Generating Concise Patches for Newly Released Programming Assignments

Leping Li, Hui Liu, Kejun Li, Yanjie Jiang, and Rui Sun

Abstract—In programming courses, providing students with concise and constructive feedback on faulty submissions (programs) is highly desirable. However, providing feedback manually is often time-consuming and tedious. To release tutors from the manual construction of concise feedback, researchers have proposed approaches such as *CLARA* and *Refactory* to construct feedback automatically. The key to such approaches is to fix a faulty program by making it equivalent to one of its correct reference programs whose overall structure is identical to that of the faulty submission. However, for a newly released assignment, it is likely that there are no correct reference programs at all, let alone correct reference programs sharing identical structure with the faulty submission. Therefore, in this paper, we propose *AssignmentMender* generating concise patches for newly released assignments. The key insight of *AssignmentMender* is that a faulty submission can be repaired by reusing fine-grained code snippets from submissions (even when they are faulty) for the same assignment. It automatically locates suspicious code in the faulty program and leverages static analysis to retrieve reference code from existing submissions with a graph-based matching algorithm. Finally, it generates candidate patches by modifying the suspicious code based on the reference code. Different from existing approaches, *AssignmentMender* exploits faulty submissions in addition to bug-free submissions to generate patches. Another advantage of *AssignmentMender* is that it can leverage submissions whose overall structures are different from those of the to-be-fixed submission. Evaluation results on 128 faulty submissions from 10 assignments show that *AssignmentMender* improves the state-of-the-art in feedback generation for newly released assignments. A case study involving 40 students and 80 submissions further provides initial evidence showing that the proposed approach is useful in practice.

Index Terms—Feedback generation, program repair, programming assignments

1 Introduction

Providing students with concise and constructive feedback on faulty programs (e.g., submitted assignments) is highly desirable [1], [2]. Quite often students in programming courses cannot complete programming assignments correctly the first time. In such cases, concise feedback from tutors is highly desirable for students because they may quickly identify and fix problems based on concise feedback [3]. However, providing feedback manually is often time-consuming and tedious. Considering the large number of assignments submitted to a small number of tutors, it is highly challenging for tutors to provide concise feedback in time to all of the students. The popularity of massive open online courses (MOOCs) in programming education makes it even more challenging because MOOCs often involve more students than traditional courses [4]. Notably, it can be much simpler to provide test cases that faulty programs fail to pass. Although failed test cases can be very helpful for experienced developers in program debugging, they are often insufficient for students in programming courses [1]. To better help students with their assignments, more personalized, more specific, and more concise feedback is indispensable [5]. An intuitive way for providing feedback on faulty submissions is to generate patches with existing automated program repair (APR) techniques. However, experimental results [4] suggest that general APR techniques do not perform well on this task because bugs made by students are often different from those made by professional developers, and they cannot take advantage of the reference programs or multiple submissions that are available in the scenario of this task.

To release tutors from the manual construction of concise patches, a number of automated feedback generation approaches have been proposed and published in prestigious conferences on software engineering (such as FSE [6] and ASE [2]) and conferences on programming languages (such as PLDI [7]). The key idea of such approaches is to repair a faulty program based on correct reference programs and to provide the patches as feedback to students [5]. Such approaches can be static code copying approaches or dynamic repair approaches. The static approaches, e.g., the framework proposed by Zimmerman and Rupakheti [2], take as input a bug-free program (called reference program) for the same programming task, compare this reference program against the to-be-fixed submission, and generate a list of editions that can change the to-be-fixed submission into the reference program. Dynamic approaches, e.g., CLARA [1], take

- *Leping Li, Hui Liu, Kejun Li, and Yanjie Jiang are with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China. E-mail: {3120170473, liuhui08, likejun, jiangyanjie}@bit.edu.cn.*
- *Rui Sun is with Baidu, Inc., Beijing 100085, China. E-mail: sr19930829@163.com.*

Manuscript received 29 July 2021; revised 9 Feb. 2022; accepted 10 Feb. 2022. Date of publication 23 Feb. 2022; date of current version 9 Jan. 2023.
This work was supported by the National Natural Science Foundation of China under Grants 61690205 and 62172037.
(Corresponding author: Hui Liu.)
Recommended for acceptance by Y. Brun.
Digital Object Identifier no. 10.1109/TSE.2022.3153522

as input bug-free reference programs whose overall structures (i.e., control-flow structures) are identical to that of the to-be-fixed submission, decompose the reference programs and to-be-fixed submission into small blocks according to their control-flow structures, and generate editions that make each block of the to-be-fixed submission equivalent to its reference blocks.

We notice that both classes of research efforts rely heavily on bug-free reference programs, and the second category even requests such bug-free references to have identical control-flow structures as those of the to-be-fixed submissions. However, for students' programs submitted at the initial state of new tasks (i.e., newly released assignments), it is likely that there are no bug-free reference programs, let alone bug-free reference programs that have the same control-flow structures as the new submission. As a result, existing approaches frequently fail to generate useful patches for such submissions. Notably, new assignments are created and released frequently. For example, Codeforces [8] released 950 new programming tasks (composed of requirements and test suites) during 2019. New assignments are also released with emerging programming textbooks. New features of programming languages and frameworks also request tutors to create new assignments. We conclude, based on the preceding analysis, that we lack highly desirable approaches for generating concise patches for newly released programming assignments.

Therefore, in this paper, we propose a fully automated approach, called *AssignmentMender*, to construct concise patches on incorrect student attempts. The key insight is that both incorrect attempts and correct attempts can be leveraged for automated program repair. For a piece of faulty code within the faulty program, we may retrieve its reference code from reference programs (that are submitted for the same assignment) even if the structures of such programs are different from those of the faulty program. Based on the reference code, we can automatically fix the faulty statements and thus fix the faulty programs. The key step in the proposed approach is leveraging static analysis to retrieve such reference code with a graph-based matching algorithm. For a given student's faulty program and other (correct or incorrect) programs submitted for the same task, our approach (*AssignmentMender*) works as follows. First, *AssignmentMender* leverages an automated fault localization algorithm (AFL) to locate suspicious statements. Second, for each suspicious statement, *AssignmentMender* retrieves its reference code from reference programs and generates candidate patches by replacing AST nodes associated with the suspicious statements in the faulty program with corresponding AST nodes of the reference code. In addition, *AssignmentMender* leverages the widely used mutation operators and code copying (as a last resort) to generate candidate patches automatically. Finally, *AssignmentMender* validates such candidate patches via regression testing and outputs valid patch.

To evaluate the proposed approach, we select 10 programming assignments from *Codeforces*, and for each of the assignments, we download the 50 latest submissions. The proposed approach is then evaluated with the 128 faulty programs in the resulting dataset. Evaluation results show that the proposed approach improves the state-of-the-art in generating concise patches for assignments when only a small number of submissions for the same task are available. It successfully generates concise patches for 68 faulty programs, whereas the state-of-the-art approach concisely fixes 39 of them. A user case study involving 40 students and 80 submissions further provides initial evidence of the practical usefulness of the proposed approach.

This paper makes the following contributions:

- A fully automated approach to repair student programs by leveraging both correct and incorrect programs submitted for the same task. To the best of our knowledge, it is the first approach that can exploit faulty submissions in generating patches for programming assignments. It is also the first to exploit a graph-matching algorithm in retrieving reference code snippets from reference programs for automated program repair.
- An evaluation of the proposed approach on 128 programs, as well as a case study with 40 students. The evaluation provides evidence that the proposed approach can improve the state-of-the-art in providing concise patches for student programs when the number of available reference programs is small.

2 Related Work

2.1 Feedback Generation for Programming Assignments

To reduce the burden on tutors when giving patches as feedback to students on their programming assignments, researchers have proposed a number of approaches to construct feedback automatically or semiautomatically. *AutoGrader* [7] repairs faulty programs based on one reference program and error model prepared by tutors. First, the tutor is asked to provide a standard reference program for a programming task. After that, *AutoGrader* utilizes EML (error model language) to represent the repair history of faulty programs submitted for the same task and obtain an error model composed of a set of rewriting rules. Finally, among a large number of correct programs, *AutoGrader* utilizes a constraint-solving mechanism to search one reference program that (1) is semantically equivalent to the tutor's standard program and (2) indicates the smallest repair modification on the faulty program to change it into the selected reference program with rewriting rules. Shalini *et al.* [5] propose an approach specifically for dynamic programming (DP) assignments. First, the approach classifies students' programs into several clusters and requests tutors to provide reference (correct) programs for each of the resulting clusters. Second, for each program, the approach checks its equivalence with its reference program constructed by tutors. If they are not equivalent (i.e., the program is faulty), the approach divides the program and reference program into several parts and repairs each part of the faulty program based on statements in its corresponding part of the reference program. Notably, both of these early approaches [5], [7] request tutors to provide reference programs before patches can be generated automatically.

To release tutors from constructing reference programs, Zimmerman and Rupakheti [2] propose a static repair technique that selects one reference program among a set of bug-free programs submitted by other students. It parses the

faulty program and the selected reference program into AST trees and attempts to generate a sequence of editions on the AST tree of the faulty program. Such editions can eliminate the differences between the two AST trees and thus essentially turn the faulty program into the selected reference program. Different from the static approach proposed by Zimmerman and Rupakheti [2], Sumit *et al.* [1] proposed a dynamic repair technique, called *CLARA*, to repair faulty assignments. *CLARA* retrieves reference (correct) programs that share the same looping structure with the faulty program and clusters such reference programs based on dynamic variable values during execution. Based on each cluster of reference programs, *CLARA* repairs each block of the faulty program by replacing expressions (synthesis of modifications on one variable) in the faulty block with expressions (related to that variable) in corresponding blocks of reference programs in the cluster. Modifications on all blocks of the faulty program make up a candidate patch. According to the AST-tree edit distance, the smallest patch is selected. *SARFGEN* [3] proposed by Wang *et al.* is similar to *CLARA*. First, *SARFGEN* retrieves reference programs having the same control-flow (looping and conditional) structure as that of the faulty program. Second, for each of the retrieved reference programs, *SARFGEN* compares it against the faulty program and generates statement adding/replacing/deleting operations to eliminate (part or entire) differences. Such statement-level editions together compose a complete patch for the faulty program. *SARFGEN* verifies all potential patches and suggests the smallest validated patch. *Refactory* [9], proposed by Yang *et al.*, is another repair approach similar to *CLARA*. In addition to block-by-block repair as in CLARA, Refactory also utilizes a series of mutation operators to generate equivalent mutants of correct programs, which increases the success rate of finding a correct program with the identical control-flow structure as a buggy program under repair.

