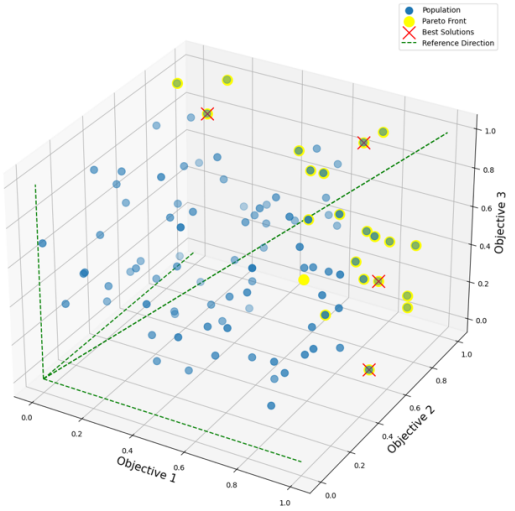

Fig. 1: Example of Selection using Pareto Front and NSGA-III.

Fig. 1 illustrates an example of the $selection$ method using NSGA-III in MENTORED. For three objectives normalized to 0-1, the $fitness\ function$ calculates the scores of each sample. Then, only the samples on the pareto front for each objective are selected. This is because, when there are multiple objective functions, it is often impossible for a single solution to be optimal across all objectives simultaneously. In such cases, the pareto front is used to find optimal solutions by considering the trade-offs between objectives. In this way, the selection is further narrowed to the best samples for each objective through the pareto front. NSGA-III then generates n reference directions and selects the solutions closest to these reference directions. In Fig. 1, four reference directions are created: one that is optimal across all objectives [1,1,1], and three that are optimal for only one objective each—[0,0,1], [0,1,0], [1,0,0]. The solutions closest to each reference direction are selected, resulting in four solutions, from which p_r is chosen via random selection to be paired with p_w.

C. Switch Variables

Students' programs can be written with different variable names. Therefore, before generating patches, the variable names of p_w and p_r must be unified. To unify the variable names of the two programs, map the variable names through four strategies. First, in Dynamic Equivalence Analysis (DEA) [6], run both programs using T and collect the Variable Value Sequences (VVS) of all variables. If there are variables with the same VVS between the two programs, map them as equivalent variables. Second, for the variables that were not mapped by DEA, calculate the Longest Common Sequence (LCS) of their VVS and use it for map the leftover variables. Third, map the variables of the two programs through type matching. In type matching, we only compare their types not values. Lastly, the fourth strategy maps variables with the same names. As previously mentioned, p_w is selected as the wrong program and p_r is selected as the reference program to repair p_w. Therefore, switch the variable names of p_r to the those of p_w with the mapped variables to unify the variable names of both programs.

D. Fault Localization

Most AFG approaches [6]–[8], [12], [15] identify the fault location of the wrong program by comparing it with syntactically or semantically different parts of the correct program. This method can identify non-fault location as faults when the correct program is not diverse or structurally different from the wrong program. To address this, we use a Fault Localization (FL) method based on test cases. Among FL methods, the most popular one is Spectrum-Based Fault Localization (SBFL) [35], which analyzes the execution spectrum of a program to identify statements that are most likely to contain faults. It calculates the suspiciousness of statements based on the execution frequency and the success/failure results of the test cases. Representative techniques include Tarantula [36] and Ochiai [37]. However, traditional FL methods were mainly developed for expert-level programs like APR. As a result, they struggle to effectively identify faults in student programs, which tend to have fewer test cases and shorter code. Specifically, since student code often has fewer branching points, these methods tend to assign the same level of suspiciousness to all lines within a block. VsusFL [38] was proposed to address this issue in student programs. VsusFL collects the VVS of the wrong program and finds the correct program with the most similar VVS. Then, it identifies the lines of the wrong program with different VVS from the correct program as fault locations and calculates the suspiciousness. In our approach, we assume the p_r selected during the $selection$ phase to be the correct program for p_w and use VsusFL to calculate the line-by-line suspiciousness for p_w.

E. Program Fix

The $programFix$ step generates a child program by crossover and mutation of the two parent programs $\langle p_w, p_r \rangle$ paired in the $selection$ step. Algorithm 2 describes how to generate a child program. In the $programFix$, it takes as input two paired parent programs p_w and p_r in the $selection$ step, the fault suspicion $susp$ identified in the $faultLocalization$ step, a set of test cases T, and a history of fixes H. And outputs the fixed program p_f, which is a child program that fixes the faults in p_w. The algorithm's process can be briefly explained as follows:

First, in $nodeMap$, represent p_w and p_r as Abstract Syntax Tree (AST) and map the nodes of p_w and p_r based on the execution trace from T, assigning them to N (line 1). Next, in $crossoverMap$, the cut point for the crossover, N_{cut}, is assigned from N (line 3). In the $mutationMap$, the mutation point, N_{mut}, is assigned from N (line 4). By combining N_{mut} and N_{cut}, we assign the final fixable nodes, N_{fix} (line 5). Generate p_f with fixed p_w using N_{fix} in $fixer$ (line 6). If the pair of $\langle p_w, p_f \rangle$ is not in the history of fixes H, then we add the pair into H, and return p_f (lines 7–9). Else, N_{fix}

