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 assignments is important for improving their programming skills. However, it is challenging for instructors to give personalized feedback to every student since each program 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 pass validation. AFG relies on correct programs from students 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 faults in their code or knowing how to repair 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, generate patches to repair them, and validate the correctness of the repaired program. 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 program. However, APR also holds great potential in programming education. When applied to student programming assignments, APR can automatically suggest modifications 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 program compared to OSS, the repair rate of the APR tool was low. This is because APR datasets primarily consist of expert-written program, 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 student program of poorly structured and many faults. 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 faults 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 repair common errors that students often make. AUTOGRADER repairs 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 modification is limited because students make various mistakes.

QLOSE [15] find a correct program that is most similar to the wrong program based on program distance, which calculates both syntactic and semantic changes in the program, for repair wrong program. QLOSE repairs based on predefined templates corresponding to linear combinations of all constants and variables in the program using the sketch program synthesizer. However, the repair is limited because more complex modifications are required in real student programs.

J Yi et al. [1] investigate the feasibility of automatically generating feedback for novice programming assignments using APR technique. 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 repaired 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 repairing their code with hints. These results suggest that APR technique is not suitable for student programs with many faults that fail most test cases.

CLARA [8] clusters correct programs with the same Control Flow Structure (CFS) for repair wrong program. It repairs the wrong program by utilizing correct programs within the cluster that have the same CFS as the wrong program. However, for minimal repair, there is a limit that correct programs must diverse, and the wrong program with different CFS cannot be repaired.

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] can repair wrong programs without correct programs through mutation-based and reference-based repair. Mutation-based repair repairs faults through simple editions such as *Operator Replacement*, *Variable Name Replacement*, and *Statement Deletion*. And Reference-based repair uses static analysis and graph-based matching algorithms to find patches for faults and repair them. However, this methods can generate new patches, but conversely, it can also generate new faults. Therefore, it shows very low performance when correct programs are not used.

PYFIXV [23] is based on OpenAI's Codex model and generates feedback by receiving a student's program with syntax errors and integrating the repaired program with natural language explanation. PYFIXV also solves the validation 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 test-based validation mechanism. In addition, it can be burdensome for instructors to use it continuously because it costs too much [24].

PYDEX [22] also repairs wrong programs using the Codex model. PYDEX generates repaired programs by combining various inputs such as wrong programs, assignment descriptions, and test cases, and finally selects the repaired program with minimal changes among several candidates. Large Language Models such as Codex model are black-box models, so it may be difficult to understand the repair process when unnecessary code is generated or when parts of the program that are not faults are repaired. This can interfere with students' understanding or learning of the modifications.

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 example of 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. While 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 such as 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 repairs**: APR techniques like GENPROG and ARJA primarily generate patches for single faults, resulting in relatively simple repairs (one-hunk changes). In contrast, AFG techniques like MENTORED typically require more complex repairs (multiple-chunk changes).

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

B. Example of Feedback

Fig. 1 shows the feedback generated by MENTORED without correct programs. The programs are sequential search programming assignment to find the location where x is inserted into seq. The wrong program in Fig. 1a (denoted by p_w) has faults in lines 8 and 9. Fig. 1b (denoted by p_{r1}) and Fig. 1c

```python
def search(x, seq):
    for i in range(len(seq)):
        if x < seq[i]:
            return i
        elif seq[i] == seq[-1] and seq[i
            ] < x:
            return i + 1
        elif seq[i] < x <= seq[i + 1]:
            return i
    return len(seq) + 1
```

(a) Wrong Program

```python
def search(x, seq):
    for i in range(len(seq) - 1):
        if x <= seq[i + 1]:
            return i
    return len(seq)
```

(b) Refernce Program 1

```python
def search(x, seq):
    for index in range(len(seq) - 1):
        if x <= index:
            return 0
        if seq[index] <= x <= seq[index
            + 1]:
            return index + 1
        if x >= seq[-1]:
            return len(seq)
    return len(seq)
```

(c) Refernce Program 2

```
Failed Testcase:
 - status: Failure
 - input:  search(-100,())
 - expect: 0
 - output: 1

Execution Traces:
 - wrong:   [1,2,9]
 - correct: [1,2,5]

Variable Value Sequence:
 - wrong:   {"x": [-100],"seq": [()]}
 - correct: {"x": [-100],"seq": [()]}



Fault Location & Patch:
 - In line 9, fix "return len(seq) + 1"
              to "return len(seq)"
```

(d) Modifications 1

```
Failed Testcase:
 - status: Failure
 - input:  search(7,[1,5,10])
 - expect: 2
 - output: 1

Execution Traces:
 - wrong:   [1,2,3,5,7,2,3,5,7,8]
 - correct: [1,2,3,5,7,2,3,5,6]

Variable Value Sequence:
 - wrong:   {"x":[7],"seq":[[1,5,10]],
             "i":[0,1]}
 - correct: {"x":[7],"seq":[[1,5,10]],
             "index":[0,1]}

Fault Location & Patch:
 - In line 8, fix "return i"
              to "return i + 1"
```

(e) Modifications 2

Fig. 1: Feedback by MENTORED without Correct Programs

(denoted by p_{r2}) are also wrong programs, but they are used as reference programs because they can be used as ingredients to repair lines 9 and 8 of Fig. 1a, respectively. Fig. 1d and Fig. 1e show the process of repairing the wrong program through GP iterations. In the first modifications, p_w outputs a different value from the expected output of the testcase, and p_{r1} outputs the correct value. Also, by comparing the execution traces of p_w and p_{r1}, we can see that the last line is different, $[1, 2, 9]$ and $[1, 2, 5]$, respectively. Therefore, we repair line 9 of p_w to line 5 of p_{r1}. However, since line 8 is still unresolved, a second repair is required. The second modifications proceeds in the same way as the first repair using p_{r2}, but this time, the variable names of the two programs are different, so we replace the variable $index$ of p_{r2} to i with the same variable value sequence. Next, we repair line 8 of p_w to line 6 of p_{r2} to repair all faults. Of course, MENTORED can repair all faults at once using lines 6 and 9 of p_{r2}. However, this sequential repairs may sometimes be easier to understand. The feedback we provide is 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 wrong 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 faults 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 faults more clearly by tracking variable changes.
4) **Fault locations & Patches**: Provides the exact fault locations in the wrong program and the patches applied to repair them. This helps students pinpoint the faults and learn specific ways to repair them.
5) **History of Repairs**: Provides the modifications of the wrong program over multiple GP iterations. This helps students to track the order of repairs and improve their debugging skills.

Unlike the black-box approach of LLM, the feedback we provide is transparent, allowing students to easily understand the repair process. It also helps students improve their debugging skills to solve problems on their own. Therefore, we expect our feedback to be helpful to students in improving their programming skills.

IV. Approach

A. Overview

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 $S \leftarrow \varnothing$;
2 $P \leftarrow P_W \bigcup P_C$; /* Population */
3 **repeat**
4 **foreach** $\langle p_w, p_r \rangle \in selection(P, pop\_size)$ **do**
5 $p_r \leftarrow swtVariable(p_w, p_r, T)$;
6 $susp \leftarrow faultLocalization(p_w, p_r, T)$;
7 $p_f \leftarrow programRepair(p_w, p_r, susp, T)$;
8 **if** $validation(p_f, T)$ *passes all* T **then**
9 $S \leftarrow S \bigcup p_f$;
10 **end**
11 **if** $fitness(p_w) < fitness(p_f)$ **then**
12 $P \leftarrow P \bigcup p_f$;
13 **end**
14 **end**
15 **until** $criterion\ is\ satisfied$;
16 **return** S

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 programs (P_F) as solutions (S). The working processes of the algorithm are briefly explained as follows:

First, in the initialization process, the S is initialized as an empty set (line 1). Then, the population (P) of the GP is assigned with sum of P_W and P_C (line 2). Then the repair processes are repeated until the criterion is satisfied (lines 3–15). The criterion is either to generate the fixed program (p_f) as the solution that passes the validation for all P_W or to repeat the processes up to the maximum number of generations (N). In the *selection*, pairs of p_w and a reference program (p_r) are selected up to the maximum pop_size (line 4, Section IV-B). Note that p_r can be either p_c or p_w. In the *swtVariable*, 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*, the suspiciousness ($susp$) of statements is calculated to locate the faults (line 6, Section IV-D). In the *programRepair*, p_f is generated by repairing the faults in p_w using p_r and $susp$ (line 7, Section IV-E). If p_f passes all T in *validation*, it is added to S (lines 8–10). Next, *fitness* calculates fitness scores of p_f and p_w, and if p_f has a better fitness score, it is added to P (lines 11–13, Section IV-B). Finally, the S for P_W is returned (line 16).

