AUTOMATED FEEDBACK GENERATION FOR PROGRAMMING ASSIGNMENTS

by

GEORGIANA HALDEMAN

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Thu D. Nguyen and He Zhu

and approved by

New Brunswick, New Jersey

October, 2021

ABSTRACT OF THE DISSERTATION

Automated Feedback Generation for Programming Assignments

By Georgiana Haldeman

Dissertation Directors:

Thu D. Nguyen and He Zhu

Autograding systems are being increasingly deployed to meet the challenges of teaching programming at scale. Studies show that formative feedback can greatly help novices learn programming. This work explores techniques for extending an autograder to provide corrective and formative feedback on programming assignment submissions using a mixed approach.

The dissertation first introduces a framework to help instructors identify common student errors for a programming assignment and write hints that the autograder can provide automatically for these errors. This approach starts with the design of a knowledge map, which is the set of concepts and skills that are necessary to complete an assignment, followed by the design of the assignment and that of a comprehensive test suite for identifying logical errors in the submitted code. Test cases are used to test the student submissions and learn classes of common errors. For each assignment, the instructor trains a classifier that automatically categorizes errors in a submission based on the outcome of the test suite. The instructor maps the errors to corresponding concepts and skills and writes hints to help students find their misconceptions and mistakes.

I apply this methodology to two assignments in the Introduction to Computer Science course and find that the automatic error categorization has a 90% average accuracy. I report and compare data from two semesters, one semester when hints are given for the two assignments and one when hints are not given. Results show that the percentage of students who complete the assignments after an initial erroneous submission is three times greater when hints are given compared to when hints are not given. However, on average, even when hints are provided, almost half of the students fail to correct their code so that it passes all the test cases.

While the first approach gave promising results as a first step towards providing formative feedback during autograding, it has its limitations, in particular, that it is labor-intensive. Thus, the second part of the dissertation explore an algorithmic approach that is much less labor-intensive. Using insights about student errors gained through applying the first approach, the second approach was designed to repair student programs and provide feedback on how to correct the program and on associated misconceptions. I start by compiling a set of programming repairs that are commonly needed to repair student programs across assignments. The repair set consists of modifications to both the statements and the program structure. These repairs fix errors that point to misconceptions about program structuring and other programming concepts. Each repair is used to modify the behavior of the student program until it matches the expected behavior, measured using a similar test suite as the first approach. Additionally, I provide a search procedure for finding a set of repairs that correct the behavior of a student program.

I apply this approach to multiple assignments from the Introduction to Computer Science course and report a higher success (i.e. fully correcting the program) rate than with previous approaches, an increase of 20% of the attempted student programs for specific assignments. Manual analysis of the repaired programs reveals that the repairs do not introduce stylistic issues such as dead code. However, for some assignments this approach is successful for less than half of the attempted student programs. More research is needed to understand how to provide accurate automated feedback to all student programs and for all types of assignments, and how this type of feedback specifically impacts learning and teaching.

Acknowledgements

I am grateful to my advisors and my collaborators for their knowledge and support.

Dedication

I dedicate this dissertation to my loved ones.

Table of Contents

Chapter 1

Introduction

As the global economy is moving more and more towards the use of computers [1], the need for computer science (CS)-trained professionals is on the rise [2]. This trend fuels "the surge in CS undergraduate degree production and course enrollment which is already straining program resources and causing further concern among faculty and administrators regarding how to best respond to the rapidly growing demand" [3]. Given the average teacher-to-students ratio in CS undergraduate courses can be as low as 1 to 73 [4], both learning and teaching computer science become challenging as students don't get enough feedback from instructors and instructors get unstructured and sporadic feedback from students about their learning progress.

> Repeated revision is central to mastery learning and depends crucially on rapid formative assessment and applying corrective feedback [5].

One example is the case of automatic grading of programming assignments. Programming assignments are tools of choice in many computing classes, giving students hands-on coding practice. However, as autograding systems are deployed to scale the grading of programming assignments in increasingly large classes, such as introductory courses CS1 and CS2 with hundreds to over a thousand students, providing meaningful feedback remains an open problem. Many autograding systems do not provide feedback besides a grade by default. Even when feedback is available, it often does not assist students in correcting errors and it does not address underlying misconceptions [6]. As a result, students receive minimal individual help with thinking about their assignments and on how to structure their programs. Meanwhile, by using autograders for grading student programs teachers fail to learn about the concepts and skills students are struggling with. This information can serve as the feedback loop in the process of teaching and it can guide teachers on how to adjust teaching to better serve students' needs.

This situation motivates a need for automatic solutions that mimic the process of hand-grading student programs [7]. We therefore seek an automated feedback generation solution that displays several properties:

1. **Accuracy.** A technique for automated feedback generation should provide feedback that correctly identifies and addresses the issues in students' programs.

2. **Usefulness to students.** A technique for automated feedback generation should help students with fixing their programs as well as help them with addressing the misconceptions corresponding to their errors.

3. **Usefulness to teachers.** A technique for automated feedback generation should provide teachers with information about students' programs, for example, an understanding of the concepts and skills that students are struggling with in order for them to design interventions to support their learning.

Concerns around the quality of CS education have motivated considerable previous research. Such previous research serves to inform students if their solution is correct or not (correctness feedback) [8, 9, 10], to point out what might be wrong with parts of their program (error-specific feedback) [10], to guide students on what to do next in order to make progress or fix their programs (next-step hints) [11, 12, 13, 14, 15, 16], to inform students on how to repair their programs (student program repair feedback) [17, 18, 19], and to propagate teacher's feedback to students with similar program structure or needing similar syntactic changes (propagated teacher feedback) [20, 21].

While this previous work serves a variety of useful purposes, it all, in one way or another, fails to possess one or more of the desired properties of a viable automatic feedback generation; most often they fail to provide teachers with an understanding of the concepts and skills that students are struggling with. Correctness feedback auto-generators lack in usefulness to students or teachers due to their summative assessment nature and lack of expressiveness. Auto-generated error-specific feedback and auto-generated next-step hints are centered around error-localization and providing repair hints, and they don't specifically address underlying misconceptions or link these repairs to their corresponding concepts and skills. Auto-generated student program repair feedback does not stimulate students to think abstractly about their program errors and sometimes the corrections are stylistically undesirable. Propagated teacher's feedback may be limited in terms of being useful to teachers, and even students because they organize the feedback in terms of program embeddings or syntactic transformations which are limiting, problem-dependent and not necessarily reflecting the domain knowledge of CS education. Moreover, students often have errors and misconceptions regarding program structuring and control flow, which many of the prior solutions don't specifically address.

To summarize, formative assessment of student programs and providing feedback are integral parts of CS education, but as the number of students increases, it can become time-consuming and challenging. The root cause of the challenge is the cognitive load on the educator to sift through students' code which may widely differ in strategy and implementation, as well as be riddled with misconceptions and errors. This situation motivates an automatic solution to alleviate at least some portion of the burden of student program assessment and feedback provision at scale; this dissertation explores different approach solutions to this problem.

1.1 CSF^2 and PR-CSF^2: A Mixed-Approach Framework for Providing Teacher's Feedback

This dissertation presents and evaluates the Concepts and Skills based Feedback Generation Framework (CSF^2) with two complementary methodologies for generating automated meaningful feedback to student programs. The second methodology which I refer to as Progam Repair - CSF^2 (PR-CSF^2) was developed as an extension based on knowledge gathered from applying the first methodology. The overarching framework is designed to be straightforward and to leverage existing practices such as code testing. These are two key features that enable teachers to easily adapt the framework to their needs[22, 6]. Both methodologies require a testcase set that describes the expected behavior from programs submitted for an assignment. Additionally, an inherent benefit of these approaches is that they provide information to the teachers about the concepts and skills that students are struggling with. Lastly, they can be used to design new grading schemes that grade student programs based on the demonstrated concepts and skills instead of passed testcases alone.

CSF^2's methodology focuses on linking errors in student programs to concepts and skills required for solving a programming assignment. It is data-driven and it relies on collecting and analyzing assignment submissions to generate hints that can be used during future semesters. It leverages the fact that student programs with similar issues fail similar testcases. PR-CSF^2's methodology concentrates on linking repairs in student programs to misconceptions students have about program structuring and other programming concepts. It leverages the fact the student programs with similar misconceptions require similar changes to the program syntax and structure.

My research is similar in spirit to other research efforts that support instructor feedback at scale. One important benefit of my approach is that it explicitly tries to bridge the teaching of abstract concepts with hands-on programming practice. This is accomplished by linking student programming errors to the concepts and skills taught in the class and program repairs to misconceptions about them. Furthermore, I provide a systematic way to tease out both concepts and skills and learn about error patterns in student programs. Additionally, I provide a program repair approach specifically designed for student programs in which repairs are mapped to misconceptions.

To summarize, the first approach is highly effective in providing feedback, but it is not always accurate, it requires substantial teacher support and a large data-set of previous student submissions for each assignment. Conversely, the second methodology necessitates less teacher support, it provides accurate feedback and it is generalizable - it can be used across multiple assignments, but it is not able to cover all the assignment-specific cases, in particular those student programs with design flaws. Finally, the two proposed methodologies are complementary. For new assignments, the second methodology can alleviate the cold-start downside of the other data-driven technique, and the first approach can be used to provide feedback to more student programs.

1.2 Evaluation Case Studies and Metrics

The case studies, benchmarks, and evaluation metrics were designed with the comparison to how teachers manually grade programming assignments and provide feedback in mind.

1.2.1 Usefulness to teachers

CSF^2 was designed specifically to assist teachers with understanding the concepts and skills students are struggling with, along with generating automated meaningful feedback to student programs. To demonstrate how teachers can use CSF^2 to identify concepts and skills students struggle with the most and how errors in student programs relate to those concepts and skills, I ran a case study in which I applied it to two programming assignments. Then I generated feedback for student programs submitted to these assignments and compiled a summary of the concepts and skills that the feedback maps to. Using the error patterns observed in student programs, I compiled a set of repairs that map to misconceptions and implemented PR-CSF^2. Similar to the previous case studies, I applied PR-CSF^2 to multiple assignments, and for the repaired programs set I show the distribution of repairs and explain the misconceptions they point to. This information can be used by the teachers to understand and further explore the relationship between the lecture material and student programming practice errors. For example, the instructors can easily tailor their teaching of concepts and skills based on the students' performance on the programming assignments.

1.2.2 Accuracy

To substantiate the claim that CSF^2 can be used to provide accurate feedback, I asked three undergraduate students who had previously taken our Introduction to Computer Science course to carefully review the submissions and automatically generated feedback, and assess its accuracy. I believe that having done the assignments themselves while taking the course gave these students a good perspective in the evaluation of the feedback. The feedback was rated accurate for 91.5% of the cases for one assignment and 87.5% of the cases for the other, with an inter-rater reliability score of 93%.

Additionally, I manually analyzed a subset of the repaired programs using PR-CSF^2, and observed that the programs repaired using program structure modification along with statement edits are more stylistically appropriate than those repaired only using statement edits.

1.2.3 Usefulness to students

To assess CSF^{2}usefulness to students, I used two approaches:

- *Empirical usefulness* aims to measure the usefulness of the feedback by analyzing differences in students' performance with the two programming assignments between semesters with and without feedback. We use two different measures:
  - *Efficacy measure* as the overall progress students made between their first and last submission measured as a difference in score.
  - *Efficiency measure* as the progress with each resubmission measured also as a difference in score.
- *Perceived usefulness* aims to assess the usefulness of the feedback as self-reported by the students using a survey.

1.3 Contributions and Outline

The primary contributions of this dissertation are:

- CSF^2 and PR-CSF^2, a mixed-approach framework for generating automated meaningful feedback to student programs as described in Chapters 3 and 8.
- I demonstrate how the framework can be used by teachers to learn which concepts and skills students are struggling with and to design alternate grading schemes in Chapter 5. I also report on the overall impact using CSF^2 has had on the CS1 course development at a state public university.

- Empirical evidence showing the feedback generated automatically using the framework is accurate in Chapter 6.
- Empirical evidence suggesting that the feedback helps students with correcting their programs in Chapter 7.
- Empirical evidence substantiating the claim that students find the feedback useful in Chapter 7.
- A program repair method that allows for program structure modification in addition to statement level edits described in Section 8.3.
- Empirical evidence showing that PR-CSF^2's repair method can repair more programs and that in some cases the repaired programs are more stylistically appropriate than equivalent tools that only perform statement-level repairs in Chapter 9.

In addition to the chapters and sections corresponding to each of the contributions listed above, Chapter 2 discusses related work and Chapter 10 concludes the dissertation.

Chapter 2

Background and Related work

Providing feedback for improving student's learning, and understanding obstacles to students' learning are dominant concerns of the CS education community. These concerns motivate the work presented in this dissertation. In this chapter, I describe some background and related work required to understand my work and its place in the context.

2.1 Why Do We Need to Provide Automated Feedback to Programming Assignments?

Giving feedback to students is such a universally adopted practice that one may find the question surprising. After reviewing the body of research on feedback in educational contexts, Shute [23] concludes that feedback is generally considered crucial to improving knowledge and skill acquisition in educational contexts, in particular, formative feedback which Shute further defines as *"information communicated to the learner that is intended to modify his or her thinking or behavior to improve learning."* Furthermore, according to Ambrose et al. [24], *"goal-directed practice coupled with targeted feedback are critical to learning"* is one of the core seven research-based principles for smart teaching. One study [25] shows that students use feedback for multiple purposes, such as decomposing the problem in smaller chunks, arriving at the correct answer, verifying their answers, and comparing their solution with an expert solution.

The need for computational thinking skills is ubiquitous [26]. As a result, enrollments in computer science courses have skyrocketed among majors and nonmajors [27]. Consequently, teachers need systems to assist them with their teaching. Programming assignments are tools of choice in many computing classes, giving students hands-on coding practice. Enabling teachers to provide feedback to students' code presents the opportunity to improve students' learning. For example, CodeOpticon [28] enables instructors to monitor multiple students simultaneously as they code and to assist them online.

Since feedback given immediately is most useful [29], for the rest of the related work I will primarily focus on those that fit this criterion. Generally, feedback provided by automated systems ranges from correctness feedback to peer or teacher feedback. Next, I summarize related work in each of these categories. For an in-depth and comprehensive comparison of various types of tools for automated programming hint generation, I recommend a recent survey on the subject [30].

2.2 Types of Automated Feedback to Student Programs for Students

2.2.1 Automated Correctness Feedback

The most basic form of automated feedback to programming measures whether the behavior of students' code is as expected. Most commonly this is done using a testcase set [9]. These tools are often referred to as autograders because they return a grade based on the student's code performance during testing. The most recently developed autograders take advantage of contemporary web-based technologies. In our Introduction to Computer Science course we have adopted Web-CAT [8] and Autolab [10], two web-based autograders. To employ these tools, the instructor provides a grading scheme and an executable file which the systems use to grade the students' code. Grades are generally calculated as a weighted sum of the passed testcases. In some cases, students are provided with feedback on the test results as well. Although grading is important, the instant binary passed-or-failed feedback has been correlated to the reduction in student engagement and increase in cheating [31]. Our experience with providing binary feedback to individual testcases has been that students try to game the system individually or collectively to figure out the testcases used for testing their code.

Moreover, using testcases results to grade students' code may not be the best solution. For example, a simple mistake may result in not passing any testcases. Singh et al. [17] offer an alternate assignment-independent approach to grading by introducing a problem-independent grammar of features. They use supervised learning to train a classifier on teacher-graded examples. The classifier maps new student code submissions to grades. Similarly, my work explores the extension of an autograding system to design alternate grading schemes by using testcases results patterns to identify the concepts and skills students are struggling with, and grade based on these concepts and skills rather than based on the testing outcomes.

2.2.2 Automated Code Quality or Style Feedback

Code quality or programming style refers to good programming practices that expert programmers agree on. One such practice is to write simple code. AutoStyle [32] clusters correct submissions using the ABC software metric and propagates hints to each cluster focused on writing simpler code. Adhering to good programming practices is a concern of the broader programming community, and many tools have been developed for analyzing code and detecting violations to these universally accepted good programming practices [33]. Most of the issues that these tools report on have to do with meta-aspects of programming such as formatting, documentation, and readability. However, some poor programming practices can lead to bugs [34, 35]. Using two of these static analysis tools to provide feedback to students, Edwards et al. [36] found that the most common feedback generated this way was on formatting and documentation which lead students to think that all the issues reported were cosmetic.

2.2.3 Automated Program-Repair Feedback

Program-repair feedback points where and how to change students' code so that it passes all the testcases. Autograder [37] requires a reference solution and an error model consisting of possible corrections to errors that students might make. Then, it tries to find the smallest possible number of corrections to the students' code using a constraint solving program synthesis approach. Qlose [19] is similar to Autograder, except it does not require an error model. Instead, it tries to find the best correction by minimizing both the syntactic and semantic distances to the reference solution. Sarfgen

[18] is a data-driven program repair framework that takes advantage of a large number of available student submissions and tries to find minimal fixes by aligning incorrect programs with similar programs that are correct. Rafazer [38] is an approach for learning syntactic transformations from examples of statements or expressions before and after the change. It then uses these transformations to automatically repair programs. One extension to Rafazer generates hints based on the synthesized correct program automatically [39]. Although such systems have shown great potential in generating the program repair closest to what a programmer would suggest, this kind of feedback does not cause students to think abstractly about their solutions.

2.2.4 Automated Next-step Hints

Systems that generate next-step hints use a step-by-step approach to guiding the students towards the correct solution. Hint Factory [13] uses a data-driven approach for hint generation that first learns from successful solutions paths from previous students and uses this information to guide new students in similar states. A problem with this approach is that there is no guarantee that new students are going to end up in the same states as previous students. In other words, the space of states extracted from previous students is discrete and future student states may not hit any of the states in that discrete space. A follow-up approach, the Continuous Hint Factory [14] tries to solve this problem by using the weighted sum of previous edits to select the best next step. A similar approach, ITAP [40] deals with this problem of reducing the variability of program states by using canonicalization [41]. During canonicalization, students' code is repetitively transformed until it reaches a standard form. Gerdes et al. [15] define a strategy language that specifies how parts of a solution may be built up from others and uses it to generate the next steps rather than using existing steps from peer data. Step-by-step approaches have the inherent benefit of suggesting student-inspired fixes, but require sequences of student programs with incremental repairs to learn the fixes from. These techniques complement my research.

2.2.5 Automated Example Feedback

Gross et al. [42] and Zhi et al. [16] are two data-driven approaches for providing examples to students as feedback. The feedback examples in the first approach are complete student solutions closest (according to a distance metric) to the students' current code. In the second approach, the feedback examples are partial solutions focused on addressing students' current programming goals. This type of feedback is complementary to the kind of feedback I aim to provide in my research.

2.2.6 Automated Error-Specific Feedback

Error-specific feedback tells the students what might be wrong with their code. One approach uses the reference solution as the problem model and provides feedback on the root cause of the bug in students' programs by comparing their execution traces [43]. Other approaches leverage programming knowledge models or error models and use testing to check for them in students' code, for example, constraint-based tutor (CBT) [44, 45, 46] and intelligent tutoring systems (ITS)[47, 48]. While very efficient for certain programming assignments, these approaches assume that the solution space for the assignment is small. However, it has been shown that programming assignment problems may vary greatly in the number of solution strategies and implementations they accept [49, 50, 51]. Additionally, assignment-dependent models like the ones used by ITS require extensive expertise and time to develop for each assignment regardless of its complexity, which does not scale well with the number of assignments [52].

One approach mentioned above, J-LATTE [45], a CBT for java, offers a concept mode option that has a similar goal of providing conceptual feedback as CSF^2, but it's limited by its design choices. In J-LATTE, concepts are mapped to statements and blocks such as loops and if-blocks. This solution works only if students are constrained to implementing a specific program structure. In the case of autograders, that is generally not the case. For so-called *"open-ended programming problems"* [53], students' programs commonly vary in their structure and syntax [49].

2.2.7 Automated Peer Feedback

Providing peer feedback is a debatable and challenging solution. While some claim that students learn best from each other because they share similar learning experiences by taking the same course [54], others point out challenges such as the lack the motivation to assess others' work faithfully and fairly [55] and propagation of misconceptions and misinformation among students. One approach [56] offers a solution to this last issue by allowing students to rank the feedback. This research direction complements my work.

2.2.8 Automated Teacher Feedback

Many approaches in this research space use clustering based on different program features. Nguyen et al. [57] cluster functionally equivalent but syntactically distinct code phrases using probabilistic semantic equivalence. Codewebs [57] leverage the statistical properties of a large number of student submissions by extracting patterns that can be used for refining the clustering and providing improved feedback. Piech et al. [20] cluster submissions using neural networks to learn program embeddings. Kaleeswaran et al. [58] cluster dynamic programming (DP) submissions according to their solution strategy and map feedback to instructor-validated submissions for each cluster. Foobaz [59] cluster student code and report common and uncommon student choices on syntax and style. MistakeBrowser and FixPropagator [21] learn in real-time syntactic changes teachers make to students' code. The feedback associated with the change is then propagated to future students with similar syntactic errors. MistakeBrowser learns transformations from incorrect to correct code from teachers fixing bugs in incorrect student submissions. When examples of fixes are not available, its counterpart FixPropagator asks teachers to fix students' code and write feedback which then gets fed back into MistakeBrowser. Instead of clustering based on program features, HelpMeOut [60] clusters bug fixes by compiler error or runtime exceptions. My mixed approach aims to provide automated feedback of this kind. However, there are some important differences between my approaches and previously used approaches as I describe next.

While all these solutions have been proven to be very effective for specific benchmarks, each approach falls short in some aspects. Some techniques work for only a subset of the issues in students' programs such a syntactical transformations [21] or force students to write their programs in a certain way (for example use specific variable) [20]. However, the main problem is that most approaches are not *"discoverable"* or *"expressive"* by which I mean it's not clear what those features represent in terms of students' learning and misconceptions. Furthermore, teachers are not able to easily get a summary of students' performance on demonstrating the knowledge required by the programming assignment and what concepts and skills they are still struggling with.

One feature in Web-CAT can be configured to work similarly as CSF^2, but only for a limited subset of cases. It allows teachers to write feedback to each testcase that is provided to students when their code fails that testcase [61]. However, writing feedback per testcase is not straightforward for many programming assignments. For example, for problems with complicated program structures, a single testcase does not offer much information about the root cause of the issue. Combining information from multiple testcases is a much more powerful technique for identifying issues in students' programs. For example, comparing faulty and successful executions is a major research direction in fault localization [62, 63, 64]. My first approach leverages patterns of passed and failed testcases similarly to identify issues in students' code. My second approach uses a fault localization technique [62] to prune the search space of modified versions of the student program and to guide the search process.

2.2.9 Discussion: Desired Automated Feedback vs the State-of-the-Art

Providing automatically generated feedback is a very attractive idea but also a very challenging one. Commonly, the better the feedback, the more requirements the approach has. These requirements span from expertise, time, and effort from the teacher, to data repositories containing student code.

Most approaches for providing automated feedback to programming assignments can be classified as *generic* or *assignment-independent* and *assignment-dependent*. There are some important trade-offs to consider between the two approaches. Generic models may require less work per assignment, but they may not be able to identify problem-dependent errors and lack the deep domain knowledge of the teacher. Assignment-dependent models like the ones used by ITS require extensive expertise and time to develop for each assignment regardless of its complexity, which does not scale well with the number of assignments [52]. The two options lie at two extremes. One approach maximizes coverage of the errors while the other minimizes the amount of effort and expertise. The quality of the feedback produced is commonly proportional to the amount of effort and expertise each approach requires. Both sets of goals are important and my research draws inspiration from both types of approaches. Through my work for this dissertation, I seek to explore the space between these extremes and to design approaches that explore the balance between them.

2.3 Program Repair Outside Automated Feedback Generation

Program repair is a well-established and active research area that intersects with other areas such as debugging, program synthesis, program analysis, recommender systems for code completion, code translation from one programming language to another, to name a few. For a detailed analysis of all these research areas and how they relate to each other, I recommend a survey by Allamanis et al. [65] for further reading. In this section, I offer a brief overview of the existing approaches and explain how they relate to my research.

Some program repair techniques take advantage of the fact that all code is transformed into logical expressions that are executed by the computer. One approach uses Boolean Satisfiability (SAT)-based techniques to represent and find a fix to buggy code [66], while another uses model-based fault localization and then produces corrections to the right-hand side of the faulty assignment statements using Satisfiability Modulo Theories (SMT) reasoning [67]. Generally, these approaches perform only a subset of all the possible corrections since they leave out a lot of the programs' features during their analysis. While such techniques are very effective in areas such as compilers research, they are not as effective in other areas in which the result is communicated back to humans (in particular non-experts). Allamanis et al. [68] keenly explains that this is due to code's intrinsic quality of communicating with both humans and machines, which he refers to as the *"bimodal nature"* of code written in high-level languages such as Java.

Another class of program repair techniques uses probabilistic models learned from code. For example, Prophet [69] first learns a probabilistic, application-independent model of correct code from a set of successful human patches obtained from open-source software repositories and then uses it to generate candidate patches to buggy code. Moreover, there are many other machine learning techniques that work similarly, many of them initially used in natural language processing (NLP). The main issue with these approaches for automated feedback generation is that they are often not *"discoverable"* by which I mean it is difficult to reason about how they arrived at one solution. This ability of reasoning about one solution path vs another is a desirable feature for automated feedback generation because it can be used to classify student errors and understand the misconceptions attached to them.

Lastly, another class of program repair techniques leverages information about code syntax to modify a buggy program until it's correct. Each approach uses a model for modifying the buggy program and a search procedure for the correct program in the space of candidate programs. Genetic programming [70] and program mutation [71] are two such approaches and they are similar to my program repair approach. However, there are some key differences between these approaches and my program repair approach which I describe in more detail in Chapter 8.

2.4 Automated Feedback about Student Programs for Teachers

Hattie et al. [72] points out the dual nature of feedback in the education context and describes it as an ongoing process of giving and receiving feedback between the teacher and the student. Huang et al. [49] cluster student submissions using the abstract syntax tree (AST) edit distance and generates a visual representation of the student programs space. While the results of this approach are informative, they are not structured enough for teachers to use them in developing interventions for supplementing students' learning. OverCode [73] offers a web-based solution that aims to automatically provide teachers with a structured view of the students' code. While the graphical interface makes it very easy to explore variations in student solutions, teachers still need to do a considerable amount of work to understand the knowledge components that students struggle with. By using CSF^2, teachers automatically get a summary of the concepts and skills that students struggle with at the end of each assignment.

2.5 How Does the Automated Feedback Impact Students' Learning and Behavior?

Generally, students find the automatically generated feedback to programming assignments to be useful [60, 74, 75]. Some studies found that students submit fewer submissions [75] and complete more problems when feedback is provided [76, 19]. In the case studies that I conducted, students are allowed to submit up to three times without penalty and after the third submission, the penalty is 5% of their grade with each new submission. Consequently, I could not test this hypothesis. A study on Web-CAT's hints per failed testcase [77] reveals that providing feedback in this way appears to advance students' learning behavior from trial-and-error to reflection-in-action in a junior-level course. It is not clear that this result can be replicated for beginner CS1 and CS2 courses. Another study [78] reports a 10% increase in performance on open-ended programming problems and an acceleration in performance on a multiple-choice quiz. My overarching approach to automatically providing feedback to programming assignments which I later describe in Chapter 3 is very similar to their proposed Misconceptions-Driven Feedback (MDF) approach. The two approaches have been developed in parallel independently. Marwan et al. [79] find that code hints with textual explanations significantly improved programming performance. Moreover, when these hints are combined with self-explanation prompts they improve performance in a subsequent post-test task with similar objectives. Hints with textual explanations are very similar to the kind of feedback I aim to provide with CSF^2. Therefore, these results build confidence in my results and the mixed-approach framework I propose.

However, the combined body of research on the effects on students' learning behaviors and overall learning is inconclusive [23]. Large gaps and questions remain, for example, there is no consensus on whether students spend less time when hints are provided compared to when they are not [80, 81]. One study [82] observes that students use feedback to game the system instead of attempting to learn from the feedback. One reason for the conflicting findings may be because feedback does not work to improve knowledge in isolation [72, 83], it builds on existing knowledge.

The improved performance metric and other similar metrics that prior studies report on don't necessarily measure improved learning. We need approaches and metrics grounded in cognitive theories [84] to understand what are the basic units of knowledge in programming, how to identify them during programming practice, and how to use them in the learning assessment process. Being able to structure knowledge in this fashion, and to refer back to it time and again throughout instruction gives students a knowledge core to build on. Concomitantly, teachers can use it to assess students' knowledge, plan programming assignments and organize instruction similar to how learning plans and learning goals work. I designed CSF^2 with these ideas in mind. Preliminary results I report on in Chapters 5, 6 and 7 show promise. However, further research is needed to substantiate the claims about the impact feedback generated in this manner has on students' actual learning.

2.6 Compiling Programming Knowledge for Automated Feedback Generation

The ability to reason about a specific domain is at the heart of teaching and learning. The process of providing feedback starts with using a student's program to build a model of the students' reasoning about a specific problem. Then that model is used in the context of the course's domain knowledge to generate and return a grade and, potentially, feedback to students. Computer programming is a complex, ill-structured design domain [85] and therefore building a generic all-encompassing domain model for programming knowledge is challenging. My proposed mixed approach aims to build a domain knowledge model that bridges instruction with programming practice and then use it to provide meaningful feedback to students about their programs.

In the case studies that I report on later in my dissertation, I leverage knowledge about errors and misconceptions in students' programs acquired by carefully analyzing vast datasets of student programs from large student groups. These large groups of students share common errors and misconceptions as has been observed in prior research [37, 57, 20, 56]. I also draw inspiration from a couple of studies. In particular, from Mayer et al.[86] I make use of the programming skills that they show are connected to thinking skills used in problem-solving. From Brennan and Resnick [87], I borrow from their three categories of computational thinking components: computational concepts, computational practices, and computational perspectives. Concept inventories [88] can also be used for building domain knowledge models.

Knowledge maps for programming assignments are a way to abstract students' knowledge and misconceptions as reflected by their programs. Programming assignments vary in their complexity and the number of solution strategies they accept [50, 51]. Commonly, the more complex the assignment is, the larger the solution space is, and with that, the more ways in which students can introduce errors in their programs. CSF^2 intends to reduce the space of errors signatures, by mapping them to the knowledge required to complete the assignment. To demonstrate this, the assignments that I use in my case studies accept multiple solution implementations and multiple solution strategies which correspond to classes 2 and 3 [50, 51].

Chapter 3

Concept and Skills based Feedback Generation Framework (CSF^2)

3.1 Overview of CSF^2's Design

Many autograding systems, such as Web-CAT [8] and Autolab [10] do not provide feedback by default. Even when feedback is available, it often does not assist students in correcting errors and it does not address underlying misconceptions [6]. Through my research with CSF^2, I aim to answer these specific research questions:

1. Is it possible for instructors to identify patterns of passed and failed test cases that point to logical errors in the students' code? Furthermore, can I partition the students' code based on logical errors using these patterns of passed and failed test cases? (Chapter 4)

2. How accurate are these patterns of passed and failed test cases for partitioning the students' code? (Chapter 6)

3. If it is possible to partition the students' code, can instructors write hints for an autograding system to give meaningful feedback that helps students identify their errors and make progress? (Chapter 7)

4. How are the hints associated with each partition perceived by students and instructors? (Chapter 7)

I propose a methodology that is similar in spirit to a previous approach [21], but, instead, focuses on linking errors to the concepts and skills required for solving a programming assignment. My work leverages code testing, which is the most common method for grading and assessment in programming. Since concepts and skills are shared among assignments, my approach offers the potential for reusing some of the work done in previous assignments.[1] My approach relies on collecting and analyzing assignment submissions to generate hints that can be used during future semesters. More specifically, when designing an assignment using my approach, the instructor explicitly defines the set of concepts and skills that students need to master to complete the assignment. After the assignment has been written, a comprehensive test suite is developed to tease out programming misconceptions and to test the correctness of the code for grading. Next, the assignment is released and student submissions are collected. The instructor runs the test suite against the submissions and manually inspects sets of submissions with the same outcome pattern (which I call the *signature*) to identify errors. In this process, the test suite can also be refined as needed or desired. Finally, the instructor maps the errors to specific concepts and skills and writes hints that are designed to guide the students toward correcting their code and improving their understanding of the corresponding concepts and skills. The obtained signatures can be used to produce a classifier that automatically categorizes erroneous submissions, enabling the autograding system to provide hints as feedback to the student code submissions.

The above methodology encapsulates two key ideas: (1) the instructor can identify high-level logical errors by inspecting sets of submissions with the same error signature if the error signatures are generated using a comprehensive and well-designed test suite, and (2) the mapping of errors to concepts and skills can produce hints that encourage students to think about their code conceptually, as opposed to suggesting local, code-specific changes that can lead to highly convoluted solutions, such as solutions containing unnecessary nested conditional statements. Understanding common errors in light of the concepts and skills they map to can also help the instructor adjust her classroom teaching.

I applied our proposed methodology to two assignments in our Introduction to Computer Science course and collected a large number of submissions for each assignment during three semesters, Spring 2016, Spring 2017, and Spring 2018. I designed a test

[1]Supporting the evolution of assignments while still making use of the collected data is an important extension that I intend to explore in the future.

suite for each assignment and used submissions from Spring 2016 to learn classes of common errors, produce classifiers for the automatic error categorization of future submissions, and writing hints. Then, I used the classifiers to attach hints to the erroneous submissions from Spring 2017. Four researchers manually reviewed the results of the classifiers and found that over 91% of the hints for the first assignment and over 87% of the hints for the second assignment fully captured the errors in the corresponding submissions. These percentages rose to over 96% for both assignments when I counted hints that partially captured the errors. Based on these promising results, starting with Fall 2017, I deployed the error categorization and corresponding hints for the two assignments. I compared submissions from two semesters, one when hints were not provided (Spring 2017) and one when hints were provided (Spring 2018), and found that when hints were provided, students submitted more often, more students made progress, and the overall progress toward completing the assignment was faster. For example, the percentage of students who completed the assignments after an initial erroneous submission was three times greater when hints were given compared to when hints were not given.

Moreover, during Spring 2018, I asked students to complete a survey regarding the usefulness of the hints. I report the students' responses and their comments about the hints in Section 7.2.2. Lastly, I asked the course instructors for feedback about the hints. I found that students and instructors thought that the hints were helpful, but also that they could be improved, in particular the information provided in the hints as well as their wording. These are highly debatable subjects that go beyond the scope of our current research.

3.2 CSF^2 step-by-step

CSF^2 aims to encourage teachers to design programming assignments based on the concepts and skills that students are required to master having taken the class. These concepts and skills are used to generate hints for assisting students in correcting errors. The approach is described as a sequence of steps, but the ordering can be modified as discussed in Section 3.3. The proposed steps are as follows.

1. Carefully list the set of concepts and skills that students need to master.

2. Write the assignment to evaluate these concepts and skills.

3. Design a test suite to assess student submissions. A full path coverage of a reference solution is typically a good start (but will need expansion). To test a submission, each test case should output a code representing the outcome of the test.

4. Release the assignment and collect student submissions.

5. Automatically run the test suite against the collected submissions and group submissions into "buckets" based on their outcome signatures, where a signature is the concatenation of the code's output by all the test cases in the test suite. Each signature may indicate one or more logical errors. I hypothesize that submissions with the same signature are likely to have similar logical errors.

6. Manually inspect each bucket of submissions to see whether subsets of submissions have different logical errors. If this is the case, then add test cases to the test suite to separate these subsets into different buckets. The list of concepts and skills may also need to be refined (for example, to include a concept that has to be mastered for the completion of the assignment but was accidentally omitted in 1).

7. Repeat 5 and 6 until submissions in each bucket have the same logical errors.

8. Map the errors identified for each bucket to concepts and skills. Then, manually inspect and combine buckets of submissions with the same errors or knowledge deficiencies. I hypothesize that mapping errors to concepts and skills will help instructors reduce the number of hints that will need to be written, as well as write hints that provide conceptual guidance rather than very specific code changes.

9. Write a hint for each bucket. The outcome of Steps 8 and 9 is a classifier, manually trained on past student submissions, that maps the outcome signatures of a well-designed test suite to meaningful hints.

10. When the assignment is run again, the autograding system can use the classifier to automatically categorize errors and provide hints for submissions that fail one or more test cases.

While the work in this paper is focused on generating meaningful autograding feedback, the above process is also useful in gaining a better understanding of the students' errors and misconceptions. The instructor can use this information to improve the quality of classroom teaching, for example, in identifying concepts and skills that should be reinforced, as well as adjusting the teaching of more challenging concepts and skills.

As described above, CSF^2 is most useful when an assignment is given during multiple semesters, with previous submissions used to generate and improve hints for subsequent semesters. Currently, CSF^2 does not directly support the evolution of an assignment over time (for example, changes to the assignment to discourage cheating or to improve the assignment), although I have successfully experimented with changing one of the assignments studied in this paper to an isomorphic assignment while still making use of analysis results from previous submissions (Section 3.3). This is an important challenge that I plan to address in the future.

3.3 Discussion: Modifications to CSF^2

Top-down and bottom-up are two well-known strategies of information processing and knowledge ordering, and our framework can be modified to employ either of them. The top-down approach starts with the whole problem and decomposes it into individual steps. The bottom-up approach pieces together individual parts into bigger parts. The process described in Section 3.2 is meant to be a guideline for best practices. As written, it describes a top-down approach to the design of assignments using specific sets of concepts and skills. The advantage of this approach is that assignments are carefully and systematically written to target specific course material. However, in practice, and as I saw in our case studies, instructors may already have the assignments written and used. In these situations, a bottom-up approach can be applied. This approach has a few advantages: previously collected submissions can be used to guide the derivation of the set of concepts and skills that map to the specific assignment, it provides a way to design or improve the test suite, and it gives instructors a way to generate the error classifiers and hints.

Much of the manual work done for this research was labor-intensive because I manually reviewed all the erroneous submissions for *PayFriend* and *TwoSmallest*. As an alternative to reviewing all the submissions, it may be sufficient to use random subsets of varying sizes to determine how the error categorization and hint generation change with sample size. I will consider this analysis in my future work.

Finally, while the manual inspection of assignments is labor intensive, it is possible to get help from advanced undergraduate students when CSF^2 is applied to early computing classes. For example, three undergraduate upperclassmen helped with my case studies.

Chapter 4

Application of CSF^2 to Two Programming Assignments

In this chapter, I describe the application of CSF^2 to two programming assignments in the Introduction to Computer Science course at Rutgers University - New Brunswick. Even though CSF^2 proposes that an instructor starts with a list of concepts and skills when designing an assignment, for this case study I start with existing assignments because all assignments are built on an implicit list of concepts and skills and I already had access to a large number of student submissions collected across several semesters before the start of this research. In essence, I reversed the order of Steps 1 and 2 in CSF^2, and extracted the concepts and skills from existing assignments.

4.1 Steps 1 and 2: Compiling the Concepts and Skills List

I studied two assignments, *PayFriend*, a class 2 assignment [51], which means that it has one solution strategy with multiple possible implementations and *TwoSmallest*, a class 3 assignment, which means that it has multiple solution strategies, as well as multiple possible implementations for each strategy. *PayFriend* asks students to compute the fee associated with making an e-payment when given a tiered fee structure, with different fees for four payment ranges. *TwoSmallest* asks the students to read a sequence of floating-point values that starts and ends with a sentinel value and output the two smallest values in the sequence.

Each assignment requires that the solution be implemented as a method with a prescribed signature. Students are also asked to submit code in a specific format, including predefined file and class names and to omit all `package` and `import` statements. Failure to follow any of these instructions results in compilation errors and a score of zero.

When I started this research, *PayFriend* and *TwoSmallest* had already been assigned

Table 4.1: The mapping of common errors to concepts and skills for two programming assignments, *PayFriend* and *TwoSmallest*. The base score is used for calculating grades as discussed in Section 5.3.

Code	**Error**	**Base Score**	**Concept or Skill**
Both assignments			
COMP	has compilation errors	0	writing code that compiles
INS	has errors regarding the required formatting, for example, incorrect file name	5	following instructions
IO	has IO errors, for example, wrong types of inputs or outputs	10-30	data representation, following instructions
INF	uses infinite loops	5-40	control flow
PayFriend			
CF	outputs only in some branches	50	control flow
COND	uses incorrect conditional statements	50	translating word problems into conditional statements
FORM	uses an incorrect calculation inside an interval	50	translating word problems into formulas
TwoSmallest			
SEQ	reads and processes incorrectly a sequence of values	40	data representation, following instructions
INIT	initializes min values incorrectly	40	algorithmic thinking
UPDT	updates min values incorrectly	40	algorithmic thinking

during several semesters. I worked with the lead instructor to determine the concepts and skills corresponding to these assignments. Some of these concepts and skills are shown in the right column of Table 4.1.

4.2 Step 3: Designing the Test Suites

I developed a reference solution for each assignment and designed 13 test cases for *PayFriend* and 20 for *TwoSmallest* that led to a full path coverage of the corresponding reference solution. Next, I considered more challenging inputs, especially for novice programmers. For example, it is well known that many programming bugs involve incorrect handling of boundary values. Thus, for *PayFriend*, I designed test cases with input values close to the tier boundaries, as well as values from the middle of the tiers. The process of designing a test suite was iterative (as discussed in Steps 5–7 of CSF^2).

Table 4.2: Summary of the data sets used in our study. During each semester, students were allowed to submit each assignment without penalty up to five times during Spring 2016 and up to three times during Spring 2017 and 2018. The penalty for each subsequent submission was 5 points.

	PayFriend			***TwoSmallest***		
	Spring 2016	**Spring 2017**	**Spring 2018**	**Spring 2016**	**Spring 2017**	**Spring 2018**
Autograder used	Web-CAT	Autolab		Web-CAT	Autolab	
Number of submissions	1152	719	936	1339	870	1071
Number of students who submitted at least once	511	432	487	488	423	476
Average number of submissions per student	2.3	1.7	1.9	2.7	2.1	2.3
% of students who submitted at least twice	62.0%	36.4%	53.0%	72.3%	54.1%	60.1%

After all the refinements were made, the test suite for *PayFriend* contained 20 test cases and the one for *TwoSmallest* contained 30. When these assignments were used during previous semesters, *PayFriend* and *TwoSmallest* were graded using 10 and 7 test cases, respectively.

4.3 Step 4: Collecting Student Submissions

Student submissions for the two assignments were collected during the Spring 2016 semester using Web-CAT [8] and during the Spring 2017 and 2018 semesters using Autolab [10]. Table 4.2 shows information about our data sets. Each submission included anonymized student information, a time stamp, and the student's code. In this study, I looked at the submitted code—all submissions were anonymized by removing all information, including comments, other than the actual code—and used the anonymized student information to link submissions from each unique student in a semester (students can submit each assignment multiple times). The submissions from Spring 2016 were used for Steps 5–9 of CSF^2 to build an error classifier and generate a set of hints for each assignment (as detailed below). The submissions from Spring 2017 and Spring 2018 were used to evaluate the accuracy of the classifiers and the effectiveness of the hints as described in Chapter 7.

4.4 Steps 5–7: Refining Tests and Partitioning Submissions

I used the test suite developed in Step 3 to test and generate the outcome signature for every code submission. Then, submissions with the same signature were grouped in the same bucket. Each signature could indicate one or more logical errors. A bucket could be mapped to two or more independent errors with their associated hints, but all the submissions in the same bucket needed to have the same errors so that a meaningful corresponding hint could be generated. If for any given bucket, some of the submissions had one error while others had a different error, we added test cases to separate submissions with different errors into different buckets.

I manually inspected the students' code in each bucket to determine the main reason why the code failed one or more test cases. For buckets containing submissions with *different* knowledge deficiencies, I refined or extended the test suite to further partition the buckets. I iterated through Steps 5–7 once for *PayFriend* and several times, making small refinements each time, for *TwoSmallest*. I found that iterations with small refinements were easier to think about. By the end of the process, I added 7 additional test cases for *PayFriend* and 10 for *TwoSmallest*. The final test suites resulted in 109 non-empty buckets for *PayFriend* (using 20 test cases) and 137 non-empty buckets for *TwoSmallest* (using 30 test cases).

4.5 Steps 8 and 9: Combining Buckets and Writing Hints

Next, I manually mapped the main reason for code failure in each bucket to a deficiency in a concept or skill as shown in Table 4.1. As already mentioned, I found that many of the buckets were different manifestations of similar knowledge deficiencies. I merged buckets accordingly, leading to 8 "super-buckets" for *PayFriend* and 7 for *TwoSmallest* (Table 4.3). Clustering student submissions for assignments in classes 2 and 3 [51] is a particularly challenging task because their solution space can be large. Since unit testing mostly focuses on the functionality of the code rather than its style, I was able to cluster stylistically different student submissions in the same bucket.

I used the clustering of the student submissions from Spring 2016 and the signatures

Table 4.3: Final buckets (classes) of errors for *PayFriend* and *TwoSmallest*. The table shows statistics for errors found in Spring 2017 submissions together with the accuracy of the error classification and hints. The latter is discussed in Chapter 6.

Error Codes	**Automatic Classification**	**Correct**		**Partially Correct**	
PayFriend					
	Total Count	**Count**	**% of Total**	**Count**	**% of Total**
COMP	34	34	100%	0	0%
INS	111	111	100%	0	0%
IO	119	114	95.8%	3	2.5%
INF	7	4	57.1%	3	42.9%
CF	38	26	68.4%	4	10.5%
COND	44	39	88.6%	4	9.1%
COND, FORM	91	82	90.1%	9	9.9%
FORM	83	72	86.7%	4	4.8%
Total	**527**	**482**	**91.5%**	**27**	**5.1%**
TwoSmallest					
	Total Count	**Count**	**% of Total**	**Count**	**% of Total**
COMP	39	39	100%	0	0%
INS	105	104	99%	1	1%
SEQ	51	47	92.2%	4	7.8%
INIT	67	56	91.8%	5	8.2%
UPDT	158	129	87.2%	19	12.8%
SEQ, INIT	157	137	91.9%	12	8.1%
SEQ, UPDT	176	145	85.8%	24	14.2%
Total	**767**	**671**	**87.5%**	**65**	**8.5%**

for each bucket to develop classifiers for every assignment. Then, I ran the classifiers on the student submissions from Spring 2017 and manually evaluated their accuracy as described in Chapter 6. Finally, I wrote a hint for each bucket. This hint was given to students after every submission, based on the result of the assignment's classifier. The next section provides examples of common errors and the corresponding hints given to students during the Spring 2018 semester.

4.6 Examples of Common Errors and Hints

Example 1. For *PayFriend*, common errors include incorrect conditional expressions leading to incorrect answers for boundary values and incorrect formulas for one or more fee tiers leading to incorrect answers for entire tiers. It is useful to differentiate between the two errors when giving students hints. I was able to make this distinction using the combined outputs of multiple test cases.

More specifically, if a submission fails test cases with input values inside one tier, I, but passes test cases with input values in other tiers, it is likely that the code does not correctly calculate the fee for tier I. This signature leads to the following hint for tier $(\$100, \$1000)$: *"It seems that you are not correctly calculating the fee for payments in the range (100, 1000). Review the assignment instructions, check that your formula for computing the fee is correct, then follow the steps used in the calculation of the fee in your code and make sure that they implement the correct formula."*

On the other hand, if the submission passes test cases with input values inside I, but fails test cases with inputs near the upper or lower boundaries of I, the code likely uses incorrect conditional expressions. This may arise from a misunderstanding of conditional statements and expressions, or a misunderstanding of the assignment instructions, or both, hence the mapping to the skill *translating word problems into conditional statements*. For example, discriminating between $\geq$ and $>$ in a conditional expression requires understanding boundary values and how they differ among data types. For integers, $x < 100$ is equivalent to $x \leq 99$, but this is not true for real values. Thus, I wrote the following hint for this class of errors: *"It seems that you did not split the input intervals correctly, where some values at the boundary between intervals may have been included under the wrong formula/rule; that is, your conditional expressions may be incorrect, for example, you may have ≥ 101 instead of > 100 which are not equivalent expressions for double values."*

Example 2. For *TwoSmallest*, given the material that has been taught in class, most students develop algorithms that have two major steps: (1) initialize two variables for storing the two smallest values, and (2) read the input sequence and update the variables correspondingly. Many students do not consider what values they should use to initialize the variables and end up using improper initial values such as 0. By using the results of several test cases with input values that are positive, negative, and mixed, I can tell whether or not a submission has this mistake. I map this error to *algorithmic thinking*, which reminds us to view the error in light of the student's algorithmic design effort. This leads to the hint: *"It seems that you did not initialize the variables used to hold the minimum and secondMinimum to reasonable values. Think about how the starting values would affect your algorithm for finding the two smallest values. In particular, what would happen if the input values in the sequence were greater, equal, or less than the starting values for your minimum and secondMinimum."*

Updating the two variables in *TwoSmallest* requires algorithmic thinking, and can be a challenge for students new to programming. Many students tend to think about the update process in fragmented, poorly coordinated pieces. To assess if the update of the variables is done correctly, we test input sequences that are permutations of two and three given numbers. If the submitted code passes all the test cases with a valid input of size two but fails the test cases where the third value is less than the minimum value, then it is highly likely that the student is not updating the minimum value correctly. The mapping of the error to "algorithmic thinking" leads us, again, to a hint designed to steer students toward developing this skill: *"It seems that you did not update the variables holding the minimum and/or secondMinimum values correctly. Think carefully about the algorithm that you are developing to update your variables. It may help to think about what would happen if the sequence had the same number appearing multiple times; for example, all possible permutations of 3 numbers with repetition."*

Chapter 5

How can Teaching Benefit from Using CSF^2?

CSF^2 was developed in the context of an introductory CS1 course. However, since comparing faulty and successful executions is extensively used in fault localization, I believe that its design is sufficiently flexible to be extended and adapted to serve the needs of other programming courses that use autograding. It can assist instructors in various tasks:

1. creating assignments that map to the course material to bridge classroom instruction with programming practice;
2. reviewing and improving the course to better fit students' learning needs;
3. designing alternate grading schemes that focus on knowledge components rather than tests results;
4. encouraging informal interactions between students and between students and instructors through discussion of the hints.

In this section, I discuss in more detail each of these uses of the concepts and skills maps.

5.1 Creating Assignments that Map to the Course Material

Traditionally, theory and programming practice are for the most part separated activities in computer science courses. For theory, we have brimming curricula. In comparison, for programming practice, the focus is on the details of coding without proper linkage to concepts and skills. In many cases, the resources for programming practice comprise only tutorials of specific systems, environments, or programming languages students are required to use in the course. The general expectation is that students should be able to take concepts taught in lectures and apply them to programming practice. However, most commonly that process does not happen as expected and the deficiencies developed as a result only amplify over time. Since dropout and failure rates are high, and often students demonstrate fragile learning of basic concepts [89], we need solutions for supporting students' learning. Bridging the gap between concepts and high-level skills to the specific of programming throughout CS courses is one such solution. CSF^2 was designed with that goal in mind. One advantage of using a specific set of concepts and skills which map to an assignment and an error classifier is the ability to query which concepts and skills students are struggling with, similar to the results shown in Table 4.3. With this information, instructors can design interventions targeting specific knowledge deficiencies. These interventions can be used before the administration of the assignment in future course offerings. Some of these proposed interventions may include additional exercises, textbook references, expert examples, or video lessons.

Moreover, the set of concepts and skills and buckets of common errors can aid in the generation of isomorphic assignments while reusing the error classifier and hints. For example, during the Fall 2017 semester, I modified *TwoSmallest* to *TwoLargest*, an assignment that asked students to output the two largest values in a sequence. In this instance, I was able to reuse the test suite, classifier, and hints with only small changes. This can be a starting point for extending CSF^2 to support the evolution of assignments over time (for example, to circumvent cheating) while reusing the classifiers and hints. By repeating the process for all the assignments, teachers can also accumulate knowledge on how to redesign course curriculums to better integrate all the individual activities and material as I describe in the next section.

5.2 Reviewing and Improving Courses

Concerns centered on the quality of education offered by higher education institutions such as universities advocate for the integration of theory and practice [90]. This is a prevalent trend across disciplines. In this section, I describe the effects of applying CSF^2

to two programming assignments in an introductory CS1 course at Rutgers University. The first step in the framework requires compiling the list of concepts and skills for each programming assignment. Compiling these lists was strikingly challenging since I could not find specific resources for them in the course curriculum. Both I and my collaborators were surprised by this realization and the case study generated a wave of changes to the course curriculum and material. For example, together with my collaborators, I developed a core system of learning goals which is now used throughout the course.

Rutgers University supplements instructor's instruction with weekly recitations which are conducted either by upperclassman students or graduate students commonly called teaching assistants (TA). Generally, recitations focus on practice and revisiting course material that was covered during the lecture. However, in large introductory courses with over 1500 students, there could be more than 30 instructors and TAs teaching different sections. Consequently, coordination is a substantial issue. Having a core system of learning goals is the first step towards a solution. In addition, instructors and TAs could use to learn from each other's experiences to improve their teachings. To enhance coordination and shared knowledge about teaching experiences, my collaborators and I developed Dynamic Recitation: A Student-Focused, Goal-Oriented Recitation Management Platform [91]. This platform has been used as an aid to teaching for CS1 at Rutgers University since 2018. The platform was received with enthusiasm by instructors and TAs, however, further research is needed to understand its impact on teaching and learning.

5.3 Designing Alternate Grading Schemes

Instructors can also take advantage of the set of concepts and skills when assigning partial credit to autograded assignments. Traditionally, grading rules for autograders have been very rigid, following the boundaries of unit testing and leading to grades being calculated as the weighted sum of the outcome of the test cases. Relying on test cases alone to grade submissions has resulted in some unexpected behaviors. For example, instructors at Rutgers University reported that students at either end of the scoring spectrum (that is, those who received no credit and those who earned nearly full credit despite still having important misconceptions and mistakes), gave up working on and completing their programming assignments. These observed behaviors may be responsible for the dreaded *"bimodal grade distributions commonly observed in CS courses"* [89]. With my proposed approach, instructors can assign scores based on the importance they allot to each concept or skill. For example, for *PayFriend* I weighed all the tested concepts. Students who understood the assignment but would have, previously, received low grades due to failed test cases were assigned scores more fairly and the scores better reflected their understanding of the problem and its solution. Conversely, I found submissions that failed a few test cases covering core concepts and would have previously received a nearly perfect score. The weighted scoring scheme lowered the scores of these submissions because the test case failures showed important misunderstandings and served as a motivation for students to improve their solutions.

Next, I report on designing and using an alternate grading scheme as part of the case studies. Table 4.2 summarizes the data sets used in my evaluation, with submissions for Spring 2016 collected using Web-CAT and submissions for Spring 2017 and Spring 2018 collected using Autolab. Web-CAT was configured to test the students' code using 10 test cases. Students were allowed to submit each assignment multiple times - five times without penalty, then with a 5 point penalty for each subsequent submission. For each submission, Web-CAT provided the following feedback: a score, whether the submission passed each test t in the test suite, and, in case of failure, a hint associated with the specific test case.[1] The instructor wrote the test cases and the hints at the same time. Scores were calculated as a weighted sum of the passed test cases.

Instructors made a few observations during Spring 2016 which led to changes during Spring 2017 and Spring 2018:

- when given feedback on which test cases passed and failed, students were guessing what were the failed test cases and adjusted their code to correct for those specific test cases rather than think about their overall solution.

[1] Students were not given information on the test cases themselves.

- some students who had gotten a good start but whose code did not compile or whose code failed all the test cases would give up because of a low score.
- some students who had gotten a sufficiently high score would stop working on their code because they were more motivated by getting a "good enough" score than by arriving at a complete solution.

For both semesters when it was used, Autolab was configured to test the students' code using 20 test cases for *PayFriend* and 30 test cases for *TwoSmallest*, generated as described in Section 4.2. Again, the students were allowed to submit each assignment three times without penalty, then with a 5 point penalty for each subsequent submission. During Spring 2017, the only feedback that the students received for their submission was a "progress signal" as described below. During Spring 2018, feedback for each submission included a progress signal and hints generated as described in Chapter 4. In my assessment of the usefulness of the hints in Chapter 7, I focused on comparing the two most similar semesters, Spring 2017 and Spring 2018, although, for completeness, I also include the data and results from Spring 2016.

The above observations lead to the following changes for the Spring 2017 and 2018 semesters:

1. After each submission, students were given a *progress signal* instead of a score. Scores were revealed only after the assignment due date when students were not allowed to submit anymore.
2. Scores were calculated using a scheme in which the weighted sum of the test results was added to a base score (Table 4.1).
3. During Spring 2018, as part of the feedback, students also received hints associated with the error class corresponding to the errors in their submission.

5.3.1 Progress signal

To discourage students from targeting their code to specific test cases, starting with Spring 2017, instructors changed the feedback given to students to a signal indicating overall progress instead of an actual score or information about the number of passed and failed test cases. In this scheme, a submission would be tagged with a red "light" for a score below 20, a yellow "light" for a score between 20 and 60 for *PayFriend* and between 20 and 80 for *TwoSmallest*, and a green "light" for a score above 60 for *PayFriend* and above 80 for *TwoSmallest*. Instructors explained to the students that red meant that a submission was very far from a correct solution, yellow meant that a submission was on the right track but was still giving the wrong answer for many test cases, and green meant that the submission was definitely on the right track, but there was no guarantee of a perfect score or that the submission had passed all the test cases. The latter was used to encourage students to think about comprehensive test plans rather than gaming the system to try to get a perfect score. The final scores were released to the students after the assignment deadline.

5.3.2 Scheme used for calculating scores

For each error class, instructors assigned a base score, shown in Table 4.1. Scores for each submission were calculated by adding the base score associated with the error class to the weighted sum of the passed test cases. The weights used for each passed test case were 1 for *PayFriend* and 1.5 for *TwoSmallest*. For example, if a code submission for *PayFriend* was labeled with COND and it passed 15 test cases, its score became $50 + 15 * 1 = 65$. As shown in Figures 7.2 and 7.3, this grading scheme made the grades approximately follow a normal distribution because it moved the scores on submissions that did not pass any test cases from zero to some partial credit and submissions that were near completion from a near-perfect score to a score that was less than 85 for each assignment. This scheme increases the scores for early but significant efforts to encourage students to keep trying and it also increases the value of "solving the few remaining bugs" to encourage students who are doing well to keep trying.

Lastly, regardless of the type of feedback provided, students may still individually or collectively resort to gaming the system to get full points [82]. Limiting the submission count without penalty to three makes it nearly impossible for one student to game the system since it considerably limits the number of data points coming from the interaction with the autograder. Instructors using CSF^2 classifiers to automatically provide feedback have observed that students often discuss the hints on the online course forums. Further research is needed to understand how the autogenerated feedback affects students and their learning.

Chapter 6

How Accurate is CSF^2 in Classifying Errors?

In this section, I detail the process used to assess the accuracy of the error classifiers developed by applying CSF^2 to two programming assignments and outline findings. These classifiers were produced from submissions collected during Spring 2016 and run on submissions from Spring 2017. After classifying Spring 2017 submissions, I asked three undergraduate students who had previously taken the Introduction to Computer Science course to carefully review the submissions and the errors produced by the classifiers and assess the accuracy of the classifications and the potential efficacy of the corresponding hints. At the time this paper was written, these students were enrolled in the computer science program or had recently graduated with a computer science degree. I believe that having done the assignments themselves while taking the course gave these students a good perspective in the evaluation of the automatic error classification.

6.1 Human Evaluation Approach

Human evaluation approaches can be subject to biases (e.g., knowing categories ahead of time can lead to conformity bias). I took some measures to reduce or minimize such biases. In particular, I asked the evaluators to assess the code and write an appropriate hint before deciding if the "autogenerated" hint was suitable. The hints they wrote were very similar to the autogenerated hints. For example, when a student got one of the formulas wrong, one of the evaluators' hints was "check your math." Hints were not given when the evaluators took the class, and so they were not biased that way. Finally, the evaluation presented here only provides evidence for the accuracy of the error classifiers. Ultimately, the impact of the error classification and the corresponding hints on the students, reported in Chapter 7, is the most important.

To evaluate the classifiers, each erroneous submission and its corresponding error class and hint was labeled either *Correct*, *Partially Correct*, or *Incorrect*. The *Correct* label meant that the automatic diagnosis and corresponding hint fully captured the logical errors in the submission, and so would potentially provide useful guidance to the student. Note that I say "potentially" since the labeling was done by people other than the owners of the submissions. The *Partially Correct* label meant that the diagnosis and hint only partially captured the errors in the submission. *Incorrect* meant that the errors were misdiagnosed and so the hint was misleading or would not have made sense. Each submission was inspected and evaluated by at least two people. The results were analyzed by me to resolve conflicts and to ensure consistency between evaluations. I computed the inter-rater reliability score by assigning 1 to all the instances where the reviewers agreed and 0 otherwise, summed for all the submissions and divided by the total number of submissions. The inter-rater reliability score we obtained was 93%.

6.2 Findings

Table 4.3 summarizes the results of the manual evaluation of the accuracy of the error classification. In particular, it shows that the classification was correct for 91.5% of the code submissions for *PayFriend* and 87.5% of the code submissions for *TwoSmallest*. Moreover, the manual inspection of the submissions and their classification for both assignments revealed errors that would have been difficult to detect with the gray box testing I used. I call it "gray box testing" because I had limited knowledge about each student's code at the time when the test cases were designed. Furthermore, the test cases were designed for a full path coverage of the reference solution as described in Section 4.2, which may or may not be close to the student solutions.

To demonstrate the inherent limitations of gray box testing for *PayFriend*, consider the student code in Figure 6.1. This code passes only the test cases for inputs smaller than 100. The classifier labels it as FORM (see Table 4.1), because it is using incorrect calculations inside three out of the four intervals. The student seems to have a poor

```
public class PayFriend {
    public static void main(String [] args) {
        double payment=IO.readDouble();
        double fee=0;

        if (payment>15000) {
            fee+=(10000*0.01)+(5000*0.02)+
                    ((payment-15000)*0.03)+5;
        }
        if (payment>10000) {
            fee+=(10000*0.01)+((payment-10000)*0.02);
        }
        if (payment>1000) {
            if ((payment*0.01)>15) {
                fee+=payment*0.01;
            } else {
                fee+=15;
            }
        }
        if (payment>100) {
            if ((payment*0.03)>6) {
                fee+=payment*0.03;
            } else {
                fee+=6;
            }
        }
        if (payment<100){
            fee+=5;
        }

        IO.outputDoubleAnswer(fee);
    }
}
```

Figure 6.1: Example of student code that was improperly labeled by my classifier.

understanding of control flow and of the difference between consecutive *if*s and *if-else*. An accurate hint would tell the student that, for payments greater than 100, the code may change the fee multiple times which would be considered incorrect behavior. The simplest fix is to add an *else* after each *if* and to put the following conditional expressions inside that *else*. The code example in Figure 6.1 shows that because of the semantic and functional brittleness of code [65], slight deviations in the code's block structure can result in an error signature that the classifier is not able to properly label. Using observations of this sort I developed an extension to CSF^2 described in Chapter 8.

Despite the above limitations, the high accuracy of the error classification and corresponding hints provides strong evidence that they are appropriate for the vast majority of erroneous submissions. Thus, the classifiers and corresponding hints were deployed in the introductory CS1 course starting with Spring 2018 semester. During Spring 2018 semester, student submissions were again collected and students were also asked to complete a survey. These submissions and the survey were used in a subsequent study which I relay in the next chapter.

Chapter 7

How does the Automated Feedback Impact Students?

In this section, I present findings based on comparing student submissions from one semester without hints (Spring 2017) to one with hints (Spring 2018) and a survey conducted in the semester with hints. Recall that the classifiers were developed using the submissions collected during Spring 2016 as described in Chapter 4.

7.1 Datasets and Significance Tests

As mentioned before, Table 4.2 summarizes the data sets used throughout the case studies, with submissions for Spring 2016 collected using Web-CAT and submissions for Spring 2017 and Spring 2018 collected using Autolab.

I compare the data from Spring 2017 and Spring 2018 because these two semesters are the most similar – they used the same autograder, the same progress signal ("light" colors instead of scores), the same grading scheme, and the same number of submissions allowed without penalty. The difference between the two semesters was in whether students were given hints or not. I added data from Spring 2016 for interest, but do not include it in the analysis of the usefulness of the hints because the two semesters when students received hints (Spring 2016 and Spring 2018) were very different. As explained above, during Spring 2016, a different autograding system was used that associated a hint to each test case, and a different formula was used to compute the score for each submission. Note that the scores shown for Spring 2016 in this section are not the actual scores that the students received, but rather the scores that students would have received with the grading scheme from Spring 2017 and Spring 2018 (Section 5.3). *Overall, the data from Spring 2016 is consistent with observations from the comparison between Spring 2017 and Spring 2018, that is, the performance of students* *on programming assignments improves when they receive hints.*

In my analysis, I use Pearson's χ^2 test to determine the significance of the difference between two proportions and the Kolmogorov-Smirnov test to determine the significance of the difference between two cumulative distributions (e.g., comparing numbers from Spring 2017 and 2018). I use a significance level of 0.05.

7.2 Usefulness of the hints

I use two measures to evaluate the usefulness of the hints generated using my framework: *empirical usefulness* and *perceived usefulness.* I say that hints are *empirically useful* if there are statistically significant differences between the semester when students did not receive hints (Spring 2017) and the semester when students received hints (Spring 2018) in terms of resubmission rate, final score, score difference between the first and last submission, and score difference between consecutive submissions. I determine whether hints are *perceived* as *useful* from the students' responses to a survey completed at the end of the Spring 2018 semester. I examine the distribution of student responses to specific questions, their comments in response to the hints, and anecdotal information from instructors.

7.2.1 Empirical usefulness of the hints

To assess the empirical usefulness of the hints, I analyzed the differences between Spring 2017 and Spring 2018 in terms of the cumulative percentage of students who resubmitted their assignment, the final scores, the score differences between the students' first and final submission, and the score difference between consecutive submissions.

First, I observe an increase between Spring 2017 and Spring 2018 in the number of assignment resubmissions, both in terms of the percentage of students who resubmitted (shown in the last row of Table 4.2) and the cumulative distribution of the percentage of submissions that are the n^{th} submission from a student out of all submissions (shown in Figure 7.1). The last row in Table 4.2 shows that the percentage of students who resubmitted increased between Spring 2017 and Spring 2018, but the difference is

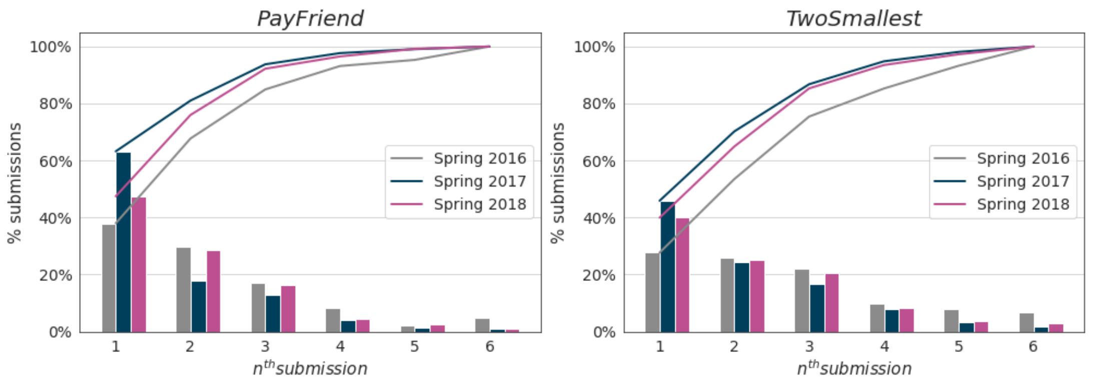


Figure 7.1: Cumulative distributions and histograms showing the percentage of submissions that are the n^{th} submission from a student out of all submissions. A "lower" CDF indicates higher percentages of later submissions, corresponding to students submitting *more* times. For Spring 2016, students were allowed to submit up to 5 times without penalty, compared to 3 times for 2017 and 2018, which is likely the reason for the observed highest rates of re-submissions in 2016.

statistically significant only for *PayFriend*. This increase is represented in a rightward translation of the line, up to three submissions. Then, the lines converge because of points being deducted after more than 3 submissions, that is *5 points* of the student's grade were subtracted from the score for every submission after the third one. I believe that the higher resubmission rate for Spring 2016 can be explained by the fact that students were able to submit their code 5 times without a penalty instead of 3 times during the other semesters.

The rate of resubmissions is important but it gives only one piece of evidence for the usefulness of the hints. I further look at additional usefulness indicators.

I define two other measures which I believe point to the empirical usefulness of the hints: the *efficacy of the hints* as the overall progress students made between their first and last submission measured as a difference in score and the *efficiency of the hints* as the progress with each resubmission measured, again, as a difference in score. I hypothesize that the increase in overall score and the increase in score with each resubmission are evidence of the usefulness of the hints.

Table 7.1: Percentages of students who completed (that is, received a prefect score for) each assignment and the number or submissions it took (one or multiple). † marks a statistically significant change between Spring 2017 and Spring 2018.

	PayFriend			***TwoSmallest***		
	Spring 2016	**Spring 2017**	**Spring 2018**	**Spring 2016**	**Spring 2017**	**Spring 2018**
Students who successfully completed the assignment in one submission	26.8%	30.4%	22.5%	15.6%	14.4%	22.5%
Students who successfully completed the assignment after resubmitting	34.4%	11.6%	31.8%†	30.9%	9.2%	28.8%†
Total students who successfully completed the assignment	61.2%	42.0%	54.3%	46.5%	23.6%	51.3%†
Total students who did not complete the assignment	38.8%	58.0%	45.7%	53.5%	76.4%	48.7%†

Efficacy of the hints

I measure the efficacy of the hints by the progress students made toward completing their assignments when they were given hints after each submission. I evaluated progress by the percentage of students who completed the assignment and by the increase in score between the initial and final submissions.

I notice a substantial difference between the percentages of students who completed the assignment in one submission, the percentages of students who completed the assignment using multiple submissions, and the percentages of students who did not complete the assignment (Table 7.1). Although I see some differences between the percentages of students who completed the assignments in one submission, analyzing them goes beyond the scope of this study given that the two populations of students come from two different semesters. These differences are also not statistically significant. When hints were provided, the percentage of students who completed their assignment after resubmitting almost tripled for both assignments compared to the semester during which no hints were provided (these numbers are shown in the second row of Table 7.1), and the difference is statistically significant.

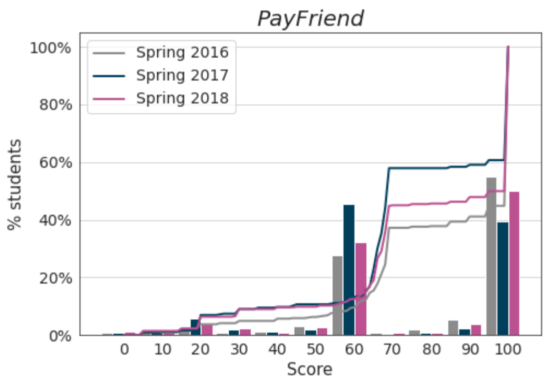


Figure 7.2: The cumulative distribution of students' final scores for each assignment along with the histogram showing the distribution of grades in buckets of 10 points. For example, at 60, the bars show the percentages of scores between 60 and 69. At 100, the bars indicate the percentages of scores equal to 100.

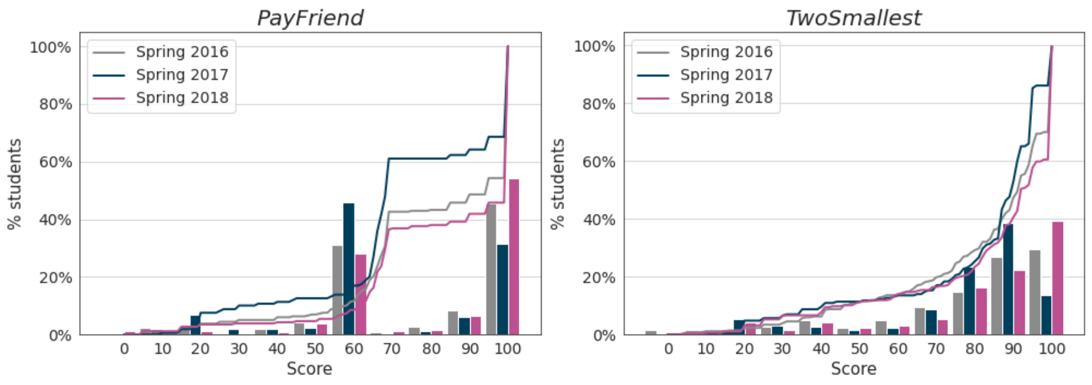


Figure 7.3: The cumulative distribution of final scores for students who submitted multiple times along with the histogram showing the distribution of grades in buckets of 10 points. For example, at 60, the bars show the percentages of scores between 60 and 69. At 100, the bars indicate the percentages of scores equal to 100.

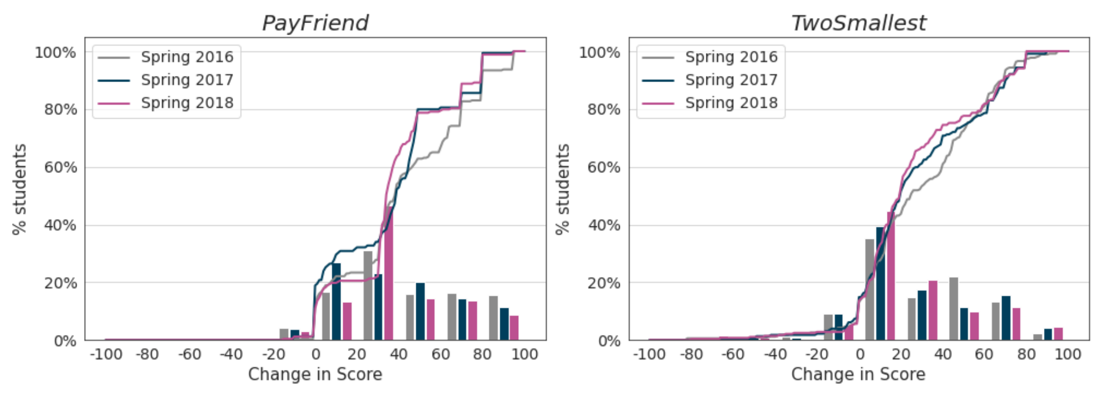


Figure 7.4: Cumulative distributions and histograms of differences in score between the final and first submissions for all students who submitted multiple times. Numbers are shown for 20 percentage point increments (for example, at 30, the bars show the percentages of students with a difference in score between 20 and 40. The *x-axis* indicates the score difference and the *y-axis* indicates the percentage of students whose score difference matches x.

Figures 7.2 and 7.3 show the distributions of the students' final scores. In particular, for students who submitted multiple times (Figure 7.3), for *PayFriend*, about 5% more students received a final score between 20 and 55, and about 20% more students received scores above 70 during Spring 2018 than during Spring 2017. This difference is statistically significant. Interestingly, the CDFs for *PayFriend* show sharp rises around 20, 60 and 100. I believe that these sharp rises (and the relative flatness in between) can be attributed to the fact that: (1) the assignment is simple, and so students tend to earn points in large chunks, and (2) 20 and 60 are boundary scores when the progress signal changes from red to yellow and from yellow to green for Spring 2017 and Spring 2018. Many of the students stopped working on their code when they received the green light signal, even though the instructors explained that getting a green light did not guarantee a perfect score. Students understood this guideline much better for TwoSmallest, which explains why more students received higher scores in the "green light" range for the second assignment. For *TwoSmallest*, I see a similar trend with about 10-20% more students receiving final scores above 85 during Spring 2018 than during Spring 2017. These differences in the cumulative distribution of the students' final scores suggest that the hints were useful.

Figure 7.4 shows, for each assignment, three cumulative distributions, one for each

Table 7.2: Percentages of students who were able to fix all their errors after one resubmission (out of all the students who did not achieve a perfect score on their first submission). † marks a statistically significant difference between Spring 2017 and Spring 2018.

	PayFriend †	***TwoSmallest***
Spring 2016	27.34%	19.86%
Spring 2017	22.83%	21.00%
Spring 2018	37.62%	23.90%

semester of interest, of the differences in score between the final and first submissions. During the semester when hints were given, a higher percentage of students made moderate progress as reflected in an increase in their score between 30 and 45 for *PayFriend* and between 20 and 40 for *TwoSmallest*.

Finally, I note that it appears that hints are most useful for students who are close to getting a perfect score. In addition, for *PayFriend*, the hints seem to also help students with a score between 20 and 30. These are students who do not use the IO module properly; the IO module is an interface for receiving input and supplying output.

Efficiency of the hints

Intuitively, the *efficiency of the hints* has to do with how much quicker students make progress toward completing the assignment when they receive hints compared to when they do not receive hints. More specifically, for all the students who submitted multiple times, I used their submission history to create pairs *(i,i+1)* of consecutive submissions. Next, I counted all the pairs for which the first submission *i* and the following submission *i+1* were labeled with the same error class. I show the results in Table 7.2 and Figure 7.5.

Firstly, I observe an increase in the probability that a student will complete their assignment (that is, their submitted code will pass all the test cases) with every resubmission as reflected by an increase in the percentage of students who fixed all their errors in one resubmission for the semester when hints were provided, which is statistically significant for *PayFriend*. In Table 7.2, for each assignment, we show the percentages of students who fixed all their errors in one resubmission, by assignment. To calculate

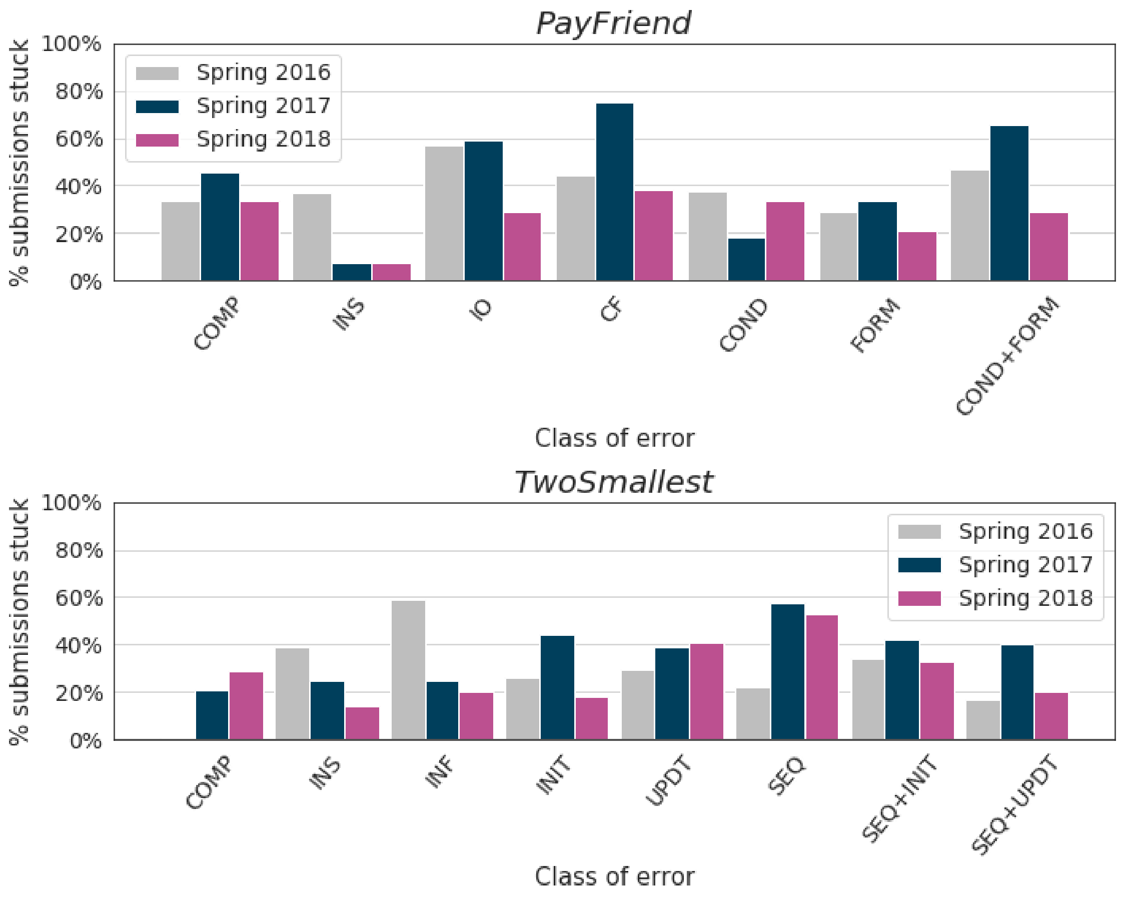


Figure 7.5: Histograms showing the distribution of submissions stuck in an error class from one submission to the next. The *x-axis* indicates the error class and the *y-axis* indicates the percentage of students who were not able to make progress from the previous submission. For Spring 2016, COMP column looks like it's missing because of the graph scale and the fact that it is only 1%.

these percentages, I counted all the pairs of consecutive submissions for which the first submission *i* was labeled with an error class and the following submission *i+1* had no error. During the semester with hints, more students were able to complete their assignment after one round of hints, but only for *PayFriend* was the difference between semesters statistically significant.

Secondly, we see a decrease between Spring 2017 and Spring 2018 in the percentages of submissions that were stuck in the same error class from one submission to the next except for INS and COND for *PayFriend* and COMP and UPDT for *TwoSmallest* as shown in Figure 7.5. I calculated the percentage of submissions stuck in each error class by using the number of pairs of submissions that started and ended in that error class and divided it by the number of all the pairs of submissions for which the first submission was in that error class. For *PayFriend*, we see progress for all error classes, except for INS, which is the error class in which the student's code was saved under the wrong filename, the class or method names were wrong, and COND, which is the error class in which the student's code failed one or multiple test cases for boundary values. It is difficult to point out these errors without giving away the answer or revealing the test cases, which I am trying to avoid. For *TwoSmallest*, we see smaller progress for the majority of the error classes, with a slight increase for UPDT and a bigger increase for COMP. I think the smaller progress is linked to the greater complexity of the assignment. For *TwoSmallest*, the solution strategy comprises three main tasks: differentiate valid entries from terminating values (SEQ), initialize the variables to store the minimum values (INIT), and update these variables (UPDT). INIT and UPDT point to parts of the strategy that are separate, that is INIT ends when the first two valid values are read, and UPDT starts with the third valid value. SEQ points to the part of the strategy that is interleaved with INIT and UPDT, but the behavior of SEQ changes from INIT to UPDT as follows: during INIT, the terminating values are discarded and new values are being read, whereas during UPDT, the read stops when the terminating value is read. To test for errors in the class INIT, students need to test for all the possible combinations of two numbers with repetition, that is 3 or 4 test cases; the two numbers represent the first two valid entries in the sequence. For the error class UPDT, in the least, students need to test for all the possible combinations of three numbers with repetition (three or more, anything more than two), that is 12 or more different test cases. Finally, an additional four or more test cases are required to test the behavior of UPDT (incorrect update of min values, Table 4.1), using number sequences with additional terminating values interleaved between the first input (the terminating value), and the first valid input (different from the terminating value), and number sequences with additional terminating values between the first two valid entries. This analysis shows that *TwoSmallest* is far more complex than *PayFriend*. However, for both assignments, we see that fewer submissions were stuck in the same error classes

Table 7.3: Assignment completion rates for all students and for students who completed the TAM survey. "One submission" encompasses students who have submitted each assignment only once; "multiple submissions" means that students submitted at least one of the assignments multiple times.

	Assignment completion		***All Students***	***Survey Respondents***
	PayFriend	***TwoSmallest***		
One Submission	yes	yes	22 (4.8%)	15 (5.2%)
	yes	no	16 (3.5%)	9 (3.2%)
	no	yes	23 (5.0%)	13 (4.5%)
	no	no	34 (7.4%)	14 (4.9%)
Multiple Submissions	yes	yes	135 (29.4%)	99 (34.5%)
	yes	no	77 (16.8%)	43 (15.0%)
	no	yes	59 (12.9%)	41 (14.3%)
	no	no	93 (20.3%)	53 (18.3%)

during Spring 2018 than during Spring 2017.

Finally, I note that hints appear to be helping at least some of the students make progress in addressing errors in their code. As shown above, for the semester with hints, there was a decrease in the percentage of students who were stuck in the same error class, and more students were able to complete their assignment after resubmitting just once, compared to the semester without hints.

In conclusion, the hint system I implemented in Autolab as proof of concept for my proposed framework appears to help students make progress from one submission to the next, as well as improve their final grades. However, they do not seem to help students equally: it appears to provide more help to students whose code is in certain error classes and those who are near the completion of their assignment.

7.2.2 Perceived usefulness of the hints

To assess the perceived usefulness of the hints, I look at the distribution of the students' responses to a survey completed at the end of the Spring 2018 semester, their comments on the hints in Autolab, and the anecdotal information provided by the instructors of the course.

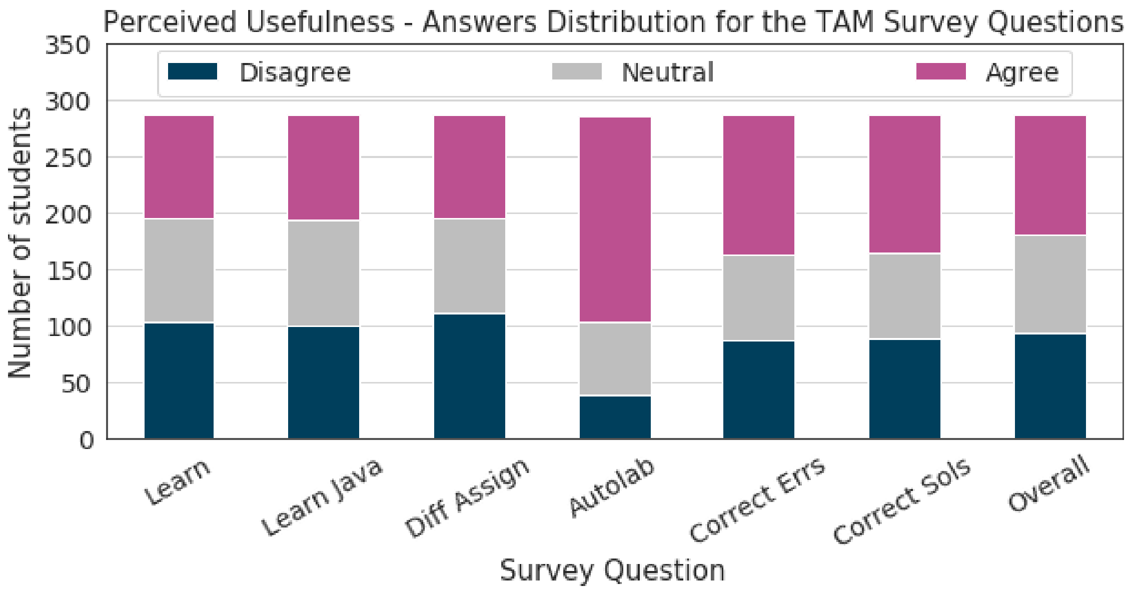


Figure 7.6: The distribution of student answers to questions in the survey: Learn - The hints provided by Autolab helped me learn, Learn Java - The hints provided by Autolab were useful for learning to program in Java, Diff Assign - The hints provided by Autolab were helpful when I was working on a difficult assignment, Autolab - Autolab improved my abilities as a Java programmer, Correct Errs - The hints provided by Autolab helped me correct my errors, Correct Sols - The hints provided by Autolab helped me write correct assignment solutions, Overall - Overall, I found the hints provided by Autolab very useful.

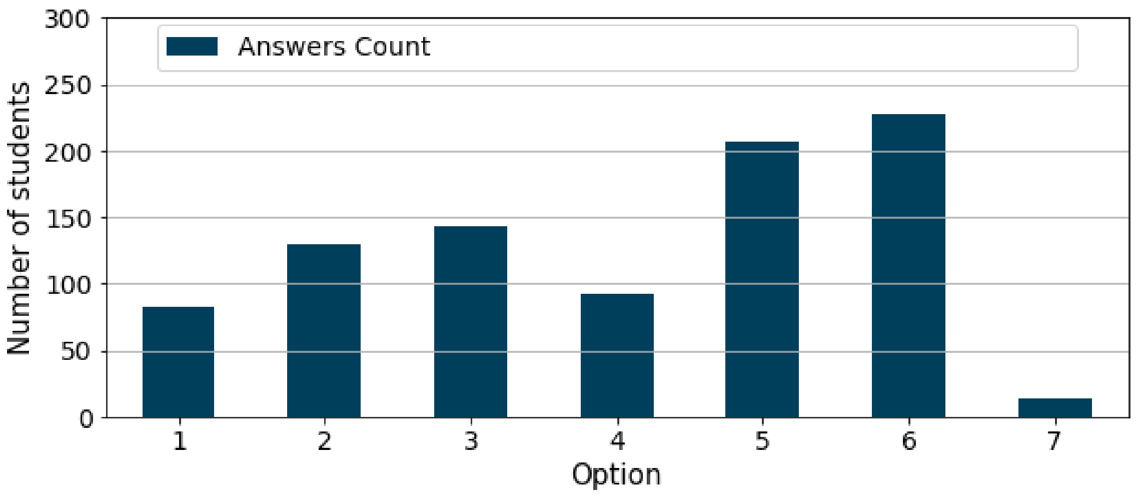


Figure 7.7: Perceived usefulness of Autolab and the hints, as reflected by the responses to the TAM survey; The plot shows what Autolab features students thought could be improved: 1 - Having more help when learning how to use Autolab, 2 - Having more assistance when having difficulties with Autolab, 3 - Having more help when something goes wrong in Autolab, 4 - Not having to use the IO module, 5 - The wording of the hints, 6 - The information provided by the hints, 7 - other, typed by the student.

Table 7.4: Percentage of students who agreed with survey statements. "One Submission" means that students submitted each assignment once and "Multiple Assignments" means that students submitted at least one assignment multiple times. Note that students submitting just once would still have gotten hints unless they received a perfect score.

	One Submission	**Multiple Submissions**	
		Neither Completed	**Completed One or Both**
Having more help when learning how to use Autolab	35.3%	**35.8%**	21.3%
Having more assistance when having difficulties with Autolab	45.1%	**50.9%**	39.3%
Having more help when something goes wrong in Autolab	47.1%	**56.6%**	44.8%
Not having to use the IO module	29.4%	28.3%	30.1%
The wording of the hints	**60.8%**	**67.9%**	**71.6%**
The information provided by the hints	**60.8%**	**69.8%**	**82.0%**
Other	5.9%	**9.4%**	3.3%
No Answer	0.0%	0.0%	1.1%

Survey Results

I built a survey using the Technology Assessment Model (TAM) [92] to assess the perceived usefulness and ease of use of Autolab, with a focus on the hints. A copy of the survey can be found in Appendix C. At the end of the Spring 2018 semester, I asked the students to complete this survey for a small credit toward their final grade. Out of the 459 students who submitted at least one of the assignments in Autolab, 287 students completed the survey, a 62.5% response rate. Table 7.3 shows assignment completion rates for survey respondents and for all students. An assignment is considered complete when it passes all of the test cases and receives a score of 100. Given that the two distributions ("All Students" and "Survey Respondents") are similar, I conclude that the sample of survey respondents is representative of the student population who submitted their assignments to Autolab during Spring 2018. Next, I present results from the analysis of the students' responses.

I start by using 7 of the survey questions, 6 of them asking about the hints and one of them asking how much the students agreed that Autolab improved their abilities as Java programmers (Figure 7.6). My first observation is that nearly two-thirds of the students agreed that Autolab improved their abilities as Java programmers, whereas only between about one-third and about one-half of the students agreed that the hints were useful. Among questions about the usefulness of hints, a slightly higher number of students (about *10%* more) agreed that the hints were useful for correcting errors and for writing correct assignment solutions. It is unclear whether students who did not find the hints useful had that perception because they were looking for more detailed and explicit feedback. Such feedback would be against the spirit of my approach: I want to give hints that enable students to *think* about the problem and then make progress, rather than follow detailed instructions on how to correct their code. It would be interesting to obtain more information about the students' thoughts on this matter in the future.

Secondly, I observe that about two-thirds of the students thought that the wording of the hints and the information provided by them needed improvement (Figure 7.7).

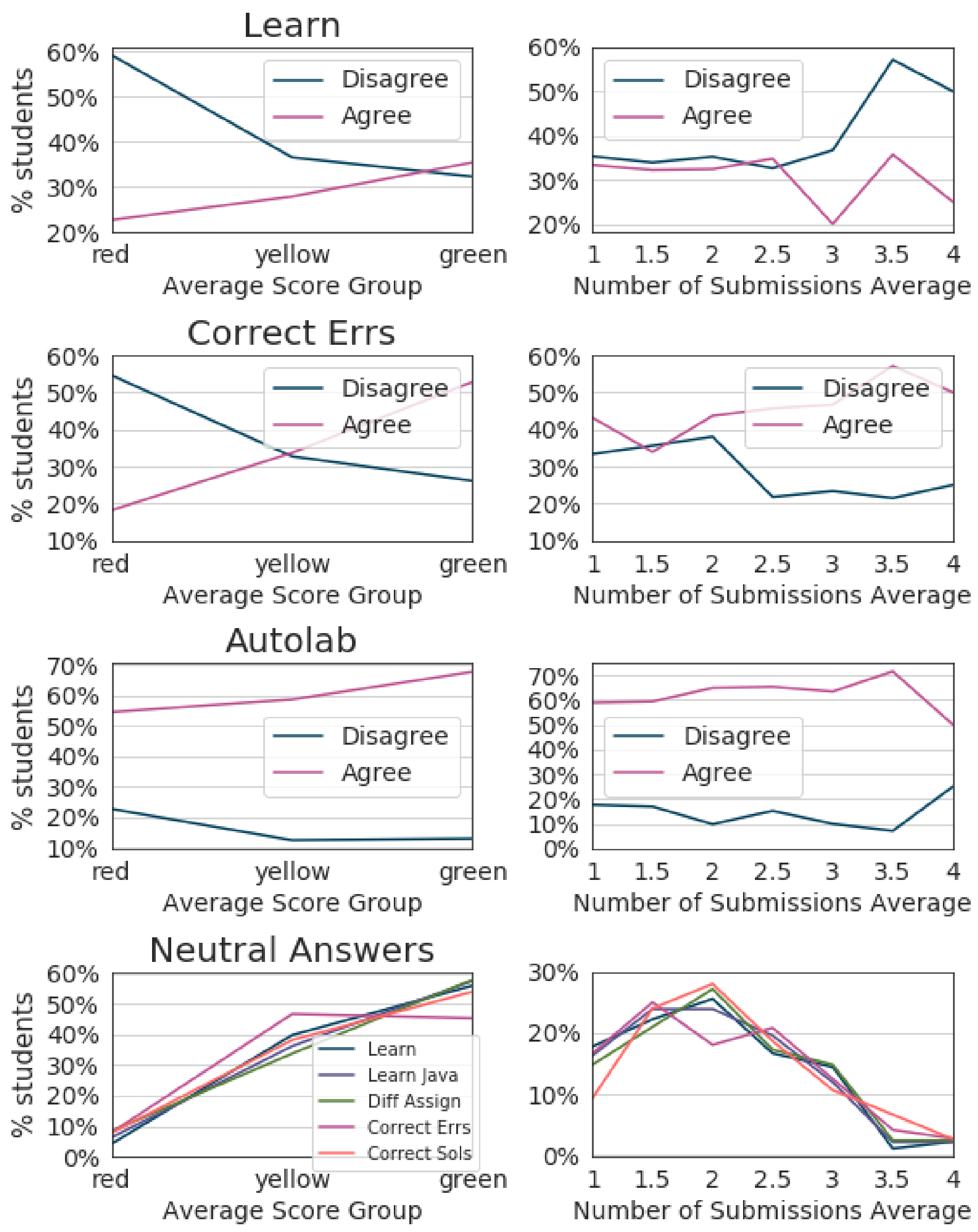


Figure 7.8: Distribution of survey responses by average grades and by average number of submissions. The first column shows the distributions of the average score between the final submissions for PayFriend and TwoSmallest from lowest (red) to highest (green) by survey responses. The second column shows the distribution of the average number of submissions by survey responses.

When combining the percentage of students who agreed with the survey statements broken down by the number of submissions (one vs multiple) and the assignment completion rates shown in Table 7.4, we see that the most selected options for improvement could be categorized as those regarding using the autograding system and those regarding the hints. Students who submitted multiple times but did not complete any of the assignments expressed more often that the autograding system needed improvement, whereas students who completed at least one of the assignments more often expressed that the language of the hints and the information provided by the hints needed improvement. Finally, fourteen students selected *Other* to answer the question of what needed improvement and typed their suggestions: five students mentioned the way files were uploaded in Autolab, five students mentioned the color "lights" shown instead of scores, three students wrote that test cases would have been more useful feedback than the color "lights", and finally two students mentioned that Autolab was slow, especially when approaching the assignment submission deadlines.

Thirdly, in Figure 7.8, we see that the likelihood that a student perceived the hints as helpful increased with their score as shown in the graphs titled *Learn* and *Correct Errs*, and that more students who obtained higher scores had neutral answers to the survey questions as shown in the graph titled *Neutral Answers*. The score is shown as the light color corresponding to the average between the student's score on the final submission for *PayFriend* and the score on the final submission for *TwoSmallest*. Moreover, we see that, as the average number of submissions increased, students were more likely to agree that the hints were useful in correcting their errors (second row, right side graph in Figure 7.8), and that fewer of them had neutral answers (fourth row, right side graph).

Students' comments on hints

In Autolab, students had the option to write a comment or feedback every time they received a hint. Students wrote 25 comments for *PayFriend* and 19 comments for *TwoSmallest*. Most commonly, the students' comments expressed disagreement with or confusion about the hints. One student's comment expressed that the error in the code had been correctly identified by the hint. Most often, the students said that they disagreed with the hints they received because their code had passed the test cases and they often asked for the test cases that their code had failed. They also asked for more details about their errors, such as what was the line number in their code containing the error. Finally, many students thought that the assignment descriptions were not clear.

Anecdotal information from instructors

Overall, the instructors communicated that they found the hints to be useful, but that they were not enough to correct the students' thinking. They offered suggestions for improving the hints such as: changing the wording of the hints, improving the accuracy of the hints, and improving the content of the hints. For example, they thought that providing failed test cases along with the wording of the hint would be very useful for the students. However, figuring out what is the best information and the best way to present it to students is an open question beyond the scope of this research. I aimed to write hints that gave students enough information to correct the logical errors in their code without giving away the answers. Moreover, the hints were provided only for a subset of the assignments and instructors reported that students were asking for hints for assignments that did not provide hints, suggesting that students at least perceived the hints to be useful (whether they were or not).

In conclusion, both students and instructors agreed that the hints were useful, but that they could be improved, in particular the information provided in them as well as their wording. I plan to explore these directions in my future work.

Chapter 8

PR-CSF^2: An Extension to CSF^2 Using Program Repair

In this chapter, I present Progam Repair - CSF^2 which is an extension to the initial framework [93] designed to handle some of the limitations of CSF^2. These limitations are:

1. **The granularity of the feedback generated using CSF^2 is too coarse for some cases.** For example, I observed errors in student programs involving variable types, integer, and double division, control flow (including dead code) that can not be caught using the testcase signatures alone.

2. **The feedback generated using CSF^2 is inaccurate at times.** This issue is related to the previous one and it happens because for some testcase signatures there could be many reasons why the program is failing in that way. Having a way to test different hypotheses about what may be wrong in students' code is a way to build confidence in the feedback and possibly to provide more information to students.

3. **Many students are unable to fully complete their programming assignments by the time they are due.** Often their programs need only minor changes. However, because they do not understand what is wrong with their program or think their program is correct, they become frustrated with not receiving a perfect score. Other times student programs have issues with basic concepts such as the difference between an integer and a double variable and how division works when variables are of one type or the other. Being able to provide program repair information along with a textual explanation of the error can alleviate students' frustration and teach them about concepts and skills.

4. **The initial approach to CSF^2 is data-driven and data-driven techniques are known to suffer from the cold-start problem.** That means assignments need to be offered during multiple semesters which may promote cheating. In addition, no feedback can be provided for new assignments or the first round of student submissions. Having an approach that can provide some feedback for new assignments complements the initial approach used by CSF^2.

5. **Generating feedback using CSF^2 is labor-intensive because it is specific for a class of assignments.** Having an assignment-independent model for providing some feedback to new assignments offers a less labor-intensive alternative to the primary approach.

Using information from the case studies conducted with CSF^2 I report on in previous chapters, I designed a student program repair model which focuses on addressing misconceptions that students might have regarding their corresponding concepts and skills along with the errors in their programs. Next, I discuss the reasons behind the design choices I have made in developing PR-CSF^2 in Section 8.1, provide working examples to substantiate my reasoning in Section 8.2 and describe how PR-CSF^2 approach works in Section 8.3.

8.1 Overview of PR-CSF^2's Design

The goal of automatic program repair is to identify a set of changes that can turn a program that is incorrect concerning a given specification into a correct one. The problem of finding a fix to a faulty program is commonly treated as a purely computational problem, even in the context of automated feedback generation for student programs [18]. Fixing student programs is generally considered much easier than fixing complex programs written by experts to solve open-ended tasks primarily because the assignment description or the dataset of prior student programs can be used to guide the process of finding a repair. Additionally, the complexity of the analysis required to find the fix commonly grows with the size and complexity of the code. These observations impel us to think that the problem of repairing student programs is an easier version of the harder problem of repairing expert programs and, by extension, that the type of repairs required by student programs are a subset of the larger set of program repairs.

The types of errors students make in their programs are not necessarily a subset of the errors experts make in their programs. In Section 8.2, I give examples of student erroneous code to substantiate this claim. Due to their fragile knowledge about the programming structures they just learned before the programming assignments, students do not know how to properly use them yet. Program repair approaches for automated feedback generation commonly leverage changes to the syntax of the program [38, 21]. However, in many cases, the appropriate fix may involve changes to the program structure.

The types of repairs required by student programs are not necessarily a subset of the larger set of program repairs observed in expert programs. Student programs commonly have structural issues which may or may not negatively impact the behavior of their programs during testing. In the cases in which they impact the behavior of the program, there may be alternative syntactical changes to the program that can make up for it. However, the resulting program repair may be undesirable and may confuse students even more. Moreover, even in the cases in which they do not impact the behavior of the program, providing feedback on the structuring of their code to students is valuable and can be used to correct their misconceptions.

Finally, to be effective in advancing students' learning, feedback received by students needs to contain at least textual explanations of the reason for the repair [79] and preferably other conceptual hints that connect it to the material taught in the lecture. Consequently, the specific question I try to answer throughout my research with PR-CSF^2 are:

1. Can I automatically identify program structure repairs to student programs in addition to statement-level repairs?

2. Can I use the repair signatures to provide additional automated feedback to students, such as feedback on the structuring of their programs?

3. How does the quality of the repairs compare to manual repairs by experts?

I propose a methodology for providing automated feedback to programming assignments that focuses on linking program repairs to the concepts and skills required for solving a programming assignment. This approach leverages the fact that student programs repairs can be broken up into smaller program repairs (which I refer to as *program repair units*) which are common among student programs and across assignments. A key benefit to this approach is that these *program repair units* can be mapped to specific misconceptions for a particular assignment. For example, casting an integer to a double before the division is needed in java to ensure that floating-point division will be computed and indicates that the student understands the difference between integer division and floating-point division. Thus, the *program repair unit* that casts an integer variable to a double variable before a division takes place signals that the student may have misconceptions about variables, variable types, and how division works based on the type of the variables. My proposed method first identifies the set of *program repair units* that are needed to correct the student program, and then provides correction feedback for each of the *program repair units*, along with textual information based on their associated misconceptions.

By analyzing student programs, I identified a set of *program repair units* that are common across student programs and assignments. These *program repair units* can be simple, such as modifying a $<$ operand to a $<=$ operand, or more complex, such as modifying the program structure. I then implemented a search algorithm for identifying which set of *program repair units* is needed and where in the program they are needed. The algorithm iteratively applies repairs to different parts of the program until it finds the correct program. This kind of evolutionary algorithm is known to work well in complex search spaces and with possibly many local optima [94] which apply to the program repair problem. However, the efficacy of the algorithm depends on the search procedure as I further discuss in Section 8.3.1.

Program Structure Edits. Student programs have program structure issues in addition to other issues that can be fixed by modifying specific statements. A key benefit to my approach over previous approaches is that it allows for a set of program structure edits to be performed in addition to statement-level edits. The search algorithm interleaves applying statement-level repairs and program structure repairs to parts of the program that are presumed to contain the error. Based on the execution trace information of passed and failed testcases, each statement receives a progress score similarly to previous fault localization techniques. Statements are then ranked based on the progress score, and the search algorithm applies repairs to parts of the code in order of their rank. I apply this approach to multiple assignments from the Introduction to Computer Science course and report a higher success rate than with previous approaches, an increase of 20% of the attempted student programs for specific assignments. Manual analysis of the repaired programs reveals that the repairs do not introduce stylistic issues such as dead code.

Repair Coverage. Similar to prior approaches, PR-CSF^2 does not attempt to repair deep semantic issues in student programs. Student programs have syntactic and semantic issues that make them behave differently than expected. My approach attempts to repair those issues due to misuse of programming elements and structures. It only performs edits within the same domain or by using elements within the program. PR-CSF^2 does not add external code structures. For that reason, if the student program is missing part of the solution or has deep design flaws, PR-CSF^2 fails to find a repair. As assignments become more and more complex, student programs are more likely to have these issues, and thus the success rate of PR-CSF^2 drops for these assignments as further shown in Section 9.3.

Next, I present examples of erroneous code from students which illustrate my observations and have motivated my research for PR-CSF^2 followed by the description of the program repair approach in Section 8.3, a summary of the implementation in Section 8.4, and a discussion of possible modifications to PR-CSF^2 in Section 8.5.

8.2 Motivating Examples

For some of their programming assignments, students must write code that executes multiple tasks one after another or in parallel. One example is reading a sequence of numbers inside of a loop while processing each number for a purpose; for instance, finding the minimum value among the numbers. Soloway and Spohrer [95] report that students have a hard time grasping loops and data reading separately and, in particular, when the assignment requires them to perform data reading inside a loop, which is the case with many assignments. If on top of that combination different rules for processing the data read are used, then the challenge of the assignment becomes so great that students produce a plethora of incomplete solutions.

In this section, I present examples of student erroneous code to substantiate my claims behind the design choices for PR-CSF^2. They are simplified code excerpts from larger student incorrect programs. The task each student is trying to code is to read a terminating value, and then read values until the terminating value is read again. At the same time, the code needs to compare the values read and keep track of the minimum value among the values read in between the terminating values. Figure 8.1 shows a reference solution for this task and Figures 8.2, 8.3, and 8.4 show erroneous student implementations. Although to a non-programmer these student programs may look very close to the reference solution, these programs demonstrate important errors and misconceptions about data reading, statement order inside loops, and program structure.

```
 1  min = MAX_VALUE;
 2  input = read();
 3  while (input != terminator)
 4  {
 5      if (min > input)
 6      {
 7          min = input;
 8      }
 9      input = read();
10  }
11  return min;
```

Figure 8.1: Reference solution for calculating the minimum value in a stream of numbers read from the console

```
 1  min = 0;
 2  input = read();
 3  while (input != terminator)
 4  {
 5      if (min > input)
 6      {
 7          min = input;
 8      }
 9      input = read();
10  }
11  return min;
```

Figure 8.2: Example of student code for which the correct fix entails changing the initial value stored in *min* at line 1 using statement-level edits

The student code shown in Figure 8.2 exhibits a bug that can be solved using

```
 1 min = MAX_VALUE;
 2 input = read();
 3 while (input != terminator)
 4 {
 5     input = read();
 6     if (min > input)
 7     {
 8         min = input;
 9     }
10 }
11 return min;
```

Figure 8.3: Example of student code for which the correct fix entails moving statement 5 after the *if* block

```
 1 min = 0;
 2 input = 0;
 3 if (input != terminator)
 4 {
 5     if (min > input)
 6     {
 7         min = input;
 8     }
 9     input = read();
10 }while(input != terminator);
11 return min;
```

Figure 8.4: Example of student code in which the student seems to have basic misconceptions about the structure of *ifs* vs loops

expression-level editing rules. In this case, the student was able to implement the loop correctly but did not realize how initializing min to 0 will impact the end result. Consequently, the code only works for cases when the minimum value happens to be 0 or negative. The fix involves modifying the statement in line 1 which initializes the variable used to store the minimum value.

Figure 8.3 shows student code in which the proper fix entails moving the statement in line 5 after the *if* block and therefore it can not be autogenerated using only expression-level edits. For this program, the proper repair requires statement-level reordering. This type of error may be due to misconceptions about control flow, and in particular about how reading input inside loops works, as pointed out by Soloway and Spohrer [95].

Lastly, Figure 8.4 shows student code with deep misconceptions, including about the proper structure of loops. Mutating if structures to while structures and while structures to if structures are common repairs in the datasets used in my research. For this kind of student error, the fix entails both expression-level editing (statement 1 and 2), as well as program structure editing of the if statement into a while statement (between statement 3 and the empty loop at line 10). Although the empty while loop in line 10 does not impact the behavior of the program, it should be deleted and the student should be informed about it being interpreted as an empty loop by the computer.

8.3 Program Repair Approach

The goal of the automatic program repair problem is to identify a set of changes that can turn a program that is incorrect concerning a given specification into a correct one. In the case of PR-CSF^2, the incorrect program is a student submission and the specification is the testcase set used to test the code.

In this section, I start by reasoning why autogenerating feedback to programming assignments fits as a search problem. Then, I explain what constitutes a valid program repair in Section 8.3.2. In Section 8.3.4, I describe the problem formally and in the context of the datasets I used in my research, followed by a description of the core of PR-CSF^2. To better understand how the search algorithm at the core of PR-CSF^2 works, I start by introducing its integral parts: how programs are represented in PR-CSF^2 in Section 8.3.4, how PR-CSF^2 decides where to make modifications inside of a program at each step and how it decides a modified program is a good candidate for finding a fix int Section 8.3.5, and what type of modifications PR-CSF^2 allows for in Section 8.3.6. Lastly, in Section 8.3.7, I outline how the search algorithm combines all these features to find repairs to student programs.

8.3.1 Why Use a Search Algorithm?

Providing Feedback as a Search Problem. Providing feedback to student programs has many characteristics of a search problem. The search objective is to identify a set of changes that would change the behavior of the incorrect program to the desired behavior, along with textual explanations that address students' misconceptions. Using their domain knowledge about specific programming artifacts coupled with their teaching expertise about students' misconceptions, teachers search through the space of possible modifications to the program and misconceptions linked to these modifications when providing feedback to student programs. PR-CSF^2 mimics this process by searching through the space of modified programs that were changed based on knowledge about the most common errors in student programs. These error patterns map to concepts and skills and point to the related misconceptions students may have about the assignment.

Search Method. The space of possible program modifications is infinite, but immediate feedback is most effective. Teachers and experts do it efficiently. Therefore, the search procedure must be efficient in finding the correct program as well. An effective search algorithm needs to avoid areas in the search space that do not contain a correct program and focus on search paths that are most likely to productively lead to a correct program. With every iteration, the algorithm needs to decide whether to *exploit* previously-evaluated good solutions or to *explore* new areas of the search space. PR-CSF^2's search algorithm uses different search methods for simple program modification versus complex modifications such as program structure modifications. For simple repairs, it discards all the modifications that negatively impact the program's overall testcases score and only further explores modifications that impact it positively. This hill-climbing search method provides *exploitation* of previously-evaluated good solutions. For complex repairs, it keeps all of the modifications and further explores them in order of their scores. The majority of the modified programs using complex edits end up with a lower score than their precursor, so exploring different pools of modified programs based on their modification history adds some randomness to the search procedure. The resulting pseudo-random search method provides *exploration* of the search space. This feature helps the search procedure move outside of the local optima search areas.

Furthermore, not all the repairs appear equally across assignments. When designing assignments, teachers select a subset of concepts and skills that are required for the assignment. Consequently, the search algorithm accounts for the variability across assignments by allowing the configuration of the *program repair units* to apply to a specific assignment. Different *program repair units* can be enabled or disabled based on the type of the assignment. By providing a configurable interface for combining simple repairs with complex repairs, I provide a robust search mechanism that effectively trades off exploration and exploitation and assignment-specific program repairs.

8.3.2 What is a Valid Repair?

PR-CSF^2 uses testcases to establish correctness, a common approach used by software engineers. I codify desired behavior by running the program on a given input, define the expected output on that input, and determine correctness based on whether the two match.

Correctness. Formally, a testcase t is a pair *(in, out)* where *in* is the input to the program and *out* is the expected output. A program P satisfies a testcase t if for its input *in*, the output of P matches the expected output *out* corresponding to the *in* of the testcase t. