# A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments

Jooyong Yi[*]
Innopolis University, Russia
j.yi@innopolis.ru

Umair Z. Ahmed
Indian Institute of Technology
Kanpur, India
umair@cse.iitk.ac.in

Amey Karkare
Indian Institute of Technology
Kanpur, India
karkare@cse.iitk.ac.in

Shin Hwei Tan
National University of Singapore,
Singapore
shinhwei@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore,
Singapore
abhik@comp.nus.edu.sg

## ABSTRACT

Despite the fact an intelligent tutoring system for programming (ITSP) education has long attracted interest, its widespread use has been hindered by the difficulty of generating personalized feedback automatically. Meanwhile, automated program repair (APR) is an emerging new technology that automatically fixes software bugs, and it has been shown that APR can fix the bugs of large real-world software. In this paper, we study the feasibility of marrying intelligent programming tutoring and APR. We perform our feasibility study with four state-of-the-art APR tools (GenProg, AE, Angelix, and Prophet), and 661 programs written by the students taking an introductory programming course. We found that when APR tools are used out of the box, only about 30% of the programs in our dataset are repaired. This low repair rate is largely due to the student programs often being significantly incorrect — in contrast, professional software for which APR was successfully applied typically fails only a small portion of tests. To bridge this gap, we adopt in APR a new repair policy akin to the hint generation policy employed in the existing ITSP. This new repair policy admits partial repairs that address part of failing tests, which results in 84% improvement of repair rate. We also performed a user study with 263 novice students and 37 graders, and identified an understudied problem; while novice students do not seem to know how to effectively make use of generated repairs as hints, the graders do seem to gain benefits from repairs.

## CCS CONCEPTS

**•Applied computing →Computer-assisted instruction; •Software and its engineering →Software testing and debugging;**

## KEYWORDS

Intelligent Tutoring System, Automated Program Repair

[*]The first author did part of this work at National University of Singapore.

*ESEC/FSE'17, Paderborn, Germany*
© 2017 ACM. 978-1-4503-5105-8/17/09…$15.00
DOI: 10.1145/3106237.3106262

## 1 INTRODUCTION

Developing and using intelligent tutoring system for novice programmers has gained renewed attention recently [7, 9, 11, 12, 14, 34, 37, 38]. The typical goal of an intelligent tutoring system for programming (ITSP) is to find bugs in student programs and provide proper feedback for the students to help them correct their programs. An ITSP can also be used to help human tutors deal with many different student programs efficiently. While an ITSP for novice programmers has already existed since at least the early 80s [40], it has not been widely adopted in the education field. The main difficulty of building an effective ITSP is in the high degree of variations of student programs, which makes it challenging to automatically generate personalized feedback, without requiring additional help from the instructor. Despite this difficulty, with the advent of Massive Open Online Course (MOOC) and increasing interest in end-user programming, the need for an effective ITSP has never been greater. With the technological advances made during the last more than three decades since an early prototype system Meno-II [40] was introduced, it may now be possible to realize the widespread use of ITSP.

Automated program repair (APR) is an emerging new technology that has recently been actively researched [8, 10, 16, 20, 21, 23, 24, 28, 31, 44, 47]. An APR system fixes software bugs automatically, only requiring a test suite that can drive the repair process. Failing tests in the test suite become passing after repair, which manifests as a bug fix. APR was originally developed to fix professionally developed large software, and an APR tool, Angelix, recently reported and automated the fix of the Heartbleed bug [24]. In this paper, we seek to study the inter-play between APR and ITSP.

Given that student programs are much simpler than professionally developed software, applying APR to student programs may seem achievable. However, when we apply four state-of-the-art APR tools, namely GenProg [19], AE [44], Angelix [24], and Prophet [21] to 661 student programs (obtained from an introductory programming course offered by the third author at Indian Institute of Technology Kanpur), repairs are generated only for 31% of these programs. The remaining about 70% of the student programs in our dataset are not repaired by any of the four tools.

One of the main reasons for a low repair rate is that student programs are often severely incorrect, and fail the majority of the tests. In our dataset, 60% of the programs fail more than half of the available tests. This is in contrast to the fact that professional software for which APR was successfully applied typically fails only a small portion of tests. To rectify an incorrect program that fails the majority of tests, it is often necessary to make sizable changes to the program. Indeed, about half of the programs in our dataset require more than one hunk of changes to reach the correct programs (our dataset contains a corresponding correct program for each incorrect program). However, the current APR tools can fix only a small number of lines; most successful repairs reported in the literature change a small number of lines, and some tools such as SPR [20] and Prophet [21] even restrict the change to a single line. Given these discrepancies, it seems infeasible to use APR tools for the purpose of tutoring programming.

**Difference between Bug Fixing and Program Tutoring.** While we report in this paper APR tools' weak capability to fix novice student programs, showing a correct program to a student is not necessarily the best way to provide students with feedback. In fact, experienced human tutors show an answer only selectively when students make simple errors such as syntactic errors [27]. For more complex errors such as semantic errors, human tutors, in general, do not directly correct the error; instead, they give students hints. That way, tutors can help students move toward a correct answer.

**Partial Repairs as Hints.** Considering this difference between bug fixing and program tutoring, we explore the possibility of using APR tools for the purpose of generating hints, for the sake of teaching programming to students. When student programs fail multiple tests, we change the repair policy of APR tools as follows. Given an incorrect student program $P$, a repair candidate $P'$ is returned as a repair if (1) all previously passing tests still pass with $P'$, and (2) at least one of previously failing tests passes with $P'$. We call such as repair a *partial repair*, distinguishing it from the complete repair that passes all tests following the original repair policy of APR. By comparing a generated partial repair with the incorrect program, students can see when a particular test fails or passes, which can help a student understand why his or her program fails the test addressed by the partial repair. Since a generated partial repair $R$ is specialized for the tests addressed by $R$, the expected usage of partial repairs is to encourage students to modify their own incorrect program by taking account of the partial repair, rather than blindly accepting it.

We note that our partial repair is conceptually similar to the *"next-step hint"* advocated in the education field [2, 29, 32, 33, 35, 36]. By looking at a next-step hint, students can make forward progress toward an answer. In contrast, recent automated feedback generation techniques that appeared in the software engineering and programming languages fields [14, 37, 38] are evaluated under a restricted assumption that student programs are almost correct.

To facilitate the use of partial repairs as hints, our modified repair strategy generates one of the following two forms of repairs. The first kind of a partial repair is: `if` $(E)$ { $S$ }, where $S$ is a modified/added/deleted statement and $E$ is the guard expression for $S$. When such a form of a repair is generated, the student can obtain a hint about a data-flow change by observing the modification/addition/deletion of $S$, along with an additional hint about when that data-flow change is necessary by observing the guard $E$. The second kind of a partial repair modifies only conditional expressions, which gives students a hint about control-flow changes.

**Improved Feedback Rate.** After changing the repair policy (allowing partial repairs) and the repair strategy, feedback rate (repair generation rate) significantly improves, showing 84% improvement. In about 60% of the programs in our dataset, either complete or partial repairs are generated. By analyzing the remaining cases where repairs are not generated, we identify a few common reasons for repair failure — the two most common reasons being the need for output string modification and array modification for which the current APR tools are not specialized. It would be most cost effective to strengthen repair operators that can manipulate strings and arrays in future APR tools.

**User Study.** A high feedback rate is only one necessary condition for using APR for programming tutoring. To see whether automatically generated repairs actually help students and graders, we perform a user study with 263 students taking an introductory C programming course and 37 teaching assistants (TAs) of the same course, part of whose duty is to grade student assignments. In our user study, students' problem solving time increases when generated repairs are provided as hints, whereas TAs' grading performance improves. This difference seems to be due to that repairs generated by APR tools overfit the provided test-suite, which is the well-known problem in APR [39]. While TAs can, in general, spot the problems of the incorrect student program based on suggested repairs, novice students are likely to be distracted by the overly specialized suggestions. To transform automatically generated repairs into feedback that can actually help students, post-processing of generated repairs seems necessary, while answering the question about which form of feedback is beneficial for students remains a future challenge. Note that even if the ideal correct repair for a given student program is available, post-processing is still necessary to give the student a hint, not a solution.

**Our Contributions.** In our feasibility study of using APR for introductory programming assignments, we found that:

- The current state-of-the-art APR tools more often than not fail to generate a repair.
- However, they can, more often than not, generate partial repairs that pass part of previously failing tests. Generating partial repairs are analogous to that human tutors guide the students gradually toward the answer by giving them hints.
- Failure of APR is often due to a few common reasons such as the weak ability of APR tools to change the output string.
- Automatically generated repairs seem to help TAs grade student programs more efficiently.
- However, novice students do not seem to know how to effectively make use of suggested repairs to correct their programs.

Overall, it seems feasible to use APR tools for the purpose of tutoring introductory programming, given that repairs can be generated more often than not after tailoring APR tools, and further improvement seems possible by addressing a few common reasons for repair failure. To facilitate further research, we share our dataset containing 661 real student programs, our toolchain implementing the partial-repair policy/strategy, and our user-study materials in the following URL: https://github.com/jyi/ITSP. A summary description is available in Section 11.

Table 1: Characteristics of our dataset

| Lab | # Prog | Topic |
|---|---|---|
| Lab 3 | 63 | Simple Expressions, printf, scanf |
| Lab 4 | 117 | Conditionals |
| Lab 5 | 82 | Loops, Nested Loops |
| Lab 6 | 79 | Integer Arrays |
| Lab 7 | 71 | Character Arrays (Strings) and Functions |
| Lab 8 | 33 | Multi-dimensional Arrays (Matrices) |
| Lab 9 | 48 | Recursion |
| Lab 10 | 53 | Pointers |
| Lab 11 | 55 | Algorithms (sorting, permutations, puzzles) |
| Lab 12 | 60 | Structures (User-Defined data-types) |

## 2 AUTOMATED PROGRAM REPAIR

We perform a feasibility study with the following four state-of-the-art APR tools: GenProg [19], AE [44], Prophet [21], and Angelix [24]. These four tools, similar to the majority of APR tools, are test-driven, meaning that a modified program $P'$ is considered repaired if $P'$ passes all tests in the provided test suite. GenProg repeatedly modifies the program using genetic programming [17] until it finds a repair or the time budget is exhausted. In contrast to GenProg where the program is modified in a stochastic fashion (the program is modified differently at each run of the tool), AE modifies the program in a deterministic way by applying mutation operators to the program. Prophet first searches for a transformation schema that can be used to repair the program, and in the next step, it instantiates the transformation schema to generate a repair. In the second step of schema instantiation, Prophet uses a repair model learned from successful human patches to prioritize the instantiation similar to human patches. Angelix first searches for a set of angelic values for potentially buggy expressions $E$; when these angelic values substitute $E$, all tests are passed. In the next step, Angelix synthesizes patch expressions that return the angelic values found in the first step. These four APR tools, while sharing the goal of generating repairs that pass all tests, internally use different repair algorithms and repair operators. We include these different APR tools in our study to gain holistic understanding of the feasibility of using APR tools for programming tutoring.

## 3 DATASET

The dataset on which we perform and report our analysis was obtained from an Introductory C Programming (CS-101) course offered at Indian Institute of Technology Kanpur (IIT-K) by the third author. The programs were collected using *Prutor* [5], a system that stores intermediate versions of programs in addition to the final submissions. This course was credited by 400+ first year undergraduate students. One of the major grading component was weekly programming assignments (termed *Lab*). The assignments were designed around a specific topic every week, as described in Table 1, so as to test the concepts learned so far. The labs were conducted in an environment where we recorded the sequence of submissions made by students towards the goal of passing as many pre-defined test-cases as possible. Multiple attempts were allowed, with only the last submission being graded. For each of these labs, we pick a random sample of ($P_b$, $P_c$) program pairs as our dataset, where $P_b$ is a version of student program which fails on one or more test-cases, and $P_c$ is a later version of the attempt by the same student which passes all the provided test-cases. We exclude from our dataset the instances of $P_b$ failed to be compiled. The second column of Table 1 shows the number of programs for each lab we include in our dataset.

Table 2: The result of our initial experiment in which the existing APR tools are used out of the box. The overall repair rate is 31%.

| Lab | # Programs | # Fixed | Repair Rate | Time |
|---|---|---|---|---|
| Lab 3 | 63 | 3 | 5 % | 6 s |
| Lab 4 | 117 | 30 | 26 % | 20 s |
| Lab 5 | 82 | 27 | 33 % | 89 s |
| Lab 6 | 79 | 32 | 41 % | 50 s |
| Lab 7 | 71 | 17 | 24 % | 75 s |
| Lab 8 | 33 | 16 | 48 % | 139 s |
| Lab 9 | 48 | 15 | 31 % | 46 s |
| Lab 10 | 53 | 24 | 45 % | 24 s |
| Lab 11 | 55 | 26 | 47 % | 83 s |
| Lab 12 | 60 | 18 | 30 % | 38 s |
| Total | 661 | 208 | 31 % | 59 s |

## 4 INITIAL FEASIBILITY STUDY

How often can the state-of-the-art APR tools fix incorrect student programs? A high repair rate of APR is a prerequisite to using APR tools for feedback generation. As the first step of our feasibility study, we investigate how well four state-of-the-art APR tools (i.e., GenProg, AE, Prophet, and Angelix) fix the incorrect student programs in our dataset. For each incorrect program, a repair is considered found if one of the four APR tools successfully generates a repair — that is, a generated repair passes all provided tests of the program. We run the four APR tools in parallel until either (a) one of the APR tools successfully generates a repair or (b) all APR tools fail to generate a repair within a time limit (15 minutes). We use the default configuration of each APR tool with slight modifications for Prophet to extend the search space of repair [22]. Our experiment was performed on an Intel Xeon E5-2660 2.60 Ghz processor with Ubuntu 14.04 64-bit operating system and 62 GB of memory.

### 4.1 Results of Initial Experiment

Table 2 shows the results of our initial experiment. Each column represents (from left to right) the lab for which the incorrect programs were submitted (Lab), the number of incorrect programs submitted to the lab (# Programs), the number of incorrect programs in the lab that are fixed by the APR tools we apply (# Fixed), repair rate, i.e., (# Fixed)/(# Programs) in percentage (Repair Rate), and average time taken to successfully generate repairs (Time), respectively. In our experiments, repairs are generated only in 31% of the programs in our benchmark, and repair rate is below 50% across each individual lab. Meanwhile, the average time taken when repairs are found is about 1 minute. Our initial experimental result suggests that a low repair rate is a severe concern.