Algorithm 2: programFix

Input: $p_w - Wrong\ program$
$p_r - Reference\ program$
$susp - Suspiciousness\ of\ faults$
$T - Set\ of\ test\ cases$
$H - History\ of\ fixes$
Output: $p_f - Fixed\ program$

1 $N \leftarrow nodeMap(p_w, p_r, T)$;
2 **repeat**
3 $N_{cut} \leftarrow crossoverMap(N)$;
4 $N_{mut} \leftarrow mutationMap(N, susp)$;
5 $N_{fix} \leftarrow N_{cut} \bigcup N_{mut}$;
6 $p_f \leftarrow fixer(p_w, N_{fix})$;
7 **if** $\langle p_w, p_f \rangle \notin H$ **then**
8 $H \leftarrow H \bigcup \langle p_w, p_f \rangle$;
9 **return** p_f;
10 **else**
11 $N \leftarrow N - N_{fix}$;
12 **end**
13 **until** $N = \varnothing$;
14 **return** $None$;

is removed from N (lines 10–12). This process is repeated until there are no more nodes in N (lines 3–13). Finally, if p_f cannot be generated, return $None$ (line 14).

Node Mapping is the process of selecting how to modify the program. Since modifications are made based on the nodes of the program, we map the nodes of p_r relative to the fix location of p_w using three operations: $replace$, $insert$, and $delete$. The node mapping is based on the execution trace of the program, and extracts the execution paths of each program using T. In this process, the line numbers of each program may not match, so we replace each line with the nodes corresponding to the reserved words of the statement that makes up the line. For example, the execution paths of p_w and p_r are as follows(subscripts represent line numbers, and subscripts represent reserved words):

$$E_{p_w} \leftarrow [n^1_{Func}, n^2_{For}, n^3_{If}, n^4_{Assign}, n^5_{If}, n^6_{Continue}, n^7_{Return}]$$
$$E_{p_r} \leftarrow [n^1_{Func}, n^2_{If}, n^3_{Return}, n^4_{For}, n^5_{If}, n^6_{Assign}, n^7_{Return}]$$

First, the node mapping for $replace$ is done using the LCS. By calculating the LCS of E_{p_w} and E_{p_r} and mapping the corresponding nodes, we get $N_{rep} \leftarrow \{n^1_{Func} : n^1_{Func},\ n^2_{For} : n^4_{For},\ n^3_{If} : n^5_{If},\ n^4_{Assign} : n^6_{Assign},\ n^7_{Return} : n^7_{Return}\}$. Next, the node mapping for $insert$ involves mapping the nodes of p_r that were not mapped in N_{rep}. Additionally, for $insert$, we need to determine the position where the node will be inserted. We determine the position based on the nodes mapped in N_{rep}. If a node at the same depth as the insert node exists in N_{rep}, the new node is inserted as a sibling of that node. If same depth of node doesn't exists, it is inserted as a child node. Therefore, after mapping the unmapped parts from N_{rep}, we get $N_{ins} \leftarrow \{n^1_{Func} : n^2_{If},\ n^1_{Func} : n^3_{Return}\}$. Finally, the node mapping for $delete$ maps the unmapped nodes of p_w in N_{rep}, resulting in $N_{del} \leftarrow \{n^5_{If} : None,\ n^6_{Continue} : None\}$. Ultimately, we have $N \leftarrow N_{rep}; \bigcup; N_{ins}; \bigcup; N_{del}$. This mapping is used to connect the faults of p_w and patches of p_r. Traditional AFG approaches rely on mapping based on Control Flow Structure (CFS), which requires the CFS of both programs to match. However, since our $nodeMap$ is based on the execution trace, it is not restricted by CFS.

Crossover is an evolutionary operation used in GP to generate a new child program by exchanging some statements between two parent programs. It models the phenomenon of gene crossover in biology and is used in GP to enhance program efficiency and diversity. To perform crossover, a cut point must be selected. Since we use $one-point-crossover$, we randomly select one of the mapped nodes N_{rep} from $nodeMap$ as N_{cut}.

Mutation is an evolutionary operation in GP that introduces random changes to a program to alter its behavior or structure. This process plays a crucial role in increasing diversity in genetic algorithms and preventing falling into local optima. We perform mutation through three edit actions: $replace$, $insert$, and $delete$. Unlike $crossoverMap$, $mutationMap$ selects N_{mut} based on the suspicion score from $susp$. For N_{mut}, a roulette wheel selection [39] is performed based on the suspicion score of the candidates from the three mapped sets—N_{rep}, N_{ins}, and N_{del} from $nodeMap$. And randomly select one of the edit actions of the selected node.