Notably, all such reference-based approaches [1], [2], [3], [9] rely heavily on bug-free reference programs. *CLARA* and *SARFGEN* even request such bug-free references to have identical control-flow structures as those of the to-be-fixed submissions. For newly released programming assignments, they may fail to provide such required references that would fail such approaches. Different from these approaches, our approach leverages all reference programs even if they are faulty. As a result, compared to existing approaches, our approach can improve the chance of success when the number of available reference programs is small and their quality is not guaranteed (e.g., the case of newly released assignments).

Refazer [10] proposed by Rolim *et al.* is different from the approaches introduced in the preceding paragraphs in that Refazer does not require reference programs. Refazer learns syntactic transformation patterns from assignment revision histories. For a given faulty assignment, it applies the resulting syntactic transformation pattern and validates the effect by regression testing. Although Refazer does not require correct reference programs, it frequently fails to fix faulty programs that require complex modifications [3].

2.2 Automated Fault Localization, Syntax-Based Clone Detection and Automated Program Repair

Our approach is also related to automated fault localization (AFL) [11] and generic automated program repair (APR) techniques [12], which provide candidate buggy points and candidate patches to software developers in practice.

AFL techniques analyze static or dynamic information and rank the program elements (classes, methods or statements) as a list based on the possibility of containing bugs. Fault localization is a necessary process before program repair. The most popular AFL technique used in program repair is spectrum-based fault localization (SBFL) [13], [14], [15], which collects the execution path of each passed/failed test case and calculates the buggy rate of each statement. The idea of such AFL technique is that the statements covered by more failed test cases and less passed test cases are more likely to be buggy.

Syntax-based clone detection [16] aims at identifying (syntactically or semantically) similar/identical source code fragments (snippets) based on syntax analysis. It parses the source code first and abstracts predefined static features. After that, it traverses the entire code dataset and filters out code snippets with similar features. Recent works [17], [18] utilize convolutional neural network in selecting valuable features (encoded from AST information). Such deep learning model results in high precision and recall. Our approach differs from syntax-based clone detection in the following aspects. First, our code searching process works at finer-grain, searching for small elements (e.g., statements and expressions) whereas clone detection often work on larger units, e.g., blocks, methods and classes. Second, most clone detection approaches exploit only local information whereas our approach exploits both local information (syntax feature of statements) and global information (control-flow structures).

APR approaches fall into four categories. The first category of APR approaches are mutation-based, such as *JAFF* [19] and GenProg [20]. *JAFF* employs six operators that can replace/insert/swap AST nodes. *GenProg* leverages genetic algorithm to generate new patches. Different from *JAFF*, which works on AST nodes, *GenProg* works on statements.

The second category of APR approaches are template-based, which generate patches according to templates (predefined or example-based). Long *et al.* propose *Prophet* [21], which utilizes the information from a large database of software revision changes. Xuan-Bach *et al.* propose history-driven repair [22] that utilizes twelve predefined templates. It searches for patches by collecting and mining existing patches across different programs. Xiong *et al.* proposed accurate condition synthesis (*ACS*) [23], an APR approach specifically designed for incorrect conditions. It repairs bugs by replacing an incorrect condition with a newly generated condition.

The third category of APR approaches are based on existing patches or similar code snippets. Xi *et al.* propose *ssFix* [24], which locates the faulty code first and searches for similar code snippets from external code database based on TF-IDF. After that, it repairs the bug by code transplantation. Jiang *et al.* propose *SimFix* [25], which is similar to *ssFix*. However, it utilizes both existing code and patches when repairing bugs. It identifies the difference between faulty code snippets and similar code snippets in other locations of the same project to obtain an expected modification. If a modification is considered frequent according to the existing patch database, it is used to generate new patches.

The fourth category of APR approaches are semantic-driven approaches, which analyze source code and encode repair problems formally. After that, the approaches attempt to solve the encoded problem. Yalin *et al.* propose *SearchRepair* [26], which exploits a database of human-written patches encoded as SMT formulas. For each potentially faulty code fragment, *SearchRepair* searches the database to find a change that might produce the desired input-output behavior. Mechtaev *et al.* propose *Angelix* [27], which can complete multiline repairs. It utilizes symbolic execution to obtain angelic paths and combines test cases and angelic paths to form an angelic forest, which is sent to a repair synthesis engine. After that, they propose *SEMGRAFT* [28], which utilizes a correct reference program t (t may be in a different language) to generate enough additional test cases. These additional test cases can help to solve the overfitting problem encountered by *Angelix*. Xuan-Bach *et al.* propose *S3* [29], which can replace potentially incorrect expressions with new expressions generated based on predefined components. For each faulty program, *S3* extracts input and desired output examples that describe passing behavior. After that, it synthesizes repairs that satisfy and generalize beyond the provided examples.

It is intuitive to provide feedback on programming assignments by applying generic APR approaches to faulty assignments. Therefore, Jooyong *et al.* perform a study [4] that utilizes four existing generic APR techniques in repairing students' programs. However, the repair rate of all the evaluated approaches in their study is much lower than the reported repair rate of programming-assignment-specific approaches (e.g., Refactory [9]). There are two major reasons for the low performance. First, the programming assignment submissions may be seriously incorrect and thus require complex modification. In contrast, bugs in software applications in the industry are often simpler, requiring only smaller changes to fix them. Second, general APR approaches cannot assume the existence of one or multiple (flawed or perfect) reference programs, so they cannot take advantage of them.

3 Motivating Example

Listing 1 presents a motivating example (noted as *FaultyPrg*) from Codeforces [8].

This is one of the submissions for programming task #271A whose requirements are presented as follows:

> "*You are suggested to solve the following problem: given a year number, find the minimum year number which is strictly larger than the given one and has only distinct digits.*
> *Input: The single line contains integer y* ($1000 \leq y \leq 9000$) *– the year number.*
> *Output: Print a single integer – the minimum year number that is strictly larger than y and all its digits are distinct. It is guaranteed that the answer exists.*"

The faulty program (as presented in Listing 1) works as follows. In the first line, it receives input from the console and stores it to variable n on line 2. The first nested iteration (lines 7-10) decomposes the number n (e.g., "1245") into four digits (e.g., '1', '2', '4', and '5'). The following nested iteration (lines 11-18) compares each digit against other digits on its right side (e.g., comparing '1' against '2', '4' and '5') and increases n if two digits are equivalent. The program breaks the WHILE loop if and only if n is composed of four distinct digits.

Listing 1. Faulty Submission (Program)

```
1  Scanner obj = new Scanner(System.in);
2  int n = obj.nextInt();
3  char c[] = new char[4];
4  boolean b=true;
5  while (b){
6  b=false;
7  for(int i=0; i<4; i++){
8  c[i] =
9  (Integer.toString(n)).charAt(i);
10 }
11 for (int j=0; j<3; j++){
12 for(int k=j+1; k<4; k++){
13 if(c[k] == c[j] && !b){
14 n=n+1;
15 b=true;
16 }
17 }
18 }
19 }
20 String s = new String(c);
21 System.out.println(s);
```

Listing 2. Reference Submission (Program)

```
1  Scanner scan = new Scanner(System.in);
2  int n = scan.nextInt()+1;
3  while(n<2019){
4  int temp = n;
5  Set<Integer> hs =new HashSet<>();
6  int count = 0;
7  while(n!=0){
8  hs.add(n%10);
9  n/=10;
10 count++;
11 }
12 n=temp;
13 if(hs.size()==count) break;
14 n++;
15 }
16 System.out.println(n);
```

A defect in the faulty program is that it would print the input as it is if the input (e.g., "1234") represents a year that is composed of four distinct digits. However, it is inconsistent with the requirement that states explicitly '*... find the minimal year number which is* **strictly larger than***the given one...*'. A concise and intuitive patch for the program is to replace *int n=obj.nextInt();* (line 2) with *int n=obj.nextInt()+1;*.

Another submission (noted as *refPrg*) for the same task is presented in Listing 2. Notably, this submission is faulty as well (the WHILE condition on line 3 should be '*true*' instead of '$n < 2019$'). State-of-the-art approaches, i.e., *CLARA* and *SARFGEN*, fail to repair the faulty program *FaultyPrg* based on this reference program (*refPrg*) because (1) such approaches exploit only correct reference programs and ignore buggy programs, e.g., *refPrg*, and (2) such approaches exploit only such reference programs whose control-flow structures are identical to those of the to-be-fixed program, whereas *refPrg* and *FaultyPrg* do not share the same structure. Another state-of-the-art approach, *Refactory*, can utilize

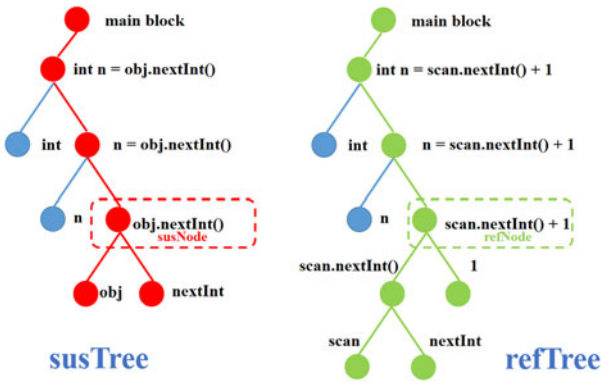

Fig. 1. (Partial) AST trees of a faulty code statement and its reference.

predefined mutation operators to construct a more correct reference program (equivalent mutant). However, as the structure of the reference program *refPrg* is essentially different from that of the faulty program *FaultyPrg* (e.g., number of iterations and presence of IF statements), *Refactory* cannot construct a correct reference program with the same control-flow structure. As a result, *Refactory* generates a patch that changes the faulty program *FaultyPrg* to its reference *refPrg*, which leads to a series of unnecessary modifications.

The proposed approach *AssignmentMender* repairs the faulty program successfully as follows.

- *AssignmentMender* employs the widely used spectrum-based fault localization algorithm (SBFL) [30] to locate suspicious statements. SBFL suggests that line 2 on Listing 1 (i.e., *int n=obj.nextInt();*) is suspicious.
- *AssignmentMender* compiles the faulty program and retrieves the resulting AST tree.
- *AssignmentMender* retrieves the AST subtree related to the suspicious statements (denoted as *susTree* as presented in the left part of Fig. 1).
- *AssignmentMender* compiles the reference program (*refPrg*, which is faulty as well), resulting in another AST tree (noted as *refTree* that is partially shown in the right part of Fig. 1).
- *AssignmentMender* enumerates the AST nodes associated with the suspicious statement and attempts to repair them with reference nodes retrieved from *refTree*. For node *susNode* in Fig. 1, *AssignmentMender* retrieves *refNode* in Fig. 1 as the reference AST node by the graph-matching method (see details in Section 4) because *susNode* and *refNode* have similarity in the AST structure (i.e., the red subgraph and green subgraph in Fig. 1 are highly similar). *refNode* represents the source code of *'scan.nextInt() + 1;'* (line 2 of *refPrg*).
- *AssignmentMender* replaces *susNode* with its reference node *refNode*. To keep the revised program syntactically correct, the replacement is augmented with an additional renaming: renaming the variable (*scan*) in *refNode* with the corresponding variable name (*obj*) in *susNode*. All such revisions constitute a candidate patch $cPatch$.
- *AssignmentMender* validates the candidate patch $cPatch$ automatically by running associated test cases and outputs the validated patch.