B. Selection

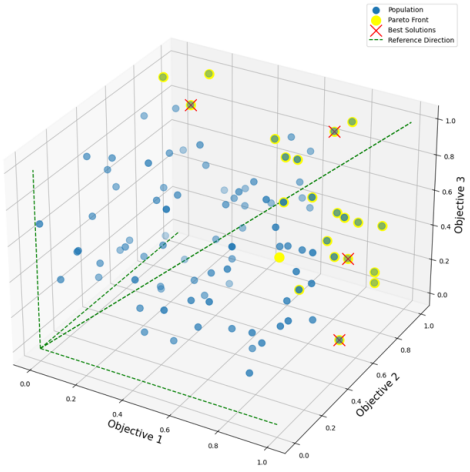

Fig. 2: Example of Selection using Pareto Optimization and NSGA-III.

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 set pop_size to $|P_W|$ and select pairs of p_w and p_r up to pop_size. 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 programs up to pop_size from P_C and S. If P_C and S does not exist, we samples from P_W. 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 for p_w from the samples. Fig. 2 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 set 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 optimization 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 set. NSGA-III then generates reference directions and selects the solutions closest to these reference directions. In Fig. 2, has four reference directions: one that is optimal across all objectives $[1, 1, 1]$, and three that are optimal for each objective $[0, 0, 1]$, $[0, 1, 0]$, $[1, 0, 0]$. At least four solutions closest to each reference direction are selected, and one p_r is finally selected through random selection and paired with p_w.

[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.

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, we 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 locations as faults when the correct program is not diverse, or structurally different from the wrong program. To address this problem, 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 ($susp$) 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 such as 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 $susp$ to all statements 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 statements of the wrong program with different VVS from the correct program as fault locations and calculates the $susp$. In our approach, we set the p_r selected during the *selection* phase to be the correct program for p_w and use VsusFL to calculate the $susp$ of statements for p_w.

E. Program Repair

To repair the wrong program, patches corresponding to the faults are required. Therefore, we utilize execution traces of the program to find patches for the faults in p_w from p_r. Before mapping the faults and patches, we first represent the two programs as Abstract Syntax Trees (AST) since we repair the program based on the nodes. Next, we map the nodes of the two programs according to the three methods of *replace*, *insert*, and *delete*. The node mapping is based on the execution traces of the program, and extracts the execution paths of each program using T. In this process, the line numbers of each program may not same, so we replace each line of execution paths with the nodes corresponding to the reserved words of the statement that makes up the line. First, the node mapping for *replace* is mapped using the LCS. Next, the node mapping for *insert* maps the nodes of p_r that were not mapped in the *replace* map. However, *insert* must also determine where to insert. Therefore, we insert the node as a sibling node if there is a node with the same depth as the *insert* node among the mapped nodes from *replace*, and as a child node if there is no node with the same depth. Finally, the node mapping for *delete* maps the nodes of p_w that were not mapped in the *replace* map as None, as they are to be deleted. The mapped nodes are used to connect the faults of p_w with the patches of p_r. The mapping method of existing AFG approaches [6]–[8] based on the Control Flow Structure (CFS) has the limitation that the CFS of the two programs must match. However, our node mapping is not limited by the CFS as it is based on the execution traces.

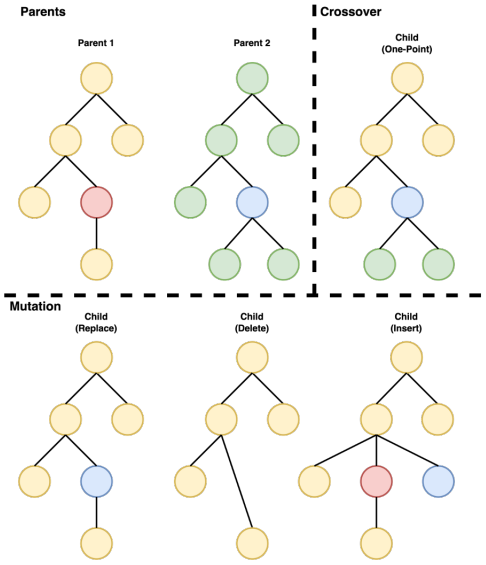

Fig. 3: Genetic operators for AST

Genetic Programming (GP) typically uses crossover and mutation to create new child programs. Crossover is an evolutionary operation used in GP to exchange parts of the code of two-parent programs. It models the phenomenon of gene crossover in biology and increase the efficiency and diversity of programs. On the other hand, Mutation introduces random changes to the code of a program to alter its behavior or structure. It also models the phenomenon of gene mutation in biology and increase diversity of programs and preventing them from falling into local optima. Therefore, we also use