Repair generates a child program p_f for p_w using N_{fix}, which is the combination of the fix patterns N_{cut} and N_{mut} selected by $crossoverMap$ and $mutationMap$. Fig 2 shows an example of possible cases that can be repaired using N_{fix}. In Fig 2, Parent1 p_w and Parent2 p_r are represented as AST. The red node in p_w is the faulty node, and the blue node in

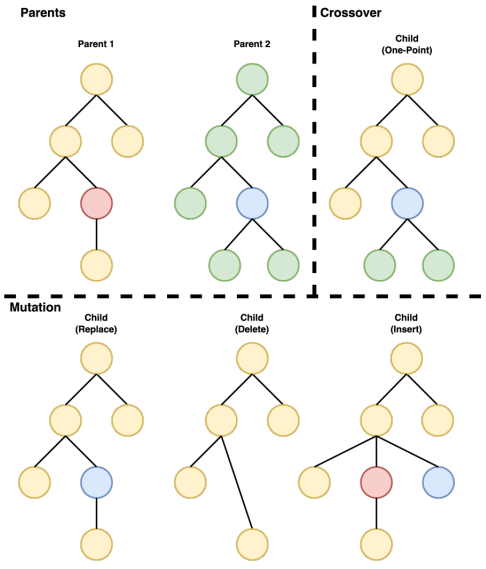

Fig. 2: Genetic operators for AST

p_r is the patch. The figure demonstrates how crossover and mutation are applied. In $one-point-crossover$, which we choose for the crossover strategy, all the child nodes of the faulty node (red) are replaced by the patch node (blue) and its child nodes. In the $replace$ mutation, the faulty node is replaced with the patch node. In $delete$ mutation, the faulty node is removed, and if it has child nodes, its parent node is connected to the child nodes. In $insert$ mutation, the patch node is inserted as a sibling node, positioned as the next sibling of the faulty node. While crossover modifies all child nodes of the selected node, mutation only modifies the selected nodes themselves.

F. Fitness Evaluation

Fitness evaluation is a function used to assess how well a program solves a given problem. This function is one of the core elements of GP, as it measures the performance of individual programs and determines which candidates will be selected for future generations. Fitness evaluation is used in the $selection$ step (Section IV-B) when selecting p_r, the pair of p_w, and in line 12 of Algorithm 1, where the best-performing programs are added to the population P for future generations. We define four objective functions to measure the performance of programs. All objective functions evaluate the performance of programs relative to p_w. Additionally, all objective functions have values between 0 and 1, where values closer to 1 indicate higher fitness. The four objective functions are formulated as follows:

$$f_1(\mathbf{p}') = \frac{|\{t \in T_F |\ \mathbf{p}'\ passes\ t\}|}{|T_F|} \tag{1}$$

$$f_2(\mathbf{p}') = \frac{|\{t \in T_P |\ \mathbf{p}'\ passes\ t\}|}{|T_P|} \tag{2}$$

$$f_3(\mathbf{p}') = \frac{\sum_{t \in T_{F \to P}} E_{sim}(\mathbf{p}', t)}{|T_{F \to P}|} \tag{3}$$

$$f_4(\mathbf{p}') = \frac{\sum_{t \in T_{P \to P}} E_{sim}(\mathbf{p}', t)}{|T_{P \to P}|} \tag{4}$$

$f_1(\mathbf{p}')$ is an objective function that calculates the number of test cases passed by p' that were failed by p_w. T_F refers to the test cases that p_w failed. $f_2(\mathbf{p}')$ is an objective function that calculates the number of test cases passed by p' that were also passed by p_w. T_P refers to the test cases that p_w passed. $f_3(\mathbf{p}')$ is an objective function that calculates the similarity based on the execution trace for the test cases that p_w failed but p' passed. $T_{F \to P}$ refers to the test cases that p_w failed but p' passed. $f_4(\mathbf{p}')$ is an objective function that calculates the similarity based on the execution trace for the test cases passed by both p_w and p'. $T_{P \to P}$ refers to the test cases that were passed by both p_w and p'. $E_{sim}(\mathbf{p}', t)$ refers to the execution trace-based similarity between the two programs. The execution trace-based similarity is calculated by dividing the LCS of the execution traces E_{trace} of p_w and p' by the MAX number of nodes. The LCS corresponds to the number of nodes mapped in the node mapping process during the program fix phase (Section IV-E). The formula for execution trace-based similarity is as follows:

$$E_{sim}(\mathbf{p}', t) = \frac{LCS(E_{trace}(p_w, t), E_{trace}(\mathbf{p}', t))}{MAX(E_{trace}(p_w, t), E_{trace}(\mathbf{p}', t))} \tag{5}$$

Determining which objective function is superior or inferior for each case is challenging. Therefore, instead of assigning weights to each objective function, we evaluate the fitness of programs using NSGA-III. This method effectively handles multiple objective functions and ensures diverse solutions, allowing for various modifications.

V. Experimental Design

In this section, we describe the research questions to be answered, baseline approaches for performance comparison, experimental datasets, evaluation metrics, and parameters for MENTORED experiments.

A. Research Questions

To evaluate MENTORED, this study aims to answer the following research questions.

RQ1. How effectively can MENTORED generate feedback with and without correct programs? As previous AFG approaches show, the more correct programs available, the better the feedback generation rate [9], [11]. Conversely, when correct programs are absent, generating feedback becomes challenging. In real-world scenarios, such as small-scale programming course assignments or new online judge problems, they may not have correct programs. Therefore, we compare how effectively MENTORED generates feedback both with and without correct programs, in relation to previous approaches.

RQ2. How effectively can MENTORED generate diverse feedback? Since AFG generates patches using correct programs, a lack of diversity in correct programs can result in feedback providing fixed programs with the same structure for different wrong program structures. This can be detrimental to students who need to improve their programming skills through exposure to diverse algorithms. Therefore, we examine how effectively MENTORED overcomes these limitations and generates diverse feedback.

RQ3. How effectively can MENTORED generate high-quality feedback? The quality of feedback is important for improving student learning and repairing errors. Therefore, we measure the quality of feedback and compare it with previous approaches to review how effectively MENTORED generates high-quality feedback.

B. Baselines

To evaluate how effectively MENTORED generates feedback, we selected publicly available state-of-the-art AFG techniques. Refactory [6] is the most recent publicly available approach with both datasets and tools, to the best of our knowledge. AssignmentMender [9] was excluded due to discrepancies between the publicly available dataset and the

dataset mentioned in the paper, as well as the tool being non-executable. PyDex [22] is not publicly available as a tool, but since it is based on LLM and the prompt was made publicly available in the author's paper, we implemented it ourselves for the experiments. Tools for MENTORED and PyDex are available in Section IX.

C. Datasets

To compare our approach, we selected two of the most recent feedback generation approaches with publicly available datasets: Refactory [6] and AssignmentMender [9]. Refactory's dataset consists of programming assignments from a Python programming basics course at the author's university, collected from five assignments, including 2,442 correct programs and 1,783 wrong programs. AssignmentMender's dataset was collected from the ten most popular programming problems on the online judge site CodeForces [40], consisting of 372 correct programs and 128 wrong programs. However, upon reviewing AssignmentMender's publicly available dataset [41], we found 320 correct programs and 142 wrong programs, differing from the numbers reported in the paper. Therefore, we conducted our evaluation based on the publicly available dataset. As a result, we experimented with a total of 15 programming assignments, consisting of 2,973 correct programs and 1,925 wrong programs.

D. Evaluation Metrics

To evaluate the performance of AFG, we use three evaluation metrics that are most commonly used in previous studies. First, Repair Rate (RR) represents the rate of fixed wrong programs. In RR, P_W refers to the wrong programs, and S refers to the solutions for P_W. Second, Relative Patch Size (RPS) measures the edit distance between the fixed program and the wrong program. TED stands for tree edit distance, AST refers to Abstract Syntax Tree, p_w represents the wrong program, and p_f represents the fixed program. Lastly, Average Time Taken (ATT) refers to the average time (in seconds) taken to fix one wrong program. The formulas for these three evaluation metrics are as follows:

$$RR = |S| \ / \ |P_W|$$
$$RPS = TED(AST(p_w), AST(p_f) \ / \ Size(AST(p_w))$$
$$ATT = Time(sec) \ of \ AFG \ / \ |P_W|$$

E. Parameters

Before conducting the experiment, we report the global parameters. Since MENTORED is based on GP, it could potentially run indefinitely without a set number of generations. Also, results may vary due to randomness. Therefore, we limit the maximum number of generation iterations N, to 30 and set the maximum population size pop_size, to the length of wrong programs $|P_W|$ to prevent the population from growing infinitely and expanding the search space due to the children generated by GP. Lastly, we measure the average performance through 100 trials. The parameter values were determined through multiple experiments to balance the trade-offs between the elements of the evaluation metrics. For example, if only RR is considered, increasing the number of generation iterations to 100 will result in a higher RR, but ATT and RPS may increase.

All experiments were conducted on a MacOS(sonoma 14.5) with an Apple M1 Ultra 20-core processor machines with 64 GB memory.

VI. Experimental Results

In this section, we describe the experimental results for the research questions.

A. RQ1. How effectively can MENTORED generate feedback with and without correct programs?

Table I shows the experimental results for each programming assignment. The values in parentheses in Table I are the results of the experiments without correct programs $W/O\ \chi$, and the values in front of the parentheses are the results of the experiments with correct programs $W/\ \chi$. However, Refactory requires at least one correct program. Therefore, only Refactory was experimented with $W/O\ \chi$ using one correct program, and PyDex and MENTORED experimented