4 Approach

4.1 Overview

An overview of the proposed approach (*AssignmentMender*) is presented in Fig. 2. The approach consists of five parts: fault localization, mutation-based repair, reference-based repair, patch validation, and the last resort. For a given faulty program p and its reference programs $RP = \{rp_1, rp_2, \ldots, rp_n\}$ that are potentially faulty as well, *AssignmentMender* works as follows:

1) *AssignmentMender* locates suspicious statements $SST = < st_1, st_2, \ldots, st_m >$ in the faulty program p. The localization is accomplished by widely used spectrum-based fault localization algorithm (SBFL).
2) *AssignmentMender* employs the first strategy *mutation-based repair* to repair the faulty program by applying common mutation operators to the suspicious statements SST.
3) Based on the suspicious statements SST, *AssignmentMender* employs the second strategy *reference-based repair* to repair the faulty program p. The key to this strategy is to repair faulty code in p with their corresponding reference code in reference programs (RP) even if such reference programs are faulty.
4) If valid candidate patches have been generated thus far, the repair terminates successfully. Otherwise,

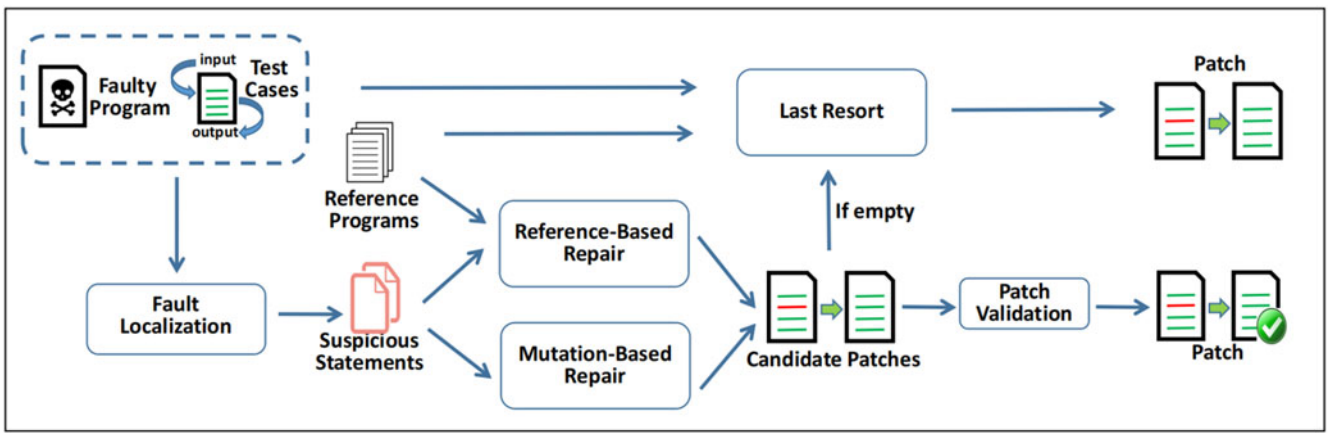

Fig. 2. Overview

5) *AssignmentMender* employs the third strategy as the last resort to repair the faulty program p. It retrieves bug-free reference programs (if there are any) from RP that have an identical control-flow structure as that of the faulty program p and attempts to make the faulty program equivalent to the reference programs except for the variable names. If the third strategy fails as well, *AssignmentMender* refuses to generate any patches for the given faulty program.

Details of the key steps are presented in the following sections.

4.2 Automatic Fault Localization

To make the following program repair strategies more effective and more efficient, we leverage the widely used spectrum-based fault localization algorithm (called SBFL) [30] to identify suspicious statements that may contain defects. SBFL identifies suspicious statements based on the execution result and path of test cases associated with the faulty program p. We utilize the *Jaccard* [14] formula for SBFL. For each statement s in p, SBFL calculates its suspicious score (likelihood of being faulty) as follows:

$$susScore(s) = \frac{|failed(s)|}{|executed(s)| + (|totalfailed| - |failed(s)|)}$$

where $|failed(s)|$ is the number of failed test cases that execute s, $|totalfailed|$ is the number of failed test cases, and $|executed(s)|$ is the number of test cases that execute s.

Based on the resulting suspicious scores, we collect a sequence of suspicious statements $SST(p) =< st_1, st_2, \ldots, st_m >$ that are sorted in descending order according to their suspicious scores, where st_i is more likely to contain defects than st_j if $i < j$. Statements whose suspicious scores are zero are excluded from further exploration for bug fixing to increase the efficiency of the proposed approach.

4.3 Mutation-Based Repair

Program mutation, which is originally proposed to evaluate the effectiveness of test cases [31], has been proven effective and efficient in repairing programs that can be repaired by simple editions, e.g., turning the operator '+' into '-' [32]. For mutation-based program repair, we leverage the following mutation operators to generate candidate patches (mutants):

- *Operator Replacement* (*OR*) [33]: Replacing a rational/arithmetic operator with a different rational/arithmetic operator.
- *Variable Name Replacement* (*VNR*) [34]: replacing a variable with a different but type-compatible variable within the scope.
- *Statement Deletion* (*SD*) [35]: Removing a statement from the program.

The mutation operators are selected because they can remedy the weakness of reference-based repair. For example, *statement deletion* is selected because reference-based repair (Section 4.4) cannot generate patches that delete statements from faulty programs.

Given a faulty program p to be fixed and a set of associated test cases $TS =< ts_1, ts_2 \ldots ts_m >$, *AssignmentMender* generates and validates mutants (candidate patches) with mutation operators introduced in the preceding paragraph as follows:

- First, it collects all of the rational/arithmetic operators $Ors = \{o_1, o_2, \ldots, o_n\}$ in suspicious statements $SST(p)$. For each operator $o_i \in Ors$, *AssignmentMender* replaces it with a different rational/arithmetic operator (selected randomly), which results in a mutant.
- Second, it collects all variables $vars = \{v_1, v_2, \ldots, v_m\}$ in suspicious statements $SST(p)$. For each variable $v_i \in vars$, *AssignmentMender* generates a mutant by replacing v_i with a different variable $v_j \in vars$ that is type compatible with v_i.
- Third, *AssignmentMender* generates mutants by deleting a randomly selected suspicious statement from the faulty program. All of the generated mutants are represented as $MTs = \{mt_1, mt_2 \ldots mt_k\}$.
- Fourth, *AssignmentMender* validates each of the mutants in MTs. A mutant mt_i represents a valid candidate patch if and only if mt_i successfully passes all of the test cases in TS.

4.4 Reference-Based Repair

4.4.1 Overview

Reference-based repair attempts to repair fine-grained suspicious statements by retrieving their reference statements from reference programs and replacing the suspicious statements with their references. The key insight is that for a fine-grained suspicious code snippet (e.g., a single statement), its fault-free reference code snippets can be retrieved from reference programs even if such reference programs are faulty.

An overview of the reference-based repair is presented in Algorithm 1. Its input includes a faulty program p and a set of reference programs RPs and the test cases associated with the faulty program $TS = \{ts_1, ts_2, \ldots, ts_m\}$. In the first step, *AssignmentMender* sorts suspicious statements within the faulty program (line 1) by the algorithm specified in Section 4.2, which results in a sorted list of suspicious statements $SST = \{st_1, st_2, \ldots, st_m\}$. Second, *AssignmentMender* attempts to repair the faulty program p by modifying the most suspicious statement $st_i \in SST$ (i.e., the top element in SST) with the help of reference programs RPs (lines 3-15). If it works, the automated program repair terminates successfully. Otherwise, *AssignmentMender* removes st_i from SST (line 16) and attempts to repair p by modifying the next most suspicious statement (i.e., the top element in the updated SST). *AssignmentMender* repeats the last two steps until p is fixed successfully (line 11) or SST is empty.

The remaining of this section focuses on the second step, i.e., how *AssignmentMender* repairs p by modifying the most suspicious statement $st_i \in SST$ with the help of a reference program rp_i. It works as follows:

1) First, *AssignmentMender* parses the suspicious statement st_i (line 4), which results in a set of AST nodes (noted as $Snodes$).
2) Second, for each node $sn \in Snodes$, *AssignmentMender* retrieves a list of reference nodes (noted as $Ref(sn)$) from the reference program rp_i (lines 5-7).

For each node $rn \in Ref(sn)$, *AssignmentMender* generates a candidate patch by replacing sn in the faulty program with rn (lines 8-9) and validates it by running associated test cases TS (line 10). Once any of the candidate patches is validated successfully, *AssignmentMender* returns the patch and terminates (line 11).

Notably, when there is more than one reference program, *AssignmentMender* conducts the process for each of the reference programs (line 6) until p is fixed successfully.

Algorithm 1. Reference-Based Repair

```
Input: p//faulty program under repair
RPs//a set of reference programs
TS// a set of test cases
Output: Patch//Generated patches
1  SST = SBFL(p, TS);
2  while SST ≠ ∅ do
3    st= TopElement(SST);
4    Snodes=Parse(st);
5    foreach sn in Snodes do
6      foreach rp in RPs do
7        Ref = RetrieveRefNode(sn, rp);
8        foreach rn in Ref do
9          Patch = GeneratePatch(p, rp, sn, rn);
10         if Patch passes TS then
11           return Patch;
12         end
13       end
14     end
15   end
16   SST.remove(st)
17 end
18 return ∅// fail
```

The following sections present more details of the key steps, i.e., retrieving reference nodes and patch generation.

4.4.2 Graph-Based Retrieval of Reference Nodes

Graph-based retrieval of reference nodes is the implementation of the method *RetrieveRefNode* that is invoked on line 7 of Algorithm 1. In this section, we explain how we retrieve the reference node from reference programs.

Definition 1. *A context graph of an AST node nd from program p (whose corresponding AST tree is noted as t) is a subgraph of t that consists of only nd and its ancestor/descender nodes as well as the edges connecting such nodes.*

Based on the definition, the context graph (noted as $cxtGraph(nd)$) of node nd contains ancestor nodes of nd such that it can describe structural information and location of the given node, e.g., whether the given node is within a looping/conditional structure. In addition, the context graph contains the descendant nodes such that it can describe the content of the AST node.

We illustrate how context graphs are constructed for given AST nodes with the example code in Listing 3. Fig. 3 is the AST tree (generated automatically by Eclipse) of the example code. For the suspicious AST node that is highlighted with a rounded rectangle in Fig. 3, its context graph is composed of all of the red nodes in the figure and the edges connecting them.

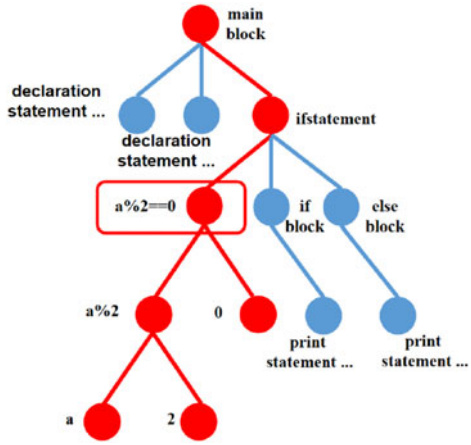

Fig. 3. Context graph of suspicious node.

Listing 3. Example Faulty Program

```
1 //faulty program:
2 Scanner obj = new Scanner(System.in);
3 int a = obj.nextInt();
4 if(a%2==0)//Expected condition:a%2==0&&a>2
5 {System.out.println("YES");}
6 else
7 {System.out.println("NO");}
```

Definition 2. *Two context graphs cg_1 and cg_2 are nearly isomorphic if and only if $\exists tree\, st_1 \subseteq cg_1, tree\, st_2 \subseteq cg_2 \Rightarrow (cg_1 - st_1) \cong (cg_2 - st_2) \wedge (st_1 = \emptyset \vee st_1 \cap leaves(cg_1) \neq \emptyset) \wedge (st_2 = \emptyset \vee st_2 \cap leaves(cg_2) \neq \emptyset)$, where $leaves(cg_1)$ and $leaves(cg_2)$ are the leaves of cg_1 and cg_2, respectively.*

According to the definition, two context graphs are nearly isomorphic if we can make them isomorphic (noted as $\cong$) by removing no more than one subtree from each of them, and the removed subtrees are empty or contain leaves of the context graphs. Such a subtree often represents subexpressions in source code statements. Notably, if a node is removed but some of its children are kept, such children are connected directly to their grandparent.

Algorithm 2 presents the algorithm to detect nearly isomorphic graphs. Lines 1-3 validate whether the two graphs $subtree(nd_1)$ and $subtree(nd_2)$ are absolutely isomorphic. If yes, they are nearly isomorphic, and the algorithm terminates. Notably, the validation can be accomplished by a deep-first traveling of the graphs (trees in fact) with a time complexity of $O(n)$, where n is the size of the graphs/trees. Lines 4-9 validate whether $subtree(nd_1)$ is a subgraph of $subtree(nd_2)$ or $subtree(nd_2)$ is a subgraph of $subtree(nd_1)$. Finally, if removing a subtree from each of the trees (Lines 11-12) makes the resulting trees absolutely isomorphic and the root nodes of the removed subtrees correspond to each other (Line 12), the two trees (graphs) are nearly isomorphic (Line 13). Notably, graph matching does not request dynamic execution of the programs; thus, its time complexity should be much smaller than that of the following execution-based patch validation.

Based on the definition of *nearly isomorphic*, we design the algorithm to retrieve reference nodes from a reference program rp for a given AST node (noted sn) associated with the suspicious statement in faulty program p. The detailed

algorithm is presented in Algorithm 3. We parse the reference program rp, which results in a set of AST nodes (line 3). For each of the resulting nodes, we construct its context graph (line 5) and compare it against the context graph of sn (line 6). Node rfn_i is considered a reference node (line 7) of sn if its context graph is *nearly isomorphic* with that of sn. After we retrieve all of the reference AST nodes, we sort them in ascending order (line 10) based on the size gap between sn and rfn_i.

Algorithm 2. Detecting Nearly Isomorphic Graphs

Input: $(cxtGraph(nd_1)$//ContextGraph of node1
$cxtGraph(nd_2)$//ContextGraph of node2
Output: $Bool$//true or false
1 **if** $cxtGraph(nd_1) \cong cxtGraph(nd_2)$ **then**
2 return true;
3 **end**
4 **if** $cxtGraph(nd_1) \subseteq cxtGraph(nd_2)$ **then**
5 return true;
6 **end**
7 **if** $cxtGraph(nd_2) \subseteq cxtGraph(nd_1)$ **then**
8 return true;
9 **end**
10 **foreach** sb_1 *in* $subtrees(cxtGraph(nd_1))$ **do**
11 **foreach** sb_2 *in* $subtrees(cxtGraph(nd_2))$ **do**
12 **if** $cxtGraph(nd_1) - sb_1 \cong cxtGraph(nd_2) - sb_2$ $\wedge\ Root(sb)_1 = \sigma\ (Root(sb_2))$ **then**
13 return true;
14 **end**
15 **end**
16 **end**
17 return false;

Algorithm 3. Retrieving Reference Nodes

Input: sn//AST node
rp_i//reference program
Output: RFN//reference nodes from rp_i
1 $RFN \leftarrow \emptyset$;
2 $cxtGraph_s$=ConstructCtxGraph(sn);
3 RFS_i = Parse(rp_i);
4 **for each** rfn_i *in* RFS_i **do**
5 $cxtGraph_r$=ConstructCtxGraph(rfn);
6 **if** *IsNealyIsomorphic*$(cxtGraph_s, cxtGraph_r)$ **then**
7 RFN.add(rfn_i)// a new reference node;
8 **end**
9 **end**
10 $RFN = Sort(RFN)$;//Sort based on the size gap between sn and rfn_i
11 **return** RFN;

Listing 4 presents a reference program of the faulty program in Listing 3, and its associated AST tree is presented in Fig. 4. From the tree, we attempt to retrieve the reference AST nodes for the suspicious node that is highlighted with a rounded rectangle in Fig. 3. Therefore, we compute the context graph for each of the nodes in Fig. 4 and compute whether the context graph is nearly isomorphic with that of the suspicious node. For the node highlighted with a rounded rectangle in Fig. 4, its context graph includes all green nodes and the edges connecting them. We also notice

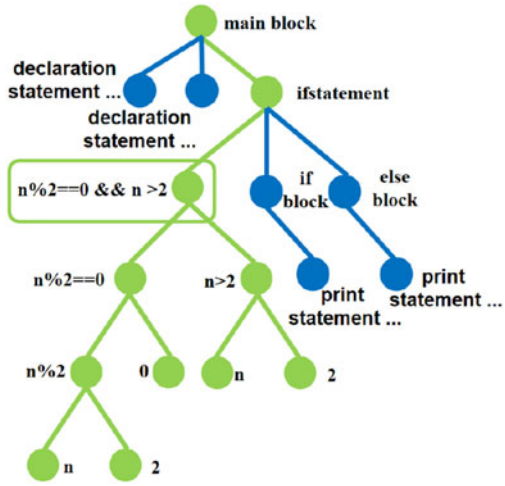

Fig. 4. Context graph of reference node.

that removing a subtree (i.e., the highlighted node and its right-hand side branch) from the context graph results in a new context graph that is identical to that of the suspicious AST node (i.e., the red subtree in Fig. 3). Consequently, the highlighted node is returned as a reference node.

Listing 4. Example Reference Program

```
1 //reference program:
2 Scanner obj = new Scanner(System.in);
3 int n = obj.nextInt();
4 if(n%2==0&&n>2)
5 {System.out.println("YES");}
6 else
7 {System.out.println("NO");}
```

4.4.3 Patch Generation

For a given AST node sn associated with a suspicious statement in faulty program p and its reference node rfn_i from a reference program, *AssignmentMender* generates a candidate patch by replacing sn with rfn_i. Notably, the replacement often results in compilation errors. The major reason for the compilation errors is that variables accessed by rfn_i may be undefined in the faulty program where rfn_i is inserted. For example, replacing $a\%2 == 0$ in Listing 3 (line 4) with its reference $n\%2 == 0\&\&n > 2$ in Listing 4 (line 4) results in a compilation error: *variable n is undefined*. To resolve the compilation errors, we replace undefined variables with those defined within the faulty program. Therefore, we build a mapping between variables in faulty program p and reference program rp_i. We compute the similarity for each pair of variables $< v_p, v_r >$, where v_p is defined in the faulty program and v_r is defined in the reference program. The similarity is computed as follows:

$$Sim(v_p, v_r) = f_t(v_p, v_r) \times (f_u(v_p, v_r) + f_n(v_p, v_r) + f_l(v_p, v_r)) \quad (1)$$

where function $f_t(v_p, v_r)$ indicates whether v_p and v_r are of the same type. $f_t(v_p, v_r) = 1$ if $type(v_p) = type(v_r)$. Otherwise, $f_t(v_p, v_r) = 0$. Similarly, function $f_n(v_p, v_r)$ indicates whether v_p and v_r share identical variable names. Function $f_l(v_p, v_r)$ concerns the order in which variables are defined, i.e., whether they are defined in the same position. For

example, if v_p is the first *String* variable defined in the faulty program and v_r is the first *String* variable defined in the reference program, we say they are defined in the same position and thus $f_l(v_p, v_r) = 1$. Function $f_u(v_p, v_r)$ represents the extent to which v_p and v_r are used in the same pattern. We represent the usage pattern of variable v_p as vector $Vuse(v_p) =< t_1, t_2 \dots t_n >$, where t_i specifies how many statements of type $StatementType_i$ access variable v_p. Based on the usage patterns, $f_u(v_p, v_r)$ is computed as follows:

$$f_u(v_p, v_r) = \frac{Vuse(v_p) \cdot Vuse(v_r)}{|Vuse(v_p)||Vuse(v_r)|} \quad (2)$$

4.5 Patch Validation

As mentioned in Section 4.1, for a given faulty program, *AssignmentMender* tries it by mutation-based repair and reference-based repair. To keep the integrity of the overall tree/program, *AssignmentMender* replaces AST nodes by traditional code transplantation technique. Besides that, the following compilation and test-based execution can filter out illegal candidates violating the integrity of the program. *AssignmentMender* replaces an AST node in suspicious statements with only matched AST nodes in reference programs, which significantly reduces the frequency of replacement and thus improves the efficiency of the approach. A generated candidate patch is valid if it makes the faulty program pass associated test cases successfully. *AssignmentMender* presents the valid patch as feedback to students.

4.6 The Last Resort

If both the mutation-based repair in Section 4.3 and the reference-based repair in Section 4.4 fail to fix a given buggy program, the last resort is to replace each block of the buggy program with a randomly selected bug-free reference program with identical structure. The replacement also involves variable mapping mentioned in section 4.4.3. Notably, the last resort is inspired by CLARA [1] that also utilizes coarse-grained (block) code search. The difference is that our last resort is based on static code transplantation only whereas CLARA utilizes dynamic clustering mechanism on reference programs and statement synthesis mechanism on each block of selected reference programs before code reuse.

5 Evaluation

5.1 Research Questions

The evaluation investigates the following research questions:

- *RQ1:* Does the proposed approach outperform the state-of-the-art approaches in generating concise patches for programming assignments when only a small number of reference programs are available?
- *RQ2:* How effective and efficient are the repair strategies employed by the proposed approach?
- *RQ3:* How does the number of available reference programs influence the performance of the proposed approach?

Research question RQ1 concerns the performance of the proposed approach (*AssignmentMender*) compared against the state-of-the-art approach. To answer this question, we compare *AssignmentMender* against *Refactory* [9] and CLARA [1]. *Refactory* [9] and CLARA [1] are selected because they are recently proposed and represent the state-of-the-art in fully automated feedback generation approaches for programming assignments. Another automated feedback generation technique, *SARFGEN*, is not selected because it is proposed earlier than *Refactory* [9], and its implementation is not publicly available. AutoGrader [7] is not selected for comparison because it requests an error model predefined by humans (teachers) that is not provided in our scenario. Besides, its implementation is not publicly available, which also prevents it from being selected for comparision. Refazer [10] is not selected for comparison because it requests hundreds of patches for the same program (called transformation that changes the state of the program from incorrect to correct) to learn fixing patterns. However, our research scenario does not provide such fixing histories of the buggy programs.

Research question RQ2 concerns the effect and efficiency of the employed repair strategies, i.e., mutation-based repair, reference-based repair and the last resort. By answering RQ2, we may discover the strengths and weaknesses of individual strategies and the collaboration of such strategies.

Research question RQ3 investigates how the performance of the proposed approach would be influenced by the number of available reference programs. More especially, it investigates whether the proposed approach is useful at the initial state of programming tasks where only a small number of reference programs are available. To answer this question, we increase the number of available reference programs (for each buggy program) from 50 to 99, and repeat the evaluation by reducing the number gradually from 99 to 0. The reduction of reference programs is accomplished by the widely used downsampling (undersampling) [3], [9].

5.2 Subject Applications and Experimental Process

To evaluate the proposed approach, we create a dataset by downloading 500 programs from *Codeforces* [8]. Codeforces continually releases a large number of simple programming tasks that are open to novice programmers, e.g., students majoring in computer science. Novice programmers submit a large number of programs to Codeforces, which results in a large code base. Although the submissions are often incorrect, Codeforces provides little feedback for programmers except for the failed test cases.

We select the 10 most popular programming tasks from Codeforces, which can be embedded into the executing framework of the compared approaches (*Refactory* and *CLARA*). The ten tasks contain various categories of programming tags (labeled by Codeforces website to classify tasks), e.g., *'brute force'*, *'math'*, and *'sorting'* . Links to the selected tasks are presented online [36]. For each task, we download its latest (in 2020) 50 submissions:

- Submitted by different developers;
- Compiled successfully and observed giving an (either correct or incorrect) output without crashing.

The resulting dataset (noted as $DATA$) is composed of 128 faulty programs and 372 functionally correct programs. Detailed information about faulty programs under repair is presented in Table 1.

For each of the 128 faulty programs in $DATA$, we employ *AssignmentMender*, *Refatory* and *CLARA* to generate patches.

TABLE 1
Distribution of Lines in Faulty Programs

Number of Lines	≤ 5	6-10	11-15	16-20	21 - 25	≥ 26
Number of Programs	24	60	31	7	5	1

Assume that the faulty program p is submitted for task t, and other submissions (in $DATA$) for the same task are $Subs4t = \{sb_1, sb_2, \ldots, sb_{49}\}$. *AssignmentMender*, *Refatory*, and *CLARA* generate patches for p as follows:

- First, we collect the submissions in $Subs4t = \{sb_1, sb_2, \ldots, sb_{49}\}$ and regard them as reference programs RP.
- Second, we apply *AssignmentMender* to p accompanied with reference programs RP.
- Finally, we apply *Refatory* and *CLARA* to p accompanied by reference programs RP. Please note that both *Refatory* and *CLARA* utilize correct submissions in $Subs4t$ only.

Our experiment is conducted on a 64-bit Windows server with one Intel(R) Core CPU and 16 GB RAM. We set the time limit for the repair of each bug to five minutes. If an approach fails to generate any valid patch in five minutes, we consider that it cannot generate patches in this case. To assess the quality of the generated patches, we compute the size of the generated patches.

In addition to the quantitative and objective measurements (i.e., correctness and size of patches), we also request two developers to manually assess the quality of the generated patches. The manual assessment considers unnecessary revision required by the patches. Such unnecessary revisions not only increase the patch size but also request students to revise some bug-free statements, which may confuse students.

Following the idea proposed by Sumit *et al.* [1], we classify patches into three categories: concise patches, overly fixing patches, and erasing patches. Assume that p is a faulty program of size $|p|$ and pt is a validated patch for p, pt is:

- a concise patch iff

$$\forall \text{ ValidPatch } pt' \Rightarrow |pt'| \geq |pt| \tag{3}$$

- an erasing patch iff

$$(|pt| > \frac{1}{2} \times |p|) \wedge (\exists \text{ValidPatch } pt' \Rightarrow |pt'| < |pt|) \tag{4}$$

- an overly fixing patch iff it is neither a concise patch nor an erasing patch.

where $|pt'|$ and $|pt|$ compute the size of the associated patches.

Concise patches are highly desirable because they do not contain any unnecessary modifications. In contrast, erasing patches are often useless because they contain overwhelming unnecessary modifications, and more than half of the statements in faulty program are modified. Such patches are common when repairing tools attempt to transfer the whole faulty program into an irrelevant reference program; they simply erase the original program (sometimes retaining a few common statements) and replace it with the reference program. Overly fixing patches are something in between. They contain some unnecessary changes but they modify only a small part of the faulty program.

TABLE 2
Comparison Against Existing Approach

	Fixed Programs	Concise Patches	Overly Fixing Patches	Erasing Patches
Assignment Mender	98	68	6	24
Refactory	94	39	18	37

5.3 RQ1: Outperform Existing Approaches

To answer research question RQ1, we compare the proposed approach against *Refactory* [9] and *CLARA* [1] on the 128 faulty submissions collected in the preceding section 5.2. Evaluation results of AssignmentMender and Refactory are presented on Table 2. Notably, *CLARA* results in errors/exceptions on 5 out of 10 tasks because some APIs (e.g., "split") within the involved programs cannot be resolved by CLARA.[1] A typical error message reported by CLARA is presented as follows: *Error occured: Exception "pyInterpreter' object has no attribute 'execute_split" on execution of 'split (input," ")'*. For fair comparison against CLARA, we remove such tasks where CLARA results in errors/exceptions, and compare the evaluated approaches on the remaining tasks (5 tasks, 84 buggy programs, and 166 correct programs) noted as *claraDATA*. Evaluation results on claraDATA are presented on Table 3.

The second column of Tables 2 and 3 presents the number of fixed programs. A faulty program is fixed if its revised version passes all of the associated test cases. The third column presents the number of programs fixed with concise patches. The fourth column represents the number of overly fixed programs (redundant patches). The last column presents the number of programs fixed with erasing patches.

From Tables 2 and 3, we make the following observations:

- The most valuable feedback generated by the employed approaches is the concise patch (as presented in the third row of Table 2) because concise patch gives students precise and useful guidance for revising the faulty program. In Table 2, we observe that compared to *Refactory*, *AssignmentMender* improves the number of concise patches by 74.4% = (68-39)/39.
- *AssignmentMender* fixes more programs than *Refactory*. Compared to *Refactory*, *AssignmentMender* fixes 4.3% = (98-94)/94 more programs.
- Most of the patches generated by *AssignmentMender* are concise, i.e., 69.4% = 68/98 of the patches generated by *AssignmentMender* are concise without unnecessary modifications. In contrast, only 41.5% = 39/94 of the patches generated by *Refactory* are concise.

1. It is likely that the exceptions are caused by the imperfect implementation (prototype for academic research) rather than the approach itself.

TABLE 3
Comparison Against CLARA (On claraDATA)

	Fixed Programs	Concise Patches	Overly Fixing Patches	Erasing Patches
Assignment Mender	68	58	5	5
Refactory	71	37	12	22
CLARA	61	30	19	12

- *Refactory* results in traceless fixing (where more than half of the statements in the original program are removed or revised) more frequently than *AssignmentMender*. 39.4% = 37/94 of the patches generated by *Refactory* are traceless fixing while only 24.5% = 24/98 of the patches generated by *AssignmentMender* are traceless fixing.
- As suggested by Table 3, *CLARA* results in 30 concise patches on claraDATA, substantially smaller than that generated by the proposed approach on the same dataset. The number (30) is close to that (37) generated by Refactory because both CLARA and Refactory exploit the block-based repair mechanism.
- We notice that all of the generated patches are correct (although some are not concise). None of the evaluated approaches generates plausible patches that pass all test cases but are manually confirmed as incorrect. This result differs from reported studies in traditional program repair [25] on the widely used Defects4J benchmark [37]. One possible reason is that the programming assignments in our evaluation are simple and equipped with many test cases.

For the most valuable feedback (concise patches), we manually analyze how the concise patches generated by different approaches overlap. We observe that *AssignmentMender* concisely fixes 34 faulty programs that *Refactory* fails to concisely fix. *Refactory* concisely fixes 5 faulty programs that *AssignmentMender* fails to concisely fix. 34 faulty programs are concisely fixed by both *AssignmentMender* and *Refactory*.

To further evaluate generated patches and objectively quantify their size, we also follow recent studies that report on patch size (PS) [1] and relative patch size (RPS) metrics [9]. The PS is the number of modified statements, and RPS is the edit distance between the buggy and fixed versions divided by the size of the original buggy version. We inspect the 67 buggy programs that are successfully fixed by both *AssignmentMender* and *Refactory* and compute the PS and RPS of each patch. The evaluation results are shown in Figs. 5 and 6, respectively. We make the following observations:

- First, both *AssignmentMender* and *Refactory* can generate high-quality patches that fix the buggy program by modifying no more than 1 statement. As Fig. 5 shows, *AssignmentMender* and *Refactory* generate 52 and 36 such concise patches, respectively.
- Second, *AssignmentMender* generates 266.7% = (2+20-6)/6 more patches in which RPS ≤ 0.2.
- Finally, the modes of PS/RPS values of *AssignmentMender* and *Refactory* are the same. Both approaches are more likely to generate patches in which PS ≤ 1 and RPS $\in (0.2, 0.3]$.

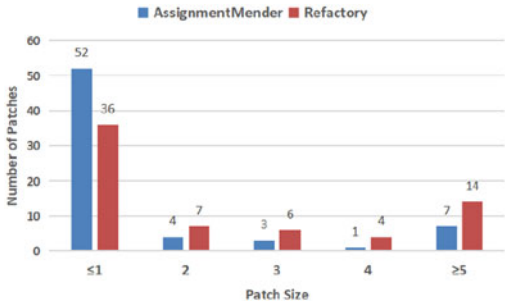

Fig. 5. Patch size distribution.

In conclusion, *AssignmentMender* generates more concise patches than *Refactory*.

To investigate the performance of the proposed approach (as well as Refactory) in repairing various faulty programs, we analyze all of the 128 faulty programs collected in Section 5.2, classify them into different categories according to various aspects, and compute the performance of the evaluated approaches on each of the categories.

First, we classify the faulty programs into three categories according to the scopes of their patches. The first category is associated with *single-statement* patches. Faulty programs of this category could be fixed by changing a single statement. The second category is associated with *single-block* patches. Faulty programs of this category could be fixed by modifying a single block (fragment) containing more than one statement. The third category is associated with *multiple-block* patches. Faulty programs of this category cannot be fixed without changing multiple blocks. The results are presented in Table 4. The first row presents different categories of bugs. The second row presents the number of faulty programs in each category. The third and fourth rows present the number of concisely repaired faulty programs (and repair rate in brackets) of AssignmentMender and Refactory, respectively. From Table 4, we observe that most of the faulty programs could be fixed by changing a single statement, accounting for 67.2% = 86/128. However, we also observe that around 22.7% = 29/128 of the faulty programs deserve complicated multiple block modification. The distribution of the patch scopes is similar to that reported by existing studies [1]. From Table 4, we observe that both AssignmentMender and Refactory are less effective in repairing complex bugs that could not be fixed by changing a single statement/block. Multiple-block faulty programs are hard to fix because existing approaches (including ours) are often designed to generate single-point patches (changes) only. Consequently, even if such faulty programs are fixed by APR approaches, it is likely that such

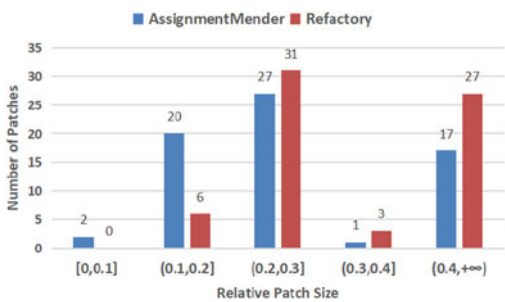

Fig. 6. Relative patch size distribution.

TABLE 4
Scopes of Patches

	Single Statement	Single Block	Multiple-Block
#Faulty Programs	86	13	29
#Fixed by AssignmentMender	61(70.9%)	6(46.2%)	1(3.4%)
#Fixed by Refactory	36(41.9%)	2(15.4%)	1(3.4%)

TABLE 5
Types of Faulty Statements in Single-Faulty-Statement Programs

	IF-Condition	Assignment and MethodInvocation	Output Statement
# Single-Faulty-Statement Programs	50	17	19
#Fixed by AssignmentMender	42(84.0%)	5(29.4%)	14(73.7%)
#Fixed by Refactory	29(58.0%)	1(5.9%)	6(31.6%)

ARP approaches have replaced a large block of the program covering all of the faulty blocks as well as bug-free code between the faulty blocks. As a result, the fixing is not concise.

Second, we further divide the *single-faulty-statement* programs (i.e., faulty programs of the first category introduced in the preceding paragraph) according to the statement type of the faulty statements within the programs. Results are presented on Table 5. The first row specifies the types of faulty statements within the single-faulty-statement programs. The second row specifies the number of faulty programs of the given types. The last two rows specify how many of the faulty statements have been concisely fixed by AssignmentMentor and Refactory, respectively. From this table, we observe that faulty If-conditions are common (account for $58.1\% = 50/86$ of the single-faulty-statement programs). It is consistent with existing studies that report similar findings [38]. We also notice that both AssignmentMender and Refactory result in high concise repair rate in repairing faulty If-conditions. One possible reason for the success is that If-conditions are closely related to control flow structures and thus it is easier for AssignmentMender and Refactory to retrieve reference statements from reference programs. Notably, because of the limited number of the involved buggy programs, Table 5 does not cover all possible statement types, e.g., looping-conditions. This is one of the limitations of the involved dataset. However, according to Liu's empirical study [39], the involved statement types in Table 5 do account for the majority (87.8%) of buggy statements in real-world bugs.

As suggested by Table 2, compared to *Refactory*, *AssignmentMender* generates more concise patches. We take a typical example in Listing 5 to illustrate how *AssignmentMender* avoids redundant (unnecessary) modifications. In the faulty program, the student types in an incorrect expression '*System.out.println(2-r+Math.abs(2-c))*' on line 7 by mistake (it should be '*System.out.println(Math.abs(2-r)+Math.abs(2-c))*'. *AssignmentMender* successfully generates a concise patch based on the reference AST node associated with the reference statement (line 18 in Listing 5). *Refactory*, however, replaces the whole faulty program with the reference program; thus, it introduces redundant modifications that change the type and declared position of variable *nums[ ]*. *Refactory* makes such unnecessary modification because it requires variables in the repaired version to be declared and have the same dynamic values in the corresponding block as they are in the reference program. Simply modifying the incorrect expression in line 7 as *AssignmentMender* does would not satisfy this requirement.

Listing 5. Concise Repair Versus Redundant Modification Repair

```
1  //faulty program:
2  Scanner scan = new Scanner(System.in);
3  for(int r=0;r<5;r++) {
4  String[ ] nums = scan.nextLine().split(" ");
5  for(int c=0;c<5;c++) {
6  if (nums[c].equals("1")){
7  System.out.println(2−r+Math.abs(2−c));//the expression is
       incorrect
8  return;}
9  }
10 }
11 //Reference program utilized by AssignmentMender
12 Scanner scan = new Scanner(System.in);
13 String[ ][ ] arr = new String[5][5];
14 for(int i=0;i<5;i++) {
15 arr[i] = scan.nextLine().split(" ");
16 for(int j=0;j<5;j++) {
17 if (arr[i][j].equals("1")) {
18 System.out.println(Math.abs(2 − i) + Math.abs(2 − j)); //
       expected expression
19 break;}
20 }
21 }
22 //AssignmentMender changes "2−r" into "Math.abs(2−r)"
```

Another advantage of the proposed approach is that it could exploit faulty reference programs (besides correct ones) to generate concise patches. To quantitatively evaluate the effect of faulty reference programs, we trace the generated concise patches to their original reference programs, and check whether the original programs are faulty. Our evaluation results suggest that out of the 68 concise patches generated by the proposed approach, 23 originated from faulty reference programs. It may suggest that faulty reference programs contribute significantly to the success of the proposed approach. We take the example in Listing 6 to illustrate how the proposed approach leverages faulty references. Faulty program A contains incorrect literals '*aeiou*' (should be '*aeiouy*') on Line 5. Although the reference program (Lines 16-32) is faulty, it does contain the requested literals ('*aeiouy*' on Line 20) to replace the faulty literals on Line 5. Notably, the control-flow structure of the reference program differs from that of the faulty program, and thus neither *Refactory* nor *CLARA* could utilize the reference program to fix the faulty one. In contrast, AssignmentMender is able to retrieve the expected AST node for the buggy one because the buggy statement (Line 5) and reference statement (Line 20) have identical AST subtree structure and neither of them is included by any looping/conditional structure. Consequently, AssignmentMender generates a candidate

patch by replacing the buggy statement with reference statement, which successfully fixes the buggy program.

We also notice that utilizing reference programs with faults or different control flow structures enables AssignmentMender to concisely repair hard-to-find bugs. We take the example in Listing 7. The faulty program contains incorrect input statement *'{scan.nextLine()}'* (should be *'scan.nextLine().split("")'*) on Line 5. Although the reference program contains the requested expression (*'scan.nextLine().split("")'* on Line 21) to repair the faulty statement on Line 5, neither *Refactory* nor *CLARA* could utilize the reference program because the faulty program and the reference program have different control flow structures. As a result, such block-based repairing mechanism (i.e., *Refactory* and *CLARA*) replaces the whole program because the faulty statement is directly contained by the main block. Notably, it is also challenging for students to spot the faulty statement because the statement is nearly correct (i.e., requesting minor changes only). The proposed approach successfully fixes the buggy program by utilizing fine-grained repair mechanism, and thus it has the potential to retrieve the reference statement (because both faulty statement and the reference statement are directly contained by the main blocks) and to generate concise patch.

Listing 6. Fixing a Buggy Program by Exploiting a Faulty Reference

```
1  //Faulty program A:
2  Scanner scan = new Scanner(System.in);
3  String s = scan.nextLine();
4  s = s.toLowerCase();
5  String v="aeiou";
6  //AssignmentMender changes String v="aeiou" into String v="
       aeiouy"
7  String temp ="";
8  for(int i=0;i<s.length();i++) {
9          if(!v.contains(s.substring(i,i+1))) {
10         temp+=s.substring(i,i+1);
11         }
12 }
13 String a = "."+String.join(".",temp.split(""));
14 System.out.println(a);
15
16 //Faulty reference program utilized by AssignmentMender
17 Scanner scan = new Scanner(System.in);
18 String s = scan.nextLine();
19 s = s.toLowerCase();
20 String v="aeiouy";
21 int i=0;
22 while(i<s.length()) {
23         if(v.contains(s.substring(i,i+1))) {
24         s=s.replaceFirst(s.substring(i,i+1),"");
25         }
26         else {
27                 String x='.'+s.substring(i,i+1);
28                 s=s.replaceFirst(s.substring(i,i+1),x);
29                 i+=2;
30                 }
31 }
32 System.out.println(s);
```

We conclude from the preceding analysis that when there are only a small number of reference programs available, the proposed approach has a greater chance of concisely repairing faulty programs than the existing approach. It is also highly complementary to the existing approach, and thus, it may work as a useful complement to them.

To evaluate the practicability of the proposed approach, we also record the execution time of the evaluated approaches in fixing programs during the previous evaluation. The results suggest that on average, it takes *AssignmentMender* (with a generate-and-validate mechanism) , *Refactory* and *CLARA* (without a generate-and-validate mechanism) 18.2, 5.4 and 6.3 seconds, respectively, to fix a buggy program. Thus, the time cost is acceptable when the number of references is limited to 50. However, if the number of reference programs increases greatly, we set an upper limit for the execution time.

Listing 7. Discovering and Eliminating Hard-to-Find Bug

```
1  //Faulty program:
2   Scanner scan = new Scanner(System.in);
3  int n = scan.nextInt();
4  scan.nextLine();
5  String[] lst = {scan.nextLine()}; //Faulty statement
6  int left = 0;
7  int right = lst.length − 1;
8  int count = 0;
9  for(int i=0;i<lst.length;i++) {
10         if (left < right) {
11                 if (lst[left].equals(lst[left + 1])) {
12                         count += 1;}}
13         left += 1;
14 }
15 System.out.println (count);
16
17 //Reference program with different control flow structure
       utilized by AssignmentMender
18 Scanner scan = new Scanner(System.in);
19 int n = scan.nextInt();
20 scan.nextLine();
21 String[] x= scan.nextLine().split("");//reference statement
22 int c=0;
23 for (int i=0;i<n−1;i++) {
24         if(x[i].equals(x[i+1]))
25                 c=c+1;
26 }
27 System.out.println(c);
```

5.4 RQ2: Effectiveness of Each Repair Strategy

The proposed approach consists of three strategies: mutation-based repair (MTR), reference-based repair (RR), and the last resort (LR). To investigate the quantitative effect of the three strategies, we keep only one of them at a time and repeat the evaluation. Evaluation results are presented in Fig. 7. We make the following observations:

- First, all three repair strategies are effective and generate some concise patches. MTR (mutation-based repair) alone generates concise patches for 22.1% = 15/68 of the faulty programs; RR (reference-based repair) alone generates concise patches for 80.9% = 55/68 of the faulty programs; and LR alone generates concise patches for 48.4% = 33/68 of the faulty programs.
- Second, compared to the other two strategies, MTR results in the lowest performance regarding the number of concisely fixed programs.
- Third, RR results in the largest number of concise patches. In addition, RR concisely fixes the largest number of bugs (22), which other strategies fail to fix. This may suggest that RR is the essential contribution to the proposed approach.

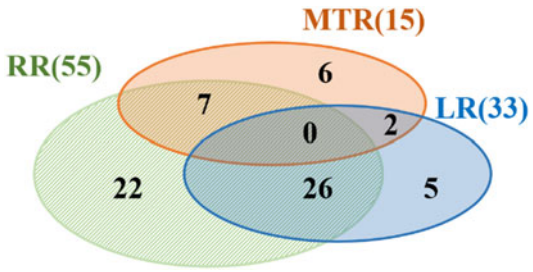

Fig. 7. Concise patches generated by each repair strategy.

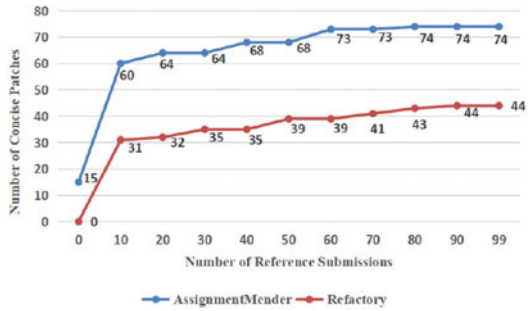

Fig. 8. Concise patches generated by each approach with different numbers of reference submissions.

We conclude from the preceding analysis that all of the employed strategies are indispensable.

5.5 RQ3: Reducing/Rising the Number of Available Reference Programs

For newly released programming tasks, there might be only a limited number of (correct or incorrect) reference programs to be exploited for program repair. To investigate the usefulness of the proposed approach under this scenario, we further expand our dataset in Section 5.2 from 500 to 1000 by selecting another 500 programs for the given ten tasks, following the same selection rules specified in Section 5.2. More specifically, for each of the ten tasks, we selected the latest 51-100th submissions from different developers in 2020 that compile successfully and give (correct or incorrect) output value without crashing. Notably, the latest 50 submissions have already been added to the dataset in Section 5.2, and thus now the dataset contains the latest 100 (instead of 50) submissions to the task. After that, we repeat the evaluation, gradually reducing the number of available reference programs from 99 to 0.

To reduce the number of reference programs, we order the submissions by time and select the earliest submissions only. Evaluation results are shown in Fig. 8. The horizontal axis specifies how many submissions are leveraged as reference programs, and the vertical axis specifies the number of concise patches generated by the evaluated approaches.

In Fig. 8, we observe that the performance of both approaches is influenced by the number of available reference submissions. The number of concise patches generated by the proposed approach increases from 15 to 60 when the number of reference programs increases from 0 to 10. After that, the number of concise patches increases slightly from 60 to 68 when the number of references increased from 10 to 50, and further increases to 74 when the number of references increased to 99. From the figure, we also observe that the proposed approach outperforms the baseline regardless of the number of available reference programs.

5.6 Threats to Validity

A threat to construction validity is that manual validation involved in the evaluation is potentially biased. It heavily weights the results in favor of the proposed approach if the participants know it in advance. To reduce the threat, we exclude the authors of the paper from the manual validation and recruit students who are not aware of the proposed approach for the evaluation.

A threat to internal validity is the selection of mutation operators. The selection of the mutation operators may significantly bias the conclusions. As introduced in Section 4, the proposed approach reuses mutation operators employed by existing approaches [34] where the selected mutation operators have been well justified. Notably, when bugs are more complicated than, or different from predefined mutation operators, the proposed approach would employ the reference-based repair technique (as introduced in Section 4.4) or the last resort (as introduced in Section 4.6) instead of the mutation-based repair.

A threat to external validity is that only a small number of programming tasks and assignments are involved in the evaluation. We only select 10 assignments and 128 faulty submissions for the evaluation because of the cost of manually checking/validating candidate patches. Special characteristics of such submissions may have biased the evaluation results, and thus, the conclusions drawn on such submissions may not hold on other submissions. To the best of our knowledge, the baseline approaches, CLARA [1] and Refactory [9], were evaluated on 6 and 5 programming tasks by their authors, respectively. The limited number of tasks is usually caused by the extensive human intervention requested by the evaluation.

It may be highly valuable to further evaluate the proposed approach on more students' submissions for various tasks in programming courses, e.g., those collected by Sumit *et al.* [1] and Wang *et al.* [3]. However, to the best of our knowledge, such submissions [1], [3] are not publicly available. Although the students' submissions collected by Hu *et al.* [9] are publicly available, such submissions are in Python and thus cannot be used in the proposed approach. Notably, the Codeforces website (where we retrieved evaluation data) is a platform for both introductory programming education and programming competition. We only select the easier programming tasks (with the lowest A-level difficulty) that are specially designed for introductory programming education. As a result, the collected submissions are often simple and short, and the average size (10 LOC) of the submissions is close to that (11 LOC) of Sumit's dataset [1].

6 Case Study

6.1 Setup

In this section, we conduct a case study to investigate the usefulness of the proposed approach. More specifically, the case study investigates the following research question:

- Does *AssignmentMender* outperform existing approaches concerning the usefulness of generated patches?
- Are the patches generated by *AssignmentMender* useful for students in understanding the bugs in their faulty assignments?

We recruit 40 students from the Beijing Institute of Technology for the evaluation. All of them are majoring in computer science, and none of them has any industry experience. Before the case study, we briefly introduce our feedback generation system to the participants. The introduction focuses on external functionality but not on the underlying rationale of the feedback generation approaches.

For the case study, we select the 10 tasks collected in Section 5.2. Each of the selected tasks contains some example test cases to specify the interface of the expected submission. Each task is assigned to exactly 8 participants, and each of the participants is assigned exactly 2 tasks. Once we collect the submissions from participants, we validate the correctness of such submissions with predefined test cases. All faulty submissions (programs) are fed into our feedback generation system. Notably, the system is composed of two feedback generation approaches: *AssignmentMender* (the proposed approach) and *Refactory*. Such approaches generate feedback independently on each of the faulty programs, and thus, it is likely that the system may generate up to two feedback patches for a single submission. To simulate the scenario of newly released programming assignments, we provide the submissions for each assignment (collected in Section 5.2) to the evaluated approaches as reference programs.

We present the patches generated for faulty program *bp* to the author of the program (*bp*) and request her/him to evaluate the quality of feedback patches concerning their usefulness. The evaluation follows a Likert-type scale, varying from 1 (not at all, indicating absolutely useless) to 5 (strongly agree, indicating extremely useful). Notably, each patch is evaluated by a single participant, i.e., the author of the associated faulty submission. Patches generated by different approaches for the same submission are evaluated by the same participant.

6.2 Results and Analysis

In total, we receive 80 submissions from the participants. Among these submissions, 42 are bug-free and thus are not involved in the following patches evaluation. We also notice that 10 submissions contain compilation errors that are out of the field of the proposed approach (and baseline approaches as well); thus, they are also excluded from the following patches evaluation.

We generate patches for the remaining 28=80-42-10 faulty submissions with feedback generation systems. The proposed approach (*AssignmentMender*) successfully generates patches for 15 of the submissions. *Refactory* generates patches for 16 of the submissions. The rating of such patches is summarized in Table 6.

From this table, we make the following observations:

- First, both *AssignmentMender* and *Refactory* successfully generate some useful patches.

TABLE 6
Usefulness of Generated Patches

	Assignment Mender	Refactory
Extremely useful	6	3
Useful	7	5
Neutral	2	3
Useless	0	4
Absolutely useless	0	1
Total	15	16

- Second, *AssignmentMender* generates more highly useful patches (rated as '*extremely useful*') than *Refactory*, as suggested by the second row of the table.
- Third, compared to *Refactory*, *AssignmentMender* has a greater chance of generating useful patches; 86.7% = (6+7)/15 of its patches are extremely useful or useful. However, the probability is reduced to 50% = (3+5)/16 for *Refactory*. One possible reason is that patches generated by *AssignmentMender* are often more concise than those generated by *Refactory* (as specified in Section 5.3).
- Overall, according to Table 6, we calculate the mean value and variance in patch scores for *AssignmentMender* and *Refactory*. The results show that the mean values of *AssignmentMender*'s and *Refactory*'s patch scores are 4.27 and 3.31, and the variance values are 0.46 and 1.46, respectively. We conduct a Wilcoxon rank-sum test [40] whose result confirms that the AssignmentMender patches receive higher scores than those of Refactory with statistical significance ($p = 0.034 < 0.05$).

For each patch recommended by *AssignmentMender* or *Refactory*, we request the receiver (student) to indicate whether the generated patches actually help them determine the root causes of bugs. In total, we receive feedback for all 31 recommended patches. By analyzing such feedback, we make the following observations:

- Most (87% = 13/15) of the 15 participants receiving *AssignmentMender*'s patches suggest that the patches explicitly and accurately point to the buggy statements, which helps them figure out the root causes of the bugs. Others (13% = 2/15) suggest that the patches are helpful in figuring out the root causes of the bugs, but they fail to accurately point to the buggy statements, e.g., involve some irrelevant statements.
- Half of the 16 participants receiving *Refactory*'s patches suggest that the patches explicitly and accurately point to the buggy statements. Three of them suggest that the patches fail to accurately point to the buggy statements, but they are still valuable in determining the root causes of the bugs. Others (five out of sixteen) suggest that the generated patches are useless for determining the root causes of the bugs because such patches replace the whole to-be-fixed program.

Based on the preceding analysis, we conclude that the patches generated by our approach are often helpful, and it outperforms the baseline approach concerning the usefulness of generated patches.

6.3 Threats to Validity

A threat to internal validity is that the manual rating may be biased. To reduce the bias, the patches generated by different approaches are presented to participants anonymously; we do not provide the name (or any other clues) of the approach that generates a specific patch.

A threat to external validity is the limited size of the case study, including the number of participants, the number of programming tasks, and the number of faulty submissions. To reduce the threat, we will conduct a large-scale case study in the future, involving more tasks and more participants.

7 Discussion

The first limitation is that AssignmentMender may need much time to search for reference statements when programs get longer. Notably, the time complexity in pinpointing the reference statement is linearly dependent on the size of the reference program. Considering that students' submissions to programming assignments are often small, the time complexity in pinpointing the reference statements may not serve as a bottleneck in the approach. According to our execution log of AssignmentMender, compiling and verifying a generated patch takes around 1 second, which serves as the most time-consuming step of the approach. In contrast, searching for a reference node from reference programs is less time-consuming, taking 0.001 seconds on average. However, if the candidate patch is invalid, the proposed approach would continue generating and validating the next patch, and on average it took the approach around 18 seconds to repair an assignment during our evaluation.

The second limitation is that inaccurate automated fault localization (AFL) may influence the performance of automated program repair (APR) [41] because the latter depends on the former. Notably, students' submissions for programming tasks are often rather small. Consequently, even if the faulty statements are not top-ranked by AFLs, APRs still have the potential to reach and fix the faulty statements because the total number of suspicious statements for a small program is often small. Notably, if the assignments are incorrect because of violating functional or nonfunctional requirements of the assignment instead of buggy implementation (e.g., misused APIs), existing fault localization algorithms may fail and thus the proposed approach may not work.

The third limitation is that the proposed approach may fail to concisely fix programs with multiple buggy statements. The proposed approach has the potential to fix multiple consecutive statements with a single reference AST node from a single reference program. However, it cannot fix a buggy statement with a reference AST node from program A and then fix another buggy statement with another reference AST node from program B. To the best of our knowledge, the baseline approach (e.g., Refactory) suffers from the same limitation.

The fourth limitation is that the proposed approach may not work in the initial phase of a programming task if the number of available correct solutions is small (or no correct solutions). Therefore, the proposed approach searches both correct solutions and partially correct solutions to increase the chance of success. Furthermore, it leverages mutation operators that do not depend on reference solutions.

The fifth limitation is that *AssignmentMender* judges the correctness of the repaired program based on test cases. Thus, it may generate plausible patches that are not correct but can pass all test cases because of the risk that some plausible reference programs might mislead *AssignmentMender*. In our experiment, all of the generated patches are correct. One possible reason is that those student programming assignments are much simpler than real word programs, and the associated test cases published on *Codeforces* are enough to characterize the specification of expected functions.

The sixth limitation is that *AssignmentMender* cannot yet generate textual description or suggestions about error reason. Patches alone may not enough for students to understand the errors, and textual description and suggestions about the errors could be more helpful than patches. However, the proposed approach, and the baseline approaches as well, cannot yet generate such textual description or suggestions. The concise patches generated by the proposed approach (or the baseline approaches), in fact, serve as a guidance for fixing the issues, specifying where and how to change the source code, without any human-like explanation.

8 Conclusion and Future Work

It is highly valuable to generate concise patches automatically for faulty submissions to programming assignments. Although existing approaches have achieved a high success rate in the generation of useful feedback, they often depend on diverse and correct reference programs. For submissions in the early state of new assignments, such approaches frequently fail to generate concise and effective patches. Therefore, in this paper, we propose a new approach, called *AssignmentMender*. The key idea of *AssignmentMender* is to exploit code from reference programs regardless of their correctness. Evaluation results on 128 faulty submissions suggest that *AssignmentMender* outperforms the state-of-the-art approach in feedback generation when only a small number of reference programs are available. We also conduct a case study, and the results of our case study provide initial evidence of the usefulness of patches generated by *AssignmentMender*. To facilitate replication, we make the replication package publicly available at [36].

In the future, we would like to investigate how to prevent nonsense patches. According to our experience, it is often nonsense for students to replace the whole faulty program with a completely different (correct) program. However, we have not yet discovered quantitative conditions that can accurately filter out nonsense patches. Furthermore, it is unknown yet whether (or when) such nonsense patch is better than no patches at all. In addition, we have not yet attempted to fix buggy programs with half-finished programs as references. We believe that providing hints for students' partially finished programs is also helpful. Finally, we plan to validate the proposed approach with a large-scale case study in the future and extend it to other programming languages.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments and constructive suggestions.

REFERENCES

[1] 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.

[2] K. Zimmerman and C. R. Rupakheti, "An automated framework for recommending program elements to novices (N)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2015, pp. 283–288.

[3] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: Data-driven feedback generation for introductory programming exercises," in *Proc. 39th ACM SIGPLAN Conf. Prog. Lang. Des. Implementation*, 2018, pp. 481–495.

[4] 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 *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 740–751.

[5] S. Kaleeswaran, A. Santhiar, A. Kanade, and S. Gulwani, "Semi-supervised verified feedback generation," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 739–750.

[6] S. Gulwani, I. Radiček, and F. Zuleger, "Feedback generation for performance problems in introductory programming assignments," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 41–51.

[7] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *Proc. 34th ACM SIGPLAN Conf. Prog. Lang. Des. Implementation*, 2013, pp. 15–26.

[8] *CodeForces*, 2019. [Online]. Available: https://codeforces.com/

[9] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, and A. Roychoudhury, "Re-factoring based program repair applied to programming assignments," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2019, pp. 388–398.

[10] R. Rolim *et al.*, "Learning syntactic program transformations from examples," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 404–415.

[11] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

[12] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 17:1–17:24, Jan. 2018.

[13] R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proc. 12th Pacific Rim Int. Symp. Dependable Comput.*, 2006, pp. 39–46.

[14] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2002, pp. 595–604.

[15] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proc. 24th Int. Conf. Softw. Eng.*, 2002, pp. 467–477.

[16] A. Sheneamer and J. Kalita, "A survey of software clone detection techniques," *Int. J. Comput. Appl.*, vol. 137, pp. 1–21, Mar. 2016.

[17] H.-H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proc. 26th Int. Joint Conf. Artif. Intell.*, 2017, pp. 3034–3040..

[18] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 87–98.

[19] A. Arcuri, "On the automation of fixing software bugs," in *Proc. Companion Int. Conf. Softw. Eng.*, 2008, pp. 1003–1006.

[20] W. Weimer, T. V. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, 2009, pp. 364–374.

[21] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proc. 43rd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Prog. Lang.*, 2016, pp. 298–312.

[22] X. B. D. Le, D. Lo, and C. L. Goues, "History driven program repair," in *Proc. IEEE Int. Conf. Softw. Anal.*, 2016, pp. 213–224.

[23] Y. Xiong *et al.*, "Precise condition synthesis for program repair," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.*, 2017, pp. 416–426.

[24] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 660–670.

[25] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2018, pp. 298–309.

[26] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun, "Repairing programs with semantic code search (T)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2015, pp. 295–306.

[27] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, 2016, pp. 691–701.

[28] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.*, 2018, pp. 129–139.

[29] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: Syntax- and semantic-guided repair synthesis via programming by examples," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 593–604.

[30] A. Rui, P. Zoeteweij, and A. J. C. V. Gemund, "Spectrum-based multiple fault localization," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2009, pp. 88–99.

[31] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep./Oct. 2011.

[32] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Proc. 3rd Int. Conf. Softw. Testing*, 2010, pp. 65–74.

[33] F. Y. Assiri and J. M. Bieman, "An assessment of the quality of automated program operator repair," in *Proc. IEEE Int. Conf. Softw. Testing*, 2014, pp. 273–282.

[34] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Proc. 13th Annu. Conf. Genet. Evol. Comput.*, 2011, pp. 1427–1434.

[35] R. Kou, Y. Higo, and S. Kusumoto, "A capable crossover technique on automatic program repair," in *Proc. Int. Workshop Empirical Softw. Eng. Pract.*, 2016, pp. 45–50.

[36] *Links to Tasks*. Accessed: Jan. 23, 2022. [Online]. Available: https://github.com/CoPaGe/FeedbackExperimentTask

[37] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 437–440.

[38] J. Xuan *et al.*, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 34–55, Jan. 2017.

[39] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandandé, and Y. Le Traon, "A closer look at real-world patches," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2018, pp. 275–286.

[40] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 428–439.

[41] K. Liu *et al.*, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *Proc. 42nd Int. Conf. Softw. Eng.*, 2020, pp. 615–627.

Leping Li received the BE and MS degrees from the Department of mathematics, Harbin Institute of Technology in 2015 and 2017, respectively. He is currently working toward the PhD degree with the School of Computer Science and Technology, Beijing Institute of Technology. His research interests include fault localization and program repair.

Hui Liu received the BS degree in control science from Shandong University in 2001, the MS degree in computer science from Shanghai University in 2004, and the PhD degree in computer science from Peking University in 2008. He is currently a professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. He was a visiting research fellow with the Centre for Research on Evolution, Search, and Testing, University College London, U.K. His research interests include software refactoring, AI-based software engineering, software quality, and developing practical tools to assist software engineers. He served on the program committees and organizing committees of prestigious conferences, such as ICSME, RE, ICSR, and COMPSAC.

Kejun Li received the BE degree from the College of Information Engineering, Northwest A&F University in 2019. His is currently a graduate student with the School of Computer Science and Technology, Beijing Institute of Technology. His research interests include software testing and machine learning.

Yanjie Jiang received the BE degree from the College of Information Engineering, Northwest A&F University in 2017. She is currently working toward the PhD degree with the School of Computer Science and Technology, Beijing Institute of Technology. Her current research interests include software refactoring and software quality.

Rui Sun received the MS degree from the School of Computer Science and Technology, Beijing Institute of Technology in 2019. His research interests include software testing and data mining.