crossover and mutation for our repair process. First, to perform crossover, we use *one-point-crossover* to repair p_w. Thus, we randomly select one of the nodes mapped in *replace* as a cut point and exchange the node of p_w with the node of p_f. Next, we perform mutation based on the three edit actions of *replace*, *insert*, and *delete*. Mutation selects the location to mutate based on the $susp$. This is done by performing roulette wheel selection [39] on the nodes mapped in *replace*, *insert*, and *delete* as mutation point, and applying the selected action and node. Fig 3 shows the repair process through crossover and mutation. Assuming $Parent1$ is p_w and $Parent2$ is p_r. The red node of $Parent1$ are faults, and the blue node of $Parent2$ are patches. In crossover, *one-point-crossover* exchanges fault node (red) and its children with patch node (blue) and its children. In mutation, *replace* replaces the fault node with the patch node. *delete* deletes the fault node, if there are nodes below the fault node it connects the parent node and child node of the fault node. *insert* inserts the fault node as the next sibling node of the fault node if two nodes have the same depth otherwise, it inserts to children. In crossover, the patch node and all of its children are repaired, but in mutation, only the patch node are repaired. Through these repair methods, the faults in p_w are repaired to generate p_f.

F. Fitness Evaluation

Fitness evaluation is used to evaluate how well the program solves a given problem. Fitness evaluation is one of the core elements of GP, as it calculates the performance of individual programs and determines which candidates will be selected for future generations. we use it in the *selection* process (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 calculate 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 means 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 means the test cases that p_w passed. $f_3(\mathbf{p}')$ is an objective function that calculates the similarity based on the execution traces for the test cases that p_w failed but p' passed. $T_{F \to P}$ means 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 traces for the test cases passed by both p_w and p'. $T_{P \to P}$ means the test cases that were passed by both p_w and p'. $E_{sim}(\mathbf{p}', t)$ means the execution traces-based similarity between the two programs. It 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 repair process (Section IV-E). The formula for execution traces-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

A. Research Questions

Since AFG techniques provide different forms of feedback, we treat only patch as feedback for comparison experiments. We set the following research questions to evaluate MENTORED.

RQ1. How effectively can MENTORED generate feedback with and without correct programs? Previous AFG approaches demonstrate that having more correct programs available improves the rate of feedback generation [9], [11]. In contrast, the absence of correct programs makes it challenging to generate feedback. In real-world situations, such as small-scale programming course assignments or new online judge problems, there may be no correct programs. Therefore, we assess how effectively MENTORED generates feedback both with and without correct programs, comparing its performance 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 faults. 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]. Dataset from REFACTORY consists of programming assignments from a Python programming introductory course at the author's university, collected from 5 assignments, including 2,442 correct programs and 1,783 wrong programs. Dataset from ASSIGNMENTMENDER was collected from the 10 most popular programming problems on the online judge site CodeForces[2], consisting of 372 correct programs and 128 wrong programs. However, upon reviewing dataset of ASSIGNMENTMENDER [40], 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 means the wrong programs, and S means the solutions for P_W. Second, Relative Patch Size (RPS) calculates TED between the fixed program (p_f) and the wrong program (p_w). Lastly, Average Time Taken (ATT) means the average time (in seconds) taken to repair single wrong program. The formulas for these three evaluation metrics are as follows:

$$RR = \frac{|S|}{|P_W|}$$

$$RPS = \frac{TED(AST_w, AST_f)}{Size(AST_w)}$$

$$ATT = \frac{time\ taken\ of\ approach}{|P_W|}$$

[2]https://codeforces.com

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

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 are the results

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.

without correct programs ($W/O\ \chi$), and the front of the parentheses are the results with correct programs ($W/\ \chi$). Notably, only REFACTORY requires at least one correct program. Therefore, $W/O\ \chi$ of REFACTORY used a single correct program. Whereas, PYDEX and MENTORED were tested without any correct programs. The single correct program used in REFACTORY corresponds to programming assignments NO. 11∼15, which are actual university assignments where the instructor's correct program is selected. For programming assignments NO. 1∼10, which corresponds to online judge problems, doesn't have instructor's correct program. Therefore, the first submitted correct program was selected.