TABLE I: Results of AFG approaches on 15 programming assignments

NO	Title	#Correct Programs	#Wrong Programs	$W/\ \chi$ ($W/O\ \chi$)								
				Repair Rate (RR)			Relative Patch Size (RPS)			Average Time Taken (ATT/sec)		
				Refactory	PyDex	MENTORED	Refactory	PyDex	MENTORED	Refactory	PyDex	MENTORED
1	Theatre Square	30	20	0.45 (0.95)	**1.00** (**1.00**)	**1.00** (0.60)	0.51 (0.97)	**0.26** (**0.25**)	0.33 (0.90)	2.6 (3.3)	0.5 (1.5)+240	**1.8** (**1.8**)
2	Watermelon	14	31	0.97 (0.19)	0.97 (0.26)	**1.00** (**1.00**)	0.54 (0.90)	0.50 (**0.19**)	**0.26** (0.28)	0.4 (3.0)	0.4 (1.2)+240	**0.3** (**0.3**)
3	Domino Piling	40	10	0.80 (0.00)	0.80 (**0.80**)	**1.00** (**0.80**)	0.26 (nan)	**0.07** (**0.07**)	0.21 (0.66)	**0.3** (3.3)	0.5 (1.5)+240	0.6 (**0.5**)
4	Petya and Strings	38	13	0.92 (0.77)	**1.00** (**0.85**)	**1.00** (0.62)	0.68 (0.89)	**0.35** (**0.20**)	0.44 (0.29)	12.6 (6.8)	0.6 (1.6)+240	**0.9** (**0.8**)
5	String Task	32	16	0.94 (0.94)	**1.00** (**1.00**)	**1.00** (**1.00**)	0.47 (1.16)	**0.13** (**0.15**)	0.17 (0.17)	5.6 (6.9)	1.4 (1.5)+240	**3.2** (**1.8**)
6	Next Round	24	25	**1.00** (**1.00**)	**1.00** (0.44)	**1.00** (**1.00**)	0.53 (0.77)	0.37 (**0.15**)	**0.35** (0.61)	4.6 (5.4)	0.8 (1.5)+240	**0.7** (**0.8**)
7	Beautiful Matrix	42	8	**1.00** (**0.88**)	**1.00** (0.75)	**1.00** (0.12)	0.45 (0.76)	**0.23** (**0.03**)	0.47 (**0.03**)	1.7 (4.1)	0.6 (1.4)+240	**1.2** (**0.9**)
8	Stones on the Table	44	9	0.67 (0.89)	**1.00** (0.67)	**1.00** (**1.00**)	0.49 (0.52)	**0.31** (**0.18**)	**0.31** (0.41)	4.9 (5.5)	0.8 (1.8)+240	**2.4** (**1.9**)
9	Word Capitalization	45	6	0.33 (**1.00**)	0.50 (0.33)	**1.00** (**1.00**)	0.72 (1.63)	**0.56** (**0.50**)	0.61 (1.22)	**0.2** (**0.2**)	0.4 (1.2)+240	0.4 (0.3)
10	Helpful Maths	47	4	0.75 (**1.00**)	**1.00** (**1.00**)	**1.00** (0.25)	0.78 (1.87)	**0.09** (0.24)	0.45 (**0.10**)	4.8 (27.0)	0.8 (1.6)+240	**3.9** (**1.9**)
11	Sequential Search	768	575	0.98 (0.60)	**1.00** (0.89)	0.97 (**0.98**)	0.39 (0.68)	**0.29** (**0.26**)	**0.29** (0.33)	3.0 (3.3)	1.8 (2.3)+240	**1.7** (**3.1**)
12	Unique Dates and Months	291	435	0.79 (0.55)	**0.90** (**0.81**)	0.62 (0.54)	0.46 (0.63)	**0.25** (**0.22**)	0.27 (0.24)	**2.8** (**4.1**)	1.7 (2.0)+240	4.7 (7.1)
13	Duplicate Elimination	546	308	0.97 (0.63)	0.98 (**1.00**)	**1.00** (0.99)	0.33 (0.67)	**0.31** (**0.32**)	0.38 (0.35)	4.5 (3.2)	1.7 (1.7)+240	**2.6** (**3.1**)
14	Sorting Tuples	419	357	0.84 (0.62)	**1.00** (**0.98**)	0.97 (0.97)	**0.26** (1.75)	0.44 (0.31)	0.39 (**0.30**)	6.7 (7.8)	1.7 (1.7)+240	**4.6** (**4.4**)
15	Top-K	418	108	0.84 (0.95)	**1.00** (**0.99**)	**1.00** (0.97)	0.29 (0.78)	**0.22** (**0.25**)	**0.22** (0.37)	14.4 (**4.9**)	0.8 (1.2)+240	**7.9** (6.5)
Overall		2973	1925	0.82 (0.73)	0.94 (0.78)	**0.97** (**0.80**)	0.48 (1.00)	**0.29** (**0.22**)	0.34 (0.41)	4.6 (5.9)	1.0 (1.6)+240	**2.5** (**2.4**)

Results are shown as with correct program $W/\ \chi$ and without correct program(in parentheses) $W/O\ \chi$.
In $W/O\ \chi$, Refactory uses a single correct program by policy. PyDex and MENTORED do not use any correct programs.

with zero correct programs. In real programming assignments corresponding to NO. 11∼15, the one correct program for Refactory is selected as the instructor's correct program, not a student's correct program. For Online Judge problems NO. 1∼10, since there were no correct programs from the instructor, the first correct program submitted by a student was selected.

First, in terms of RR, our MENTORED(0.97, 0.80) repaired the most wrong programs compared to Refactory(0.82, 0.73) and PyDex(0.94, 0.78) in both $W/O\ \chi$ and $W/\ \chi$. Also, all techniques show better performance in $W/\ \chi$ than in $W/O\ \chi$. Naturally, the more correct programs there are, the more correct patches can be created, so more wrong programs can be fixed. Therefore, the important point is to fix wrong programs well even when there are few correct programs, such as in small programming assignments. MENTORED repaired all wrong programs with a RR of 1.00 in NO. 2, 5, 6, 8, 9 assignments, even though it was $W/O\ \chi$. This shows that MENTORED can effectively generate feedback even in small-scale programming assignments.

Next, in terms of RPS, Refactory generated patches with few modifications with 0.48 when $W/\ \chi$, but with many modifications with 1.00 when $W/O\ \chi$. This means that AFG techniques dependent on correct programs can generate patches when $W/O\ \chi$, but many modifications are required. Therefore, an effective AFG technique should have a low RPS even when $W/O\ \chi$. MENTORED generated patches with a smaller size than Refactory in all aspects with 0.34 when $W/\ \chi$ and 0.41 when $W/O\ \chi$. However, PyDex generated patches with a smaller size than MENTORED with 0.29 when $W/\ \chi$ and 0.22 when $W/O\ \chi$. This is because the Large Language Model (LLM) used in PyDex was trained on 45 terabytes of text [42]. Since the training data for the model used in PyDex hasn't been disclosed, we can't be completely sure. However, given that the experimental dataset consists of problems from the well-known online judge CodeForces and that only PyDex shows that RPS is lower than $W/O\ \chi$ (0.22) when $W/\ \chi$ (0.29), it is highly likely that the model was trained on this dataset. Therefore, considering this possibility, MENTORED generated feedback with a simple RPS difference of about 0.1 despite the much less modification material, so MENTORED can be considered effective in terms of RPS.

Before explaining the results of ATT, it should be noted that PyDex required additional time of "+240". This means the time taken for API calls of the LLM model in PyDex. This additional time adds about 240 seconds to fix one wrong program, as it is the time taken to call the API m times for each of the n prompts. In terms of ATT, MENTORED(2.5, 2.4) generated feedback faster than Refactory(4.6, 5.9) and PyDex(241, 241.6) in both $W/O\ \chi$ and $W/\ \chi$. Also, MENTORED generated feedback twice as fast as Refactory and 120 times faster than PyDex. In programming courses where real-time feedback is important, AFG's ATT should be provided within seconds, so MENTORED is the most effective.

Answer to RQ1. MENTORED shows the highest RR and the fastest ATT both with and without correct programs and shows the smallest RPS except for PyDex, where the use of correct programs is uncertain. These results show that MENTORED can provide the most effective feedback when correct programs are absent or insufficient.

B. RQ2. How effectively can MENTORED generate diverse feedback?

TABLE II: Comparison of Program Control Flow Structure

	wrong ⇔ patch		patch ⇔ correct	
	$Same$	$Diff$	$Exist$	$Unique$
Refactory	548	656	810	394
PyDex	995	724	1439	280
MENTORED	1116	600	1316	400

To measure how diversely the feedback generated by each technique, we measured the Control Flow Structure (CFS) of wrong, patch, and correct programs. Then, to measure the diversity of feedback, we divided them into $Same$ and $Diff$, $Exist$ and $Unique$. This is because it is good to provide new solutions by diversifying feedback, but it should not damage the structure of the original program. $Same$ and $Diff$ represent the change in CFS between wrong and patch programs. And $Exist$ and $Unique$ represent the difference in CFS between patch and correct programs. $Same$ means creating a patch program while maintaining the CFS of the wrong program, and $Diff$ means changing the structure of the wrong program. $Exist$ means that the patch program has the same CFS as the correct programs of peers, and $Unique$ means that the patch program has a unique CFS that is not in the correct programs of peers. In the experimental results of Table II, in $Same$, MENTORED maintained the CFS of the wrong program and generated a patch program with 1116, which is 200% more than Refactory and 112% more than PyDex. And in $Unique$, MENTORED generated a unique patch program with 400, which is 101% more than Refactory and 142% more than PyDex.

Answer to RQ2. Compared to existing AFG techniques, MENTORED generated the most unique patch programs with new CFS that were not in the correct programs of peers while maintaining the CFS of the wrong program. This shows that MENTORED can generate diverse feedback.

C. RQ3. How effectively can MENTORED generate high-quality feedback?

To measure the quality of feedback, we used the Pylint library [43]. Pylint is a static analysis tool for Python that comprehensively evaluates code across five categories (fatal, error, warning, convention, refactor) to quantitatively assess

quality. It has been widely employed in numerous studies [44]–[47]. Figure 3 presents a comparison of feedback quality. The quality score is out of 10. In the $W/\ \chi$ condition where correct programs are provided, MENTORED performed the highest quality score of 6.6, followed by PyDex with 6.5 and Refactory with 5.5. On the other hand, in $W/O\ \chi$, PyDex performed the highest score of 6.5, while MENTORED attained a second-highest score of 5.7, and Refactory scored 4.5. These results indicate that MENTORED delivers superior quality in $W/\ \chi$, outperforming PyDex. Although PyDex obtained a slightly higher score in $W/O\ \chi$, MENTORED still maintained consistently high quality. A key strength of MENTORED is that it does not rely on LLM. While PyDex can exhibit high performance by generating code based on LLM, the instability of LLM and issues of data bias can compromise the accuracy and consistency of the generated feedback [22]. Additionally, since LLM operate as black boxes, it is challenging to explain the logical basis for the modifications generated by PyDex or to present the process transparently. In contrast, MENTORED ensures transparency in the feedback generation process and provides interpretable feedback that aids students in understanding their mistakes. Furthermore, PyDex carries the risk of data leakage, which can undermine the reliability of the evaluation results. Conversely, MENTORED mitigates such issues through its proprietary algorithms, reduces data dependency, and provides more stable and reliable feedback.

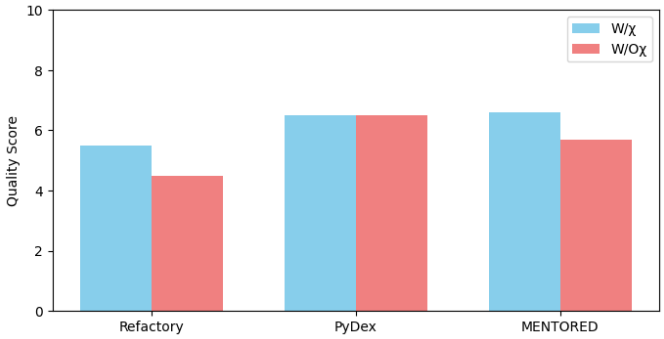

Fig. 3: Comparison of Feedback Quality

Answer to RQ3. MENTORED achieved top feedback quality with correct programs and maintained high quality even without. While PyDex performs better without correct programs, MENTORED offers more transparent and interpretable feedback than the black-box LLM-based PyDex. This feedback is especially useful in educational environments, effectively supporting student learning.

VII. Threats to Validity

To support a more reasonable comparison, we reimplemented PyDex which is the most recent and competitive approach to our knowledge. However, PyDex are not publicly available, so we tried to follow the original paper as much as possible, but there may be some differences in the implementation. For instance, the original LLM engine of PyDex is no longer supported, so we used the closest later supported engine "gpt-3.5-turbo". This change may introduce a threat to internal validity, potentially affecting the performance of PyDex. To mitigate this threat, we have made the source code of the reimplemented PyDex publicly available on Github for peer-review to reproduce our results and verify the correctness of the implementation. Additionally, our proposed method, MENTORED, is also publicly available on Github at IX.

There may be threats to external validity due to the datasets and evaluation metrics chosen for the experiments. To minimize this threat, we selected various datasets from real university programming assignments and online judges, as used in existing approaches. Additionally, we chose evaluation metrics that are commonly used in AFG approaches.

Finally, we acknowledge potential threats to conclusion validity, as statistical variations could influence the outcomes of our comparisons. To ensure that our results are robust and reliable, we performed multiple runs of 100 trials, accounting for possible variations due to the inherent randomness of GP.

VIII. Conclusion

Existing AFG techniques that rely on correct programs face limitations in generating feedback when correct programs are unavailable. In this paper, we propose a new approach, MENTORED, that can generate diverse feedback without depending on correct programs. The key idea of MENTORED is to utilize Genetic Programming (GP) to generate diverse patches and find the optimal solution through iterative repair processes. Through evaluations on real student programming assignments, we demonstrated that MENTORED can produce more diverse and higher-quality feedback compared to existing techniques. Notably, MENTORED provides interpretable feedback by transparently providing the iterative repair process, allowing students to understand and fix their faults step by step. Moreover, while maintaining the structure of the original(wrong) program, it sometimes suggests solutions with different structures, helping students learn from and experience various solutions. These strengths contribute to improving programming education, reducing the instructors' workload, and enhancing students' programming skills.

IX. Data Availability

Our MENTORED tool implementation of the approach is available at https://github.com/dwchoi95/mentored

Acknowledgment

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.RS-2024-00438686, Development of software reliability improvement technology through identification of abnormal open sources and automatic application of DevSecOps)

REFERENCES

[1] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, "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*, 2017, pp. 740–751.

[2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[3] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 356–366.

[4] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[5] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[6] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, and A. Roychoudhury, "Refactoring based program repair applied to programming assignments," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 388–398.

[7] K. Wang, R. Singh, and Z. Su, "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*, 2018, pp. 481–495.

[8] S. Gulwani, I. Radiček, and F. Zuleger, "Automated clustering and program repair for introductory programming assignments," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 465–480, 2018.

[9] L. Li, H. Liu, K. Li, Y. Jiang, and R. Sun, "Generating concise patches for newly released programming assignments," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 450–467, 2022.

[10] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "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*, 2016, pp. 39–40.

[11] J. Heo, H. Jeong, D. Choi, and E. Lee, "Referent: Transformer-based feedback generation using assignment information for programming course," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 2023, pp. 101–106.

[12] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 15–26.

[13] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 129–139.

[14] U. Z. Ahmed, Z. Fan, J. Yi, O. I. Al-Bataineh, and A. Roychoudhury, "Verifix: Verified repair of programming assignments," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–31, 2022.

[15] L. D'Antoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantitative objectives," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 383–401.

[16] D. Choi, J. Heo, and E. Lee, "Automated feedback generation for multiple function programs," in *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2021, pp. 582–583.

[17] D. Song, W. Lee, and H. Oh, "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*, 2021, pp. 328–340.

[18] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 404–415.

[19] S. Kaleeswaran, A. Santhiar, A. Kanade, and S. Gulwani, "Semi-supervised verified feedback generation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 739–750.

[20] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 31–42.

[21] X. Li, S. Liu, R. Feng, G. Meng, X. Xie, K. Chen, and Y. Liu, "Transrepair: Context-aware program repair for compilation errors," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[22] J. Zhang, J. P. Cambronero, S. Gulwani, V. Le, R. Piskac, G. Soares, and G. Verbruggen, "Pydex: Repairing bugs in introductory python assignments using llms," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 1100–1124, 2024.

[23] T. Phung, J. Cambronero, S. Gulwani, T. Kohn, R. Majumdar, A. Singla, and G. Soares, "Generating high-precision feedback for programming syntax errors using large language models," *arXiv preprint arXiv:2302.04662*, 2023.

[24] H. Joshi, J. C. Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, "Repair is nearly generation: Multilingual program repair with llms," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 4, 2023, pp. 5131–5140.

[25] S. Forrest, "Genetic algorithms: principles of natural selection applied to computation," *Science*, vol. 261, no. 5123, pp. 872–878, 1993.

[26] J. Koza, "On the programming of computers by means of natural selection," *Genetic programming*, 1992.

[27] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on software engineering*, vol. 46, no. 10, pp. 1040–1067, 2018.

[28] A. E. Eiben, J. E. Smith, A. Eiben, and J. Smith, "Genetic algorithms," *Introduction to Evolutionary Computing*, pp. 37–69, 2003.

[29] H. Cao, D. Han, F. Liu, T. Liao, C. Zhao, and J. Shi, "Code similarity and location-awareness automatic program repair," *Applied Sciences*, vol. 13, no. 14, p. 8519, 2023.

[30] K. Harada and K. Maruyama, "Towards efficient program repair with apr tools based on genetic algorithms," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2024, pp. 896–901.

[31] M. Silva-Muñoz, C. Contreras-Bolton, C. Rey, and V. Parada, "Automatic generation of a hybrid algorithm for the maximum independent set problem using genetic programming," *Applied Soft Computing*, p. 110474, 2023.

[32] V. P. L. Oliveira, E. F. Souza, C. Le Goues, and C. G. Camilo-Junior, "Improved crossover operators for genetic programming for program repair," in *Search Based Software Engineering: 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings 8*. Springer, 2016, pp. 112–127.

[33] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[34] J. Blank, K. Deb, and P. C. Roy, "Investigating the normalization procedure of nsga-iii," in *International Conference on Evolutionary Multi-Criterion Optimization*. Springer, 2019, pp. 229–240.

[35] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "Spectrum-based multiple fault localization," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 88–99.

[36] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for fault localization," in *in Proceedings of ICSE 2001 Workshop on Software Visualization*. Citeseer, 2001.

[37] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 2006, pp. 39–46.

[38] Z. Li, S. Wu, Y. Liu, J. Shen, Y. Wu, Z. Zhang, and X. Chen, "Vsusfl: Variable-suspiciousness-based fault localization for novice programs," *Journal of Systems and Software*, vol. 205, p. 111822, 2023.

[39] A. E. Eiben and J. E. Smith, *Introduction to evolutionary computing*. Springer, 2015.

[40] CodeForces, "Codeforces — codeforces.com," https://codeforces.com.

[41] H. L. Leping Li, K. L, and R. S. Yanjie J, "Dastaset of AssignmentMender," https://github.com/CoPaGe/FeedbackExperimentTask, 2022.

[42] T. B. Brown, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[43] Logilab, "Pylint - code analysis for Python — www.pylint.org — pylint.org," https://www.pylint.org.
[44] M. L. Siddiq, L. Roney, J. Zhang, and J. C. D. S. Santos, "Quality assessment of chatgpt generated code and their use by developers," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 152–156.
[45] S. Horschig, T. Mattis, and R. Hirschfeld, "Do java programmers write better python? studying off-language code quality on github," in *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*, 2018, pp. 127–134.
[46] G. D. Apostolidis, "Evaluation of python code quality using multiple source code analyzers," 2023.
[47] M. Agarwal, K. Talamadupula, S. Houde, F. Martinez, M. Muller, J. Richards, S. Ross, and J. D. Weisz, "Quality estimation & interpretability for code translation," *arXiv preprint arXiv:2012.07581*, 2020.