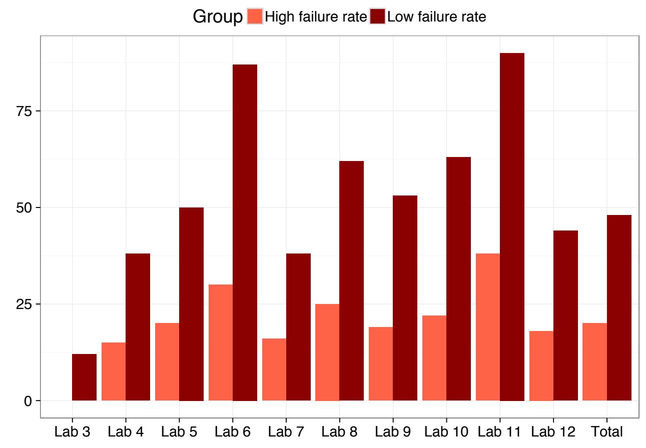


**Figure 1: This plot shows the repair rate of two different groups (Y axis) across each individual lab (X axis). The "High failure rate" group consists of the cases in which more than half of the tests fail the given program, whereas the "Low failure rate" group consists of the cases in which at least half of tests pass the given program. Repair rate is significantly lower in the high failure rate group to which 60% of the programs in our dataset belong.**

### 4.2 Reasons for Low Repair Rate

Despite the fact that student programs are simpler than programs written by professional developers for which APR tools are developed, the state-of-the-art APR tools fail to generate repairs for the majority of the incorrect program in our benchmark. Our result suggests that fixing short student programs is not easier than fixing developer programs. What makes automatically fixing student programs difficult? Answering this question may help us adjust APR to the new challenge posed by student programs. We observe in our dataset that the following two properties of student programs are likely to make automatically fixing student programs difficult: (1) student programs often fail in a majority of the tests, and (2) student programs often require complex fixes. We describe them in more detail in the following sections.

*4.2.1 High test failure rate.* Student programs are often significantly incorrect, and fail the majority of the tests. In our dataset, 60% of the programs fail more than half of the available tests. This is in contrast to the fact that professional software for which APR was successfully applied typically fails only a small portion of tests. High test failure rate is likely to make automated program repair difficult. Figure 1 compares the repair rate between the following two groups of our benchmark programs: the high test failure group in which more than half of the tests fail the given program and the low test failure group where at least half of tests pass the given program. While about half (48%) of the programs of the low failure rate group are successfully repaired, the repair rate of the high failure rate group is only 20%.

*4.2.2 Complex fixes.* The majority of bugs reported to be successfully repaired by APR tools are cosmetically simple, mostly restricted to one-line changes of the given buggy program. Still, the promise of APR is that it can save developers from manual search for a simple fix in large software. To investigate the distribution between simple fixes (one-hunk changes) and complex fixes (multiple-chunk changes) in our dataset, we compare each incorrect program in our dataset with its correct version. Recall that our dataset contains both an incorrect program and its correct version written by the same student. In our dataset, about half of the incorrect programs (46%) are fixed by adding more than 1 hunk of changes. For these programs requiring complex fixes, the repair rate is shown to be 26%, lower than the repair rate for the rest of the programs (36%).

## 5 TUTORING PROGRAMMING

Our initial experiment reveals that repair rate of the current APR tools for novice student programs is prohibitively low. Does this imply that it is infeasible to use APR for intelligent programming tutoring (IPT)? Or, given that APR was originally not developed for IPT, is it possible to tune up APR for the purpose of IPT?

One big difference between fixing a bug and tutoring programming is in the different degrees of their interactivity with the users. Tutoring is a highly interactive process between a tutor and a student. To complete a program, a student takes multiple steps of actions, and at each step, the tutor provides feedback. The tutor offers a confirmatory feedback if the student follows the right track toward a correct solution. Meanwhile, if the student goes astray, the tutor provides a hint for the student to get the student back on track. In this highly interactive tutoring process, the tutor does not simply show a correct program all at once. Instead, the tutor provides for the student a series of feedback to help the student stay on track toward a correct solution. This behavior of human programming tutors is recorded in detail in [27]. Intelligent tutoring systems expected to mimic human tutors should provide interactive feedback for the students where each feedback should help the students move to the next step toward a correct solution. In contrast, the ideal of APR is to synthesize a correct bug fix at once, without involving a long feedback loop with the developer. Given this difference between bug fixing and programming tutoring, we believe APR can be used for intelligent tutoring only after it is tailored to the new needs of programming tutoring.

## 6 FROM BUG FIXING TOWARD TUTORING

Given the difference between APR and IPT described in the previous section, the problem of APR and the problem of IPT can be described differently as follows.

*Definition 6.1 (Automated Program Repair (APR)).* Given a program $P$ and its specification $S$, the following holds true initially, reflecting the fact that $P$ is buggy: $P \nvdash S$. The problem of APR is to generate an alternative program $P'$ that satisfies $P' \vdash S$.

*Definition 6.2 (Intelligent Programming Tutoring (IPT)).* Given a program $P$ and its specification $S$ where $P \nvdash S$, the problem of IPT is to generate a series of alternative programs, $P'_1, P'_2, \ldots, P'_k, P'_{k+1}, \ldots, P'_n, P'_{n+1}$ that satisfies the following, through an iterative interaction with the student.

(1) For all odd numbers $k$, $P'_k$ is an automatically generated program by the tutoring system, and $P'_{k+1}$ is a program constructed by the student, using $P'_k$ as a hint.

(2) $\forall 1 \leq k \leq n : P'_k \preceq P'_{k+1}$, where $P'_k \preceq P'_{k+1}$ denotes that program $P'_{k+1}$ is closer to the specification $S$ than $P'_k$ (one definition of $\preceq$ will be described later).
(3) $\forall 1 \leq k \leq n : P'_k \nvdash S$
(4) $P'_{n+1} \vdash S$

Notice that in IPT, the final correct version of the program ($P'_{n+1}$) is sought for through a series of feedback generation (represented by $P'_k$ for all odd numbers $k$), interspersed with student programming (represented by $P'_{k+1}$ for all odd numbers $k$).

We describe in the following how we tailor APR to IPT. In particular, we tailor test-driven APR, given that the majority of APR tools use a test-driven approach. In test-driven APR, a test suite is used as the specification of the program. That is, given a test suite $T$ and a buggy program $P$ where $P$ does not pass all tests in $T$ (i.e., $P \nvdash T$), test-driven APR generates a repaired program $P'$ satisfying $P' \vdash T$, which denotes $P'$ passes all tests in $T$.

## 6.1 Tailoring Repair Policy

We tailor ITP described in Def. 6.2 to test-driven APR as follows. First, we replace in Def. 6.2 specification $S$ with test suite $T$. Thus, intermediate programs $P'_k$ do not pass all tests in $T$ ($P'_k \nvdash T$), while they are gradually approaching the final version that passes all tests. Meanwhile, the partial relation $\preceq$ used in Def. 6.2 can be naturally defined as follows. We say $P'_k \preceq P'_{k+1}$ if all tests passed in $P'_k$ also pass in $P'_{k+1}$. Similar to test-driven development, the progression of the student can be achieved by gradually passing more tests. While the number of tests may not be a precise measure of student progression, its practicality is high, given that tests are widely used in evaluating student programs. Related but orthogonal issues are how to construct an effective test suite for the purpose of IPT, and in which order each test should be satisfied by the program; for instance, given multiple failing tests $T_f$ and an incomplete program, which tests among $T_f$ should be addressed first? These orthogonal issues are not addressed in this initial feasibility study.

To implement progressive program construction ($P'_1 \preceq P'_2 \preceq \ldots \preceq P'_n$), we modify the repair policy of APR as follows. Taking as input a student program $P'_k$, we generate $P'_{k+1}$ that satisfies $P'_k \prec P'_{k+1}$. Note that it is not required for $P'_{k+1}$ to pass all tests, unlike in the original APR. This different repair policy can be implemented in an APR tool in a straightforward way by generating a partial repair defined as follows.

*Definition 6.3 (Partial Repair).* Given $n$ positive tests, where $n \geq 0$, and $m$ negative tests, where $m > 0$, a partial repair $P'$ satisfies the following:
(1) $P'$ passes all $n$ positive tests, and
(2) $P'$ passes at least one of $m$ negative tests.

In comparison, we define a complete repair generated in the original APR as follows.

*Definition 6.4 (Complete Repair).* Given $n$ positive tests, where $n \geq 0$, and $m$ negative tests, where $m > 0$, a complete repair $P'$ satisfies the following:
(1) $P'$ passes all $n$ positive tests, and
(2) $P'$ passes all $m$ negative tests.

The expected usage of partial repairs is to encourage students to modify their own incorrect program by taking into account the partial repair as a hint. In fact, a partial repair is specialized for the tests it addresses (the tests that turn from negative to positive after the partial repair), the student needs to generalize the partial solution shown to him or her. By comparing a generated partial repair with the incorrect program, students can see when a particular test fails or passes, which can help a student understand why his or her program fails the test addressed by the partial repair.

## 6.2 Tailoring Repair Strategy

Partial repairs are generated as hints, not as solutions. Typical hints partial repairs can provide are as follows.

(1) **Control-flow hints.** Students can see that a test can pass by changing the control flow of the program — which includes changing the direction of an if-conditional, skipping over a loop, and exiting a loop at a different iteration than before.
(2) **Data-flow hints.** Students can see that a test can pass by adding or deleting statements which affects the data flow of the program.
(3) **Conditional data-flow hints.** It is often the case that the data-flow of the program should be changed only under a certain circumstance. In this case, statement addition/deletion can be guarded with a condition. The deleted/added statements provide data-flow hints, while the guard conditions provide control-flow hints. Note that a data-flow hint can be viewed as a special case of a conditional data-flow hint where a statement $S$ is guarded with either false (suggesting the deletion of $S$) or true (suggesting the addition of $S$).

To facilitate the use of partial repairs as hints, we tailor the repair strategy of APR, following Algorithm 1. Our repair strategy searches for a control-flow hint and a conditional data-flow hint in parallel (a data-flow hint is the special case of a conditional data-flow hint). This parallel use of tools is shown in Line 2 of the algorithm: controlFix($P_b$,$T_p$,$T_n$) || condDataFix($P_b$,$T_p$,$T_n$), where $P_b$,$T_p$, and $T_n$ represent an input buggy program, positive tests (passing tests), and negative tests (failing tests), respectively. Function controlFix and condDataFix search for a partial repair that can be used as a control-flow hint and a (conditional) data-flow hint, respectively. Parallel search for a partial repair stops when either a repair is found or the time budget is exhausted.

In function controlFix by which a control-flow hint is searched for, we invoke in parallel two APR tools, Angelix and Prophet, both of which have repair operators that can modify the conditional expressions of the if/loop statements. We restrict the repair space only to conditional expression changes when looking for a control-flow hint. Meanwhile, in function condDataFix by which a conditional data-flow hint is searched for, we use the following two-step repair process. In the first step, we modify the data-flow of the program by adding/deleting/modifying statements such that one of the negative tests becomes positive after the modification. At this step, we do not preserve positive tests; that is, the modified program $P_i$ in line 9 may fail some/all of positive tests. However, in the second step, we refine $P_i$ such that the refined program $P_r$ (obtained in either line 12 or 16) passes all positive tests. More specifically, our refinement process takes place as follows. Given a statement $S$ that is added or deleted in the first step, we transform $S$ into "`if (true)`

**Algorithm 1** Partial Repair Generation Using Our Repair Strategy

```
Input: buggy program $P_b$, test suite $T$
Output: partially repaired program $P_r$

    ▹ Run $P_b$ with $T$ to find out positive tests $T_p$ and negative tests $T_n$.
1:  $(T_p, T_n) \leftarrow$ run($P_b$, $T$)
    ▹ Parallel call. Successful termination of one function (termination with
    a non-NULL value) kills the remaining function.
2:  $P_r \leftarrow$ controlFix($P_b$,$T_p$, $T_n$) || condDataFix($P_b$,$T_p$, $T_n$)

    ▹ Function controlFix searches for a partial repair changing the
    control-flow of the program, using Angelix and Prophet. If a partial
    repair is not found, NULL is returned.
3:  function controlFix($P_b$,$T_p$,$T_n$)
        ▹ Set the repair configuration such that a partial repair changing
        the control-flow of the program is searched for.
4:      $C \leftarrow \{control, partial\}$
5:      return runAngelix($C$,$P_b$,$T_p$,$T_n$) || runProphet($C$,$P_b$,$T_p$,$T_n$)
6:  end function

    ▹ Function condDataFix searches for a partial repair changing the
    data-flow and/or the control-flow of the program. If a partial repair is
    not found, NULL is returned.
7:  function condDataFix($P_b$,$T_p$,$T_n$)
        ▹ Set the repair configuration such that a partial repair changing
        the data-flow of the program is searched for.
8:      $C \leftarrow \{data, partial\}$
        ▹ Search for a program $P_i$ that makes at least one of the tests in
        $T_n$ pass, while ignoring $T_p$. $T_i$ represents a set of tests in $T_n$ that
        pass with $P_i$.
9:      $(P_i, T_i) \leftarrow$ runGenProg($C$,$P_b$,$T_n$) || runAE($C$,$P_b$,$T_n$) ||
            runAngelix($C$,$P_b$,$T_n$) || runProphet($C$,$P_b$,$T_n$)
        ▹ If $P_i$ is found (i.e., $P_i$ != NULL), refine $P_i$ such that not only $T_i$
        but also all the tests in $T_p$ pass. This is achieved by looking for a
        complete (not partial) repair that passes all tests in $T_p \cup T_i$
10:     if $P_i$ != NULL then
11:         $C \leftarrow \{control, complete\}$
12:         $P_r \leftarrow$ RunAngelix($C$,$P_i$,$T_p \cup T_i$)
13:     end if
        ▹ If refinement with Angelix fails (i.e., $P_r$ == NULL), try with
        Prophet.
14:     if $P_i$ != NULL && $P_r$ == NULL then
15:         $C \leftarrow \{control, complete\}$
16:         $P_r \leftarrow$ RunProphet($C$,$P_i$,$T_p \cup T_i$)
17:     end if
18:     return $P_r$
19: end function
```

{ S }" or "if (false) { S }", respectively. Similarly, if a statement $S$ is modified into another statement $S'$ in the first step, we transform $S$ into "if (true) { S' } else { S }". This transformation takes place internally inside the APR tools we modify for this purpose. The refined program $P_r$ is obtained by replacing the tautological conditions (true or false) guarding the added/deleted/modified statement with different expressions with which $P_r$ passes all positive tests and the negative tests addressed in the first step. We invoke multiple APR tools in parallel in the two-step repair process of finding a conditional data-flow hint. In the first step, we invoke four tools, that is, GenProg, AE, Prophet, and Angelix (for Prophet and Angelix, we turn off the options that allow conditional expression changes). In the second step where guards are modified, we invoke only Prophet and Angelix, since GenProg and AE do not support expression-level modifications.

## 6.3 Incremental Repair

Our overall repair algorithm optionally allows incremental repair, that is, generating a series of partial repairs incrementally. More specifically, a new partial repair $P_{i+1}$ is generated based on the previous partial repair $P_i$ generated at the $i$-th iteration. The number of passing tests grows as the iteration proceeds, and the tests passed by $P_i$ are also passed by $P_{i+1}$. The iteration proceeds until either there is no remaining negative (failing) test or a partial repair is not found. A repair obtained through the incremental repair approach can be useful for graders to whom showing as many changes as possible can provide hints about why the student program is wrong.

## 7 EVALUATION

We evaluate the feasibility of using our partial repair algorithm for introductory programming assignments. The following are our research questions.

**RQ1** How often are repairs generated when our partial repair algorithm is employed in addition to the complete repair algorithm of the existing APR tools? A high repair rate is a prerequisite for using APR for introductory programming assignments. The current state-of-the-art APR tools fail to generate repairs more often than not, as shown in Section 4. How significantly does a new repair strategy allowing both complete and partial repairs improve repair rate?
**RQ2** When are repairs not generated even after employing our partial repair algorithm? If there are common reasons for those cases of repair failure, they should be addressed in future tools.
**RQ3** Do tool-generated partial repairs help students in finding a solution more efficiently than when repairs are not shown?
**RQ4** Similarly, do tool-generated repairs help graders in grading student programs more efficiently than when repairs are not shown?

To investigate our research questions, we conduct a tool experiment (to address RQ1 regarding repair rate), repair failure analysis (to address RQ2), and user study (to address RQ3 and RQ4).

## 7.1 Tool Experiment

We developed a tool that implements our partial repair algorithm on top of the same four existing APR tools as used in our initial experiment. We apply our tool to the same dataset as used in our initial experiment modulo the incorrect programs for which complete repairs are already generated in the initial experiment. Recall that the purpose of this tool experiment is to investigate how significantly a new repair strategy allowing both complete and partial repairs improves repair rate. The experiment was performed on the same environment as used for the initial experiment.

Table 3 shows the results of our tool experiment. As compared to our initial experiment that does not allow partial repairs, the overall repair rate increases from 31% to 57%, showing about 84% of improvement. Repair rate increases significantly across all labs, as shown in Figure 2. Meanwhile, the average successful repair time stays as low as 58 seconds.

Table 3: **The result of an experiment in which partial repairs are sought for in case a complete repair is not found out. The overall repair rate is about 60%.**

| Lab | # Programs | # Fixed | Repair Rate | Time |
|---|---|---|---|---|
| Lab 3 | 63 | 14 | 22 % | 3 s |
| Lab 4 | 117 | 61 | 52 % | 27 s |
| Lab 5 | 82 | 52 | 63 % | 85 s |
| Lab 6 | 79 | 49 | 62 % | 69 s |
| Lab 7 | 71 | 44 | 62 % | 51 s |
| Lab 8 | 33 | 28 | 85 % | 99 s |
| Lab 9 | 48 | 26 | 54 % | 70 s |
| Lab 10 | 53 | 36 | 68 % | 35 s |
| Lab 11 | 55 | 33 | 60 % | 77 s |
| Lab 12 | 60 | 35 | 58 % | 52 s |
| Total | 661 | 378 | 57 % | 58 s |

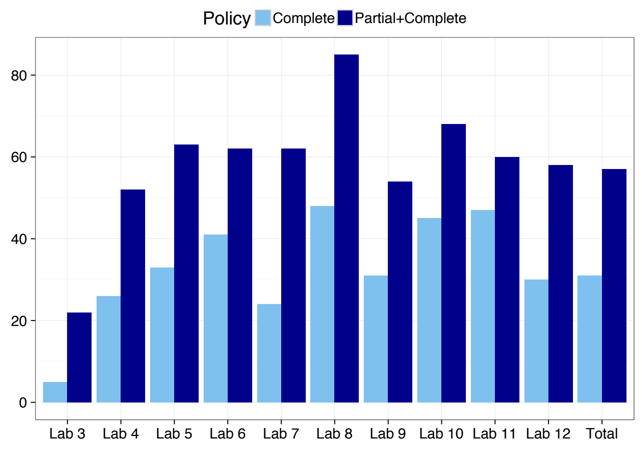


**Figure 2: This plot shows the repair rate in percentage (Y axis) across each individual lab (X axis). The "Complete" represents the cases in which only complete repairs are counted, whereas the "Partial+Complete" represents the cases in which partial repairs are also allowed in case a complete repair does not exist.**

## 7.2 Repair Failure Analysis

Despite the increase of repair rate after allowing partial repairs, neither complete repair nor partial repair was generated in 43% of our subject programs. We compare these 43% of programs with their correct versions to look for common reasons for repair failure. Specifically, for each defect represented by the buggy version $P_b$ and the correct version $P_c$, we obtain the AST differences between $P_b$ and $P_c$ using Gumtree [6], an AST differencing tool. We first perform manual inspection of the AST differences to derive a set of common characteristics observed in the differences between $P_b$ and $P_c$. Then, we detect other such instances in our dataset, using our extension of Gumtree where we encode the AST difference patterns corresponding to the common characteristics we identified. We repeat this process until all programs for which repairs are not generated are covered. Note that some programs are labeled with multiple characteristics in this process.

Table 4: **This table shows the distribution of the difference characteristics of the two programs, a buggy program ($P_b$) and its correct version ($P_c$), for which neither complete nor partial repair is generated by the APR tools.**

| $P_c - P_b$ | # Instances | Portion |
|---|---|---|
| String | 125 | 40 % |
| Array | 44 | 14 % |
| Missing Function | 38 | 12 % |
| Complex Control | 35 | 11 % |
| Unsupported | 30 | 10 % |
| Others | 16 | 5 % |
| Empty Implementation | 12 | 4 % |
| Wrong Parameters | 6 | 2 % |
| Wrong Usage | 6 | 2 % |

Table 4 shows our analysis result. The first column categorizes the characteristics of the differences between the buggy program ($P_b$) and its corrected version ($P_c$). The second and third column show the number of instances and portion of each category, respectively, by which the table is sorted. The following describes each category for $P_c - P_b$ which we represent as $\delta$:

**String** This corresponds to the case where $\delta$ involves changing the string constants used in the program, such as adding a missing space or a new line. It is observed that this category takes the most number of instances of repair failure (40%).

**Array** This corresponds to the case where $\delta$ involves changes in arrays that include array index changes, array size changes, adding/deleting array access expressions, and using array lengths in the program.

**Missing Function** This corresponds to the case where $\delta$ involves adding a function call.

**Complex Control** This corresponds to the case where $\delta$ involves complex control-flow changes that include control-flow changes in a nested loop and control-flow changes in multiple conditionals. While Angelix and Prophet can change conditional expressions, they do not exhaustively consider all possible control-flow changes.

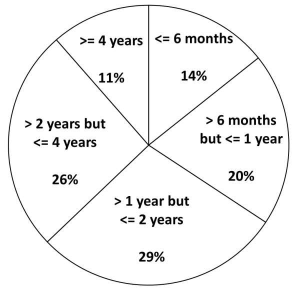
**Unsupported** This corresponds to the case where $P_c$ requires expressions that cannot be synthesized by the current APR tools such as the expressions involving the modular operator and non-linear expressions.

**Empty Implementation** This corresponds to the case where the main function of $P_b$ is empty or contains only a return statement. We do not label other characteristics for the programs belonging to this category.

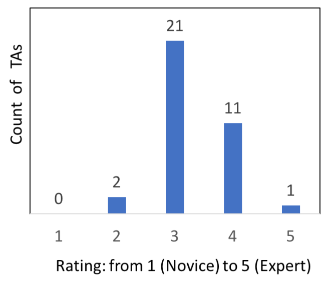
**Wrong Parameters** This corresponds to the case where $\delta$ involves changing multiple parameters of a function call expression. While Angelix can change multiple expressions, it does not exhaustively consider all possible combinations.

**Wrong Usage** This corresponds to the case where students use language constructs in a semantically wrong way. This includes mistakenly adding a semicolon before a for-loop body (i.e., using `for(...); {...}` instead of `for(...) {...}`), using `scanf(...,x)` instead of `scanf(...,&x)` where `x` represents a variable, using `++x` when `x+1` is required, using 'x' when `x` is required, and using `*x` when `x` is required.

**Others** This covers the rest of the characteristics.



(a) Programming Experience



(b) Skill Level for C Programming

**Figure 3: Background of Teaching Assistants**

The fact that the portions of the top two categories (String and Array) take more than 50% suggests that it would be most cost effective to strengthen repair operators that can manipulate strings and arrays in future APR tools.

## 7.3 User Study

We perform a user study with novice students and graders to see

(1) whether automatically generated feedback can help students solve the problem on their own.
(2) whether automatically generated feedback can help teaching assistants (TAs) grade submissions efficiently (faster grading) and effectively (only small variation in the marks for similar submissions.)

For the student study, we selected 5 problems for which we had buggy submissions and the partial repairs generated by our algorithm. We divided the students into the experimental group for whom the generated repairs are presented and the control group for whom the repairs are not presented. Repairs are presented in the form of a comment around the repaired lines of the buggy submission. We asked each student to fix one randomly chosen buggy submission. The study was unannounced, that is, the task was provided as a bonus question along with other regular assignment problems. The weight of the fix-task was kept low so that it does not impact the overall grade of the students in the course. The participation was voluntary, and in total 263 students submitted their completed programs (140 students in the without-repair group, and 123 students in the with-repair group), out of the 400+ students crediting the course.

Similarly, to estimate the impact of repairs on the grading task, we did a study on TAs. 37 TAs volunteered for this task, out of which 35 filled in the pre-study survey and the post-study survey. Figure 3 summarizes the background of these TAs, which we collected using a pre-study survey. For the study, we randomly collected 43 buggy submissions from the subset of our dataset for which our algorithm successfully generates either complete or partial repairs. These 43 buggy submissions correspond to 8 different programming problems. We asked the TAs to grade these submissions based on how close they are to a correct program by figuring out the bugs and their corresponding repairs. The TAs were divided into two groups. The first group was given 22 tasks (set A) without repair, and 21 tasks (set B) with repairs, while the second group was conversely given set A with repairs, and set B without repairs. We

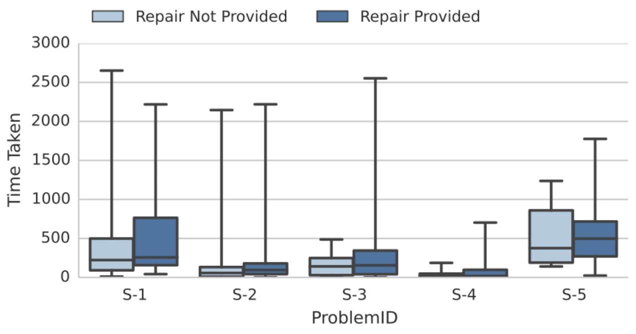


**Figure 4: Time taken by students for bug fix task**

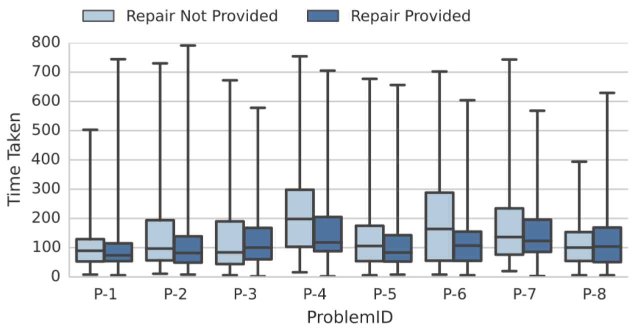


**Figure 5: Time taken by TAs for grading task**

**Table 5: The answer frequency from TAs for the question: How do you categorize the errors of the program based on the suggested repair? Multiple answers are allowed.**

| Category | Frequency |
|---|---|
| Conditionals | 29 |
| Loops | 19 |
| Missing Character | 8 |
| String Modifications | 5 |
| Array Accesses | 4 |
| User defined Functions | 4 |
| Missing Values in the Output | 4 |
| Library Functions | 3 |
| Others | 3 |
| Missing Whitespace in the Output | 2 |
| Floating Point Operations | 1 |

compared the time taken and marks assigned by the TAs for these tasks. The reference marks for these submissions were provided by the instructor who did not participate in the study, and did not have access to the repairs. With TAs, we also conducted a post-study survey to understand the experience of TAs with repairs.

Figure 4 and Figure 5 respectively show the distribution of time taken by the students for the solving task, and time taken by the TAs for the grading task. Both the figures are box-plots where X-axis shows the problem IDs and Y-axis shows the time in seconds. Each box (or rectangle) represents the first and third quartiles, with

**Table 6: The answer frequency from TAs for the question: What kind of modifications are necessary in the suggested repair to obtain a correct solution? Multiple answers are allowed.**

| Description | Frequency |
|---|---|
| Fix condition for Conditionals or Loops | 32 |
| Fix Operators | 15 |
| Insert/Delete Character (e.g., ; , &) | 15 |
| Formatting the Output (whitespaces) | 11 |
| Fix Constants | 10 |
| Fix Array Indices | 4 |
| Others | 4 |

**Table 7: Analysis of TA Grading Time. "Yes" TAs correspond to those who answered in the post-study survey that repairs were useful, while "No" TAs answered conversely.**

| Grading Time (sec) | All TAs | | "Yes" TAs | | "No" TAs | |
|---|---|---|---|---|---|---|
| | Without Repair | With Repair | Without Repair | With Repair | Without Repair | With Repair |
| **Average** | 173.76 | 135.41 | 155.08 | 124.83 | 191.40 | 145.39 |
| **Median** | 150.95 | 133.68 | 120.70 | 126.90 | 166.87 | 144.85 |
| **Stdev** | 96.70 | 40.88 | 99.98 | 40.30 | 92.82 | 39.96 |

a horizontal line inside indicating the median value. The ends of the vertical lines (or whiskers) on either side of the box represent the minimum and maximum time-taken. From these figures, we can infer that repairs affect novice students and TAs differently. While the problem solving time of the students tends to increase in the group where repairs are shown, the grading time of the TAs tends to decrease when repairs are shown. That is, when repairs are shown, the students tend to solve the problems more slowly, while the TAs tend to grade the problems more quickly. We conjecture that these opposite trends between novice students and TAs are due to their different levels of expertise and the format of feedback. In our post-study survey, we asked TAs (1) how do you categorize the errors of the program based on the suggested repair? and (2) what kind of modifications are necessary in the suggested repair to obtain a correct solution? The results are shown in Table 5 and 6. In the first question, the majority of TAs identified the errors in loops/conditionals (see Table 5), while in the second question, the most number of answers were given to the changes in loops/conditionals (see Table 6). These results suggest that TAs are capable of generalizing suggested repairs that are overly specialized to the tests. It is likely that this generalization capability of TAs helps them finish the grading tasks more efficiently. However, novice students do not seem to know how to effectively make use of suggested repairs, unlike TAs.

Table 7 shows a closer look at the grading performance of TAs. The first column (All TAs) shows the performance statistics for all TAs (both without repair and with repair), and the second column ("Yes" TAs) and the third column ("No" TAs) show the statistics for those who said in the post-study survey that the suggested repairs are useful and not useful, respectively. Half of the TAs answered the repairs are useful (the Yes group) and the rest of the half answered

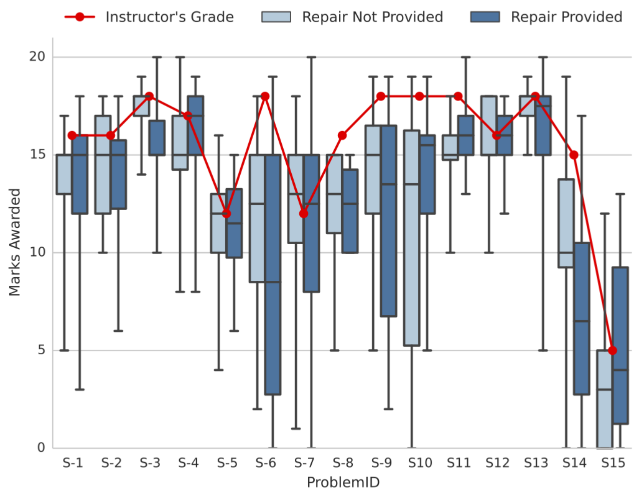


**Figure 6: Distribution of marks assigned by TAs**

not useful (the No group). Given that the average grading time is smaller in the Yes group, high performers tend to feel more strongly that the suggested repairs are useful. In both groups, the average grading time decreases when repairs are shown. Also notably, the standard deviation decreases in both groups, indicating that the gap between high performers and low performers becomes narrower when repairs are shown.

Figure 6 shows the marks awarded by the TAs for 15 randomly picked submissions out of 43 tasks. The reference marks for these submissions were provided by the instructor who did not participate in the study, and also did not look at the generated repairs used in the study. In the graph, the X-axis and Y-axis show, respectively, the problem IDs and marks awarded (between 0 and 20). The overall trends are similar among the group for whom repairs are presented (experimental group), the group for whom repairs are not presented (control group), and the independent instructor. In a majority of the cases the absolute difference between the experimental group and the control group is not much: $\leq 1$ for $22/43$ cases and $\leq 2$ for $30/43$ cases.

## 8 THREATS TO VALIDITY

In our tool experiments, one of the APR tools, GenProg, uses a random algorithm (genetic programming), which can produce different results for each run. To mitigate this threat, we applied the same seed to GenProg in our initial experiment (Section 4) and the second tool experiment (Section 7.1). Also, the fact that the rest of the APR tools employed for our experiments (AE, Prophet, and Angelix) use deterministic repair algorithm further mitigates this threat. Our repair failure analysis (Section 7.2) may be restricted by the difference categories, $P_c - P_b$, to which our analysis tool categorizes. To mitigate this threat, we manually inspected the differences and added new categories when the previously used categories were not sufficient. Our dataset, while collected from the actual students taking an introductory programming course, may not be representative of all student programs. Similarly, in our user study, participating students and TAs may not represent all novice students and graders. In terms of the programming language, our results are confined to C programs for which APR has been developed most actively. However, our proposed partial-repair policy can be applied to other programming languages. In our user study, the experimental setting—where the participating students are given buggy programs written by other students—is not identical with the actual usage scenario where the students fix the mistakes they made. We leave further investigation as future work.

## 9 RELATED WORK

Many different techniques have been applied to automated feedback generation, and each technique has different advantages and disadvantages. Program equivalence checking is used in [14] where behavioral difference between a student program and its reference program (differences in input-output relations) is reported to the student as feedback. Since program equivalence checking is generally undecidable, [14] performs equivalence checking in a constrained manner — that is, when comparing a student program $P_s$ with its reference program $P_r$, $P_r$ should be structurally similar to $P_s$. Such $P_r$ can be provided either manually (the instructor prepares $P_r$) or semi-automatically (the instructor selects $P_r$ from previously submitted correct student programs which can be automatically clustered according to their structures). Since this approach based on program equivalence uses a reference program as a specification, it can generate feedback even when there is no failing test. Meanwhile, the fact that the instructor should validate the correctness of the reference program poses not only a burden to the instructor, but also a risk of generating false feedback in the presence of a validation mistake.

A model-based approach is used in [38] where an instructor-given error model describing possible student errors defines how the given incorrect program is allowed to be modified. To search for a correct modification efficiently, a program synthesis technique is employed. While an error model can capture some common student errors and hence can guide feedback generation, it also restricts the search for feedback only to the common errors described in the error model. The fact that an error model should be prepared beforehand by the instructor is another disadvantage.

Static analysis is used in [1, 46] where dependence graphs extracted from a student program $P_s$ and the reference program $P_r$ are compared to each other, in order to identify a statement in $P_s$ that can potentially cause semantic difference from $P_r$. As usual in conservative static analysis, these approaches can guarantee not to miss a semantic error, while as a flip side, an error can be falsely reported.

A learning-based approach is used in Refazer [37] and DeepFix [13]. Refazer learns programs transformation rules from the past program changes, similar to [25, 26] where systematic edits (similar, but not identical, changes made in many program locations) are learned from the past program changes. Meanwhile, DeepFix applies deep learning to the correction of syntactic errors that cause compilation failure. While learning-based approaches can complement the existing approaches when the previous submissions of a programming assignment are available, their applicability and effectiveness are restricted by the availability and the quality of the previous submissions.

A data-driven approach is used in [36] where, in order to generate feedback, not only the reference program, but also a chain of intermediate programs leading to the reference program are exploited. The use of intermediate programs makes it more amenable to generate a next-step hint, since the buggy student program is likely to be closer to one of the intermediate programs than to the reference program. However, the hint space, consisting of the reference programs and their intermediate programs, is more restricted than the one of APR.

Automated program repair (APR) is fully automatic unlike some approaches requiring additional input from the instructor, such as an error model and multiple reference programs from multiple clusters. In APR, it is sufficient to provide a student program and a test suite. Although generated repairs can be imperfect and overly specialized to the provided test suite [39], this issue has been gradually addressed in recent work of APR [4, 18, 21, 23, 41, 42, 45]. Meanwhile, fault localization can also be used to provide hints to students, as suggested in [3]. In fact, APR also performs fault localization in the sense that APR performs fault localization before synthesizing a fix. Furthermore, students can also see how a previously failing test passes after fix, which provides an additional hint.

There have been several user studies in the area of program debugging and repair [15, 30, 43]. Unlike these user studies concerning the productivity of professional developers, our study is conducted with different target of users, that is, novice students and graders. Overall, our study provides holistic information about the feasibility of using APR for introductory programming assignments, including how often repairs are generated, why repairs are failed to be generated, and how useful generated repairs are for students and graders.

## 10 CONCLUSION

In this paper, we have explored the possibility of using APR as a feedback generation engine of intelligent tutoring systems for introductory programming. We have performed a feasibility study with four state-of-the-art APR tools (GenProg, AE, Prophet, and Angelix) and real student programs collected from a course on introductory programming. Although out-of-the-box application of APR tools seems infeasible due to the low repair rate, we have shown that repair rate can be boosted by tailoring the repair policy and strategy of APR to the needs of intelligent tutoring. Most notably, adopting a partial repair policy akin to the next-step hint generation advocated in the education field seems effective in terms of improving feedback generation rate. We have also shown through a repair failure analysis that repair failures are often caused by a few common reasons. Further improvement of feedback generation rate is expected by strengthening repair operators manipulating strings and arrays in future APR tools. Lastly, we have shown our user study results performed with novice students and graders (TAs). In contrast to the TAs who use the suggested repairs as hints to efficiently complete the grading tasks, the novice students do not seem to know how to effectively make efficient use of suggested repairs to correct their programs. We leave as future work a study of effective post-processing of repairs to transform them to hints more comprehensible to novice students.

## ACKNOWLEDGMENTS

The first author thanks Innopolis University for its support.

## 11 ARTIFACT DESCRIPTION

To facilitate further research, we share the artifacts used in this study in the following GitHub page:

```
https://github.com/jyi/ITSP
```

A detailed description about the artifacts—including how to use them—is provided in our GitHub page. In this section, we provide summarized information about the shared artifacts.

### 11.1 Available Artifacts

The following artifacts are available in our GitHub page:

(1) Dataset containing 661 student programs we used in our experiments (see Section 3)
(2) A toolchain implementing our new repair policy (see Section 6.1) and strategy (see Section 6.2)
(3) User study materials including the survey questionnaire we used and survey responses (see Section 7.3)

**Docker Image.** The provided toolchain runs on top of four APR tools, namely GenProg [19], AE [44], Angelix [24], and Prophet [21]. We provide a docker image where all these four tools are already installed. Our docker image can be downloaded from Docker Hub:

```
docker pull jayyi/itsp:0.0
```

Note that the size of the image is quite large (> 30 GB). The following more lightweight image (about 3 GB) is also available: jayyi/itsp-no-angelix:0.0, which does not contain one of APR tools, Angelix. In the lightweight image, the provided toolchain does not use Angelix when generating a repair.

**Tutorial.** We provide in our GitHub page a tutorial about how to use our toolchain and how to interpret the toolchain result.

### 11.2 Potential Users of the Artifacts

Our artifacts can be useful for:

(1) Those who want to *reproduce* our experimental results.
(2) Those who need a *benchmark* for an intelligent tutoring system for programming (ITSP). Our shared dataset contains 661 incorrect student programs, their reference programs (correct versions) and test suites.
(3) Those who want to *extend* our partial repair policy/strategy. The provided toolchain (written in Bash) implements our partial repair policy/strategy.
(4) Those who want to conduct a *user study* on ITSP. We share the survey questionnaire used in our user study.

### 11.3 A Note on Reproducibility

GenProg uses a random algorithm. Also, parallel use of multiple repair tools introduces one more layer of randomness (there is no guarantee that one repair tool always finds a repair faster than the other tools). Thus, different results may be produced at each experiment. We provide raw experimental data we obtained in our GitHub page (located in the experiment/cache directory) from which the summarized information we provide in this paper (i.e., Table 2, Figure 1, Table 3 and Figure 2) can be reproduced using the provided script (analysis.R).

## REFERENCES

[1] Anne Adam and Jean-Pierre H. Laurent. 1980. LAURA, A System to Debug Student Programs. *Artif. Intell.* 15, 1-2 (1980), 75–122.
[2] Tiffany Barnes and John C. Stamper. 2008. Toward Automatic Hint Generation for Logic Proof Tutoring Using Historical Student Data. In *Intelligent Tutoring Systems*. 373–382.
[3] Geoff Birch, Bernd Fischer, and Michael Poppleton. 2016. Using Fast Model-Based Fault Localisation to Aid Students in Self-Guided Program Repair and to Improve Assessment. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2016*. 168–173.
[4] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantitative Objectives. In *CAV*. 383–401.
[5] Rajdeep Das, Umair Z. Ahmed, Amey Karkare, and Sumit Gulwani. 2016. Prutor: A System for Tutoring CS1 and Collecting Student Programs for Analysis. *CoRR* abs/1608.03828 (2016). http://arxiv.org/abs/1608.03828
[6] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ASE*. 313–324.
[7] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Trans. Comput.-Hum. Interact.* 22, 2 (2015), 7:1–7:35.
[8] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*. 3–13.
[9] Sebastian Gross, Bassam Mokbel, Benjamin Paaßen, Barbara Hammer, and Niels Pinkwart. 2014. Example-based feedback provision using structured solution spaces. *IJLT* 9, 3 (2014), 248–280.
[10] Zhongxian Gu, Earl T. Barr, David J. Hamilton, and Zhendong Su. 2010. Has the Bug Really Been Fixed?. In *ICSE*. 55–64.
[11] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. 2014. Feedback generation for performance problems in introductory programming assignments. In *FSE*. 41–51.
[12] Philip J. Guo. 2015. Codeopticon: Real-Time, One-To-Many Human Tutoring for Computer Programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, UIST 2015, Charlotte, NC, USA, November 8-11, 2015*. 599–608.
[13] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *AAAI*. 1345–1351.
[14] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-supervised verified feedback generation. In *FSE*. 739–750.
[15] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. 2014. MintHint: automated synthesis of repair hints. In *ICSE*. 266–276.
[16] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *ICSE*. 802–811.
[17] John R. Koza. 1993. *Genetic programming - on the programming of computers by means of natural selection*. MIT Press.
[18] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *SANER*. 213–224.
[19] C. Le Goues, ThanhVu Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan 2012), 54–72.
[20] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *ESEC/FSE*. 166–178.
[21] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *POPL*. 298–312.
[22] Fan Long and Martin C. Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *ICSE*. 702–713.
[23] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *ICSE*. 448–458.
[24] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *ICSE*. 691–701.
[25] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic editing: generating program transformations from an example. In *PLDI*. 329–342.
[26] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: locating and applying systematic edits by learning from examples. In *ICSE*. 502–511.
[27] Douglas C. Merrill, Brian J. Reiser, Shannon K. Merrill, and Shari Landes. 1995. Tutoring: Guided Learning by Doing. *Cognition and Instruction* 13, 3 (1995), 315–372.
[28] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *ICSE*. 772–781.
[29] Luc Paquette, Jean-François Lebeau, Gabriel Beaulieu, and André Mayers. 2012. Automating Next-Step Hints Generation Using ASTUS. In *Intelligent Tutoring Systems*. 201–211.
[30] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *ISSTA*. 199–209.

[31] Yu Pei, C.A. Furia, M. Nordio, Yi Wei, B. Meyer, and A. Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering* 40, 5 (May 2014), 427–449.

[32] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. 2015. Autonomously Generating Hints by Inferring Problem Solving Policies. In *Proceedings of the Second ACM Conference on Learning @ Scale*. 195–204.

[33] Leena M. Razzaq, Neil T. Heffernan, and Robert W. Lindeman. 2007. What Level of Tutor Interaction is Best?. In *Artificial Intelligence in Education*. 222–229.

[34] Kelly Rivers and Kenneth R. Koedinger. 2013. Automatic Generation of Programming Feedback; A Data-Driven Approach. In *Proceedings of the Workshops at the 16th International Conference on Artificial Intelligence in Education AIED 2013, Memphis, USA, July 9-13, 2013*.

[35] Kelly Rivers and Kenneth R. Koedinger. 2014. Automating Hint Generation with Solution Space Path Construction. In *Intelligent Tutoring Systems*. 329–339.

[36] Kelly Rivers and Kenneth R. Koedinger. 2017. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 37–64.

[37] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Bjoern Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *ICSE*. 404–415.

[38] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *PLDI*. 15–26.

[39] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *ESEC/FSE*. 532–543.

[40] Elliot Soloway, Beverly Park Woolf, Eric Rubin, and Paul Barth. 1981. Meno-II: An Intelligent Tutoring System for Novice Programmers. In *IJCAI*. 975–977.

[41] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated repair of software regressions. In *ICSE*. 471–482.

[42] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *FSE*. 727–738.

[43] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically generated patches as debugging aids: a human study. In *FSE*. 64–74.

[44] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE*. 356–366.

[45] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *ICSE*. 416–426.

[46] Songwen Xu and Yam San Chee. 2003. Transformation-Based Diagnosis of Student Programs for Programming Tutoring Systems. *IEEE Trans. Software Eng.* 29, 4 (2003), 360–384.

[47] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Software Eng.* 43, 1 (2017), 34–55.