First, in terms of RR, our MENTORED (0.97, 0.80) repaired the most wrong programs in both $W/O\ \chi$ and $W/\ \chi$ compared to REFACTORY (0.82, 0.73) and PYDEX (0.94, 0.78). Additionally, all approaches performed better in $W/\ \chi$ than in $W/O\ \chi$. Naturally, as the number of correct programs increases, the number of correct patches that can be generated also increases, leading to more wrong programs being repaired. Therefore, an important consideration is that even when the number of correct programs is small, such as in assignments for small-scale programming courses, wrong programs should still be effectively repaired. Notably, MENTORED repaired all wrong programs on assignments NO. 2, 5, 6, 8, and 9 even without correct programs, demonstrating its effectiveness in small-scale classes.

Next, in terms of RPS, both MENTORED (0.34, 0.41) and REFACTORY (0.48, 1.00) generated fewer modifications to generate patches in $W/\ \chi$ than in $W/O\ \chi$. This means that AFG techniques dependent on correct programs can generate patches in $W/O\ \chi$, but require many modifications. Therefore, an effective AFG technique should have a low RPS even in $W/O\ \chi$. MENTORED generated smaller patches than REFACTORY, but PYDEX (0.29, 0.22) generated even smaller patches. This is because the LLM used in PYDEX was trained on 45 terabytes of text [41]. However, our dataset consisted from the well-known problems and that only PYDEX (0.22, 0.29) shows that RPS is lower than $W/O\ \chi$ when $W/\ \chi$, 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 fix ingredients, MENTORED can be considered effective in terms of RPS.

Before explaining the results of ATT, PYDEX required additional time due to the API call time of the LLM model. This additional time is the time taken to call the API m times for each of the n prompts, adding approximately 240 seconds to repair a single wrong program. In terms of ATT, our 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 $W/\ \chi$. Furthermore, MENTORED was twice as fast as REFACTORY and 120 times faster than PYDEX. In programming courses where real-time feedback is crucial, feedback should be provided within seconds, demonstrating that MENTORED is the most effective approach in ATT.

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, fixed, 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 fixed programs. And $Exist$ and $Unique$ represent the difference in CFS between fixed and correct programs. $Same$ means generating a fixed program while maintaining the CFS of the wrong program, and $Diff$ means changing the structure of the wrong program. $Exist$ means that the fixed program has the same CFS as the correct programs, and $Unique$ means that the fixed program has a unique CFS that is not in the correct programs. In the experimental results of Table II, in $Same$, MENTORED maintained the CFS of the wrong program and generated a fixed program with 1116, which is 200% more than REFACTORY and 112% more than PYDEX. And in $Unique$, MENTORED generated a unique fixed 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 fixed programs with new CFS that were not in the correct programs while maintaining the CFS of the wrong program. This shows that MENTORED can generate the most effective diverse feedback.

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

To measure the quality of feedback, we used the Pylint library. Pylint[3] is a static analysis tool for Python that comprehensively evaluates code across five categories (fatal, error,

[3]https://www.pylint.org

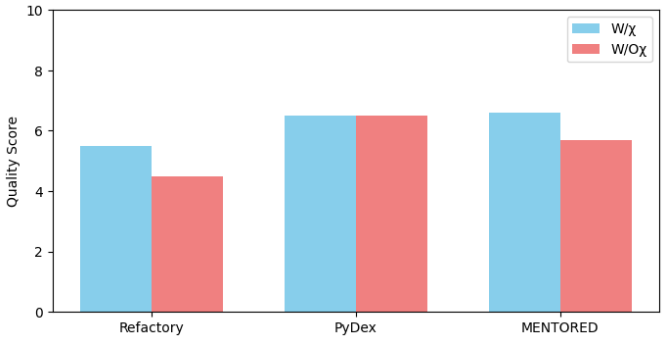

Fig. 4: Comparison of Feedback Quality

warning, convention, refactor) to quantitatively assess quality. It has been widely employed in numerous studies [42]–[45]. Fig 4 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.

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 the best of 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 rely on correct programs. The key idea of MENTORED is to utilize genetic programming 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 repair 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] H. L. Leping Li, K. L, and R. S. Yanjie J, "Dastaset of AssignmentMender," https://github.com/CoPaGe/FeedbackExperimentTask, 2022.

[41] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[42] 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.
[43] 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.
[44] G. D. Apostolidis, "Evaluation of python code quality using multiple source code analyzers," 2023.
[45] 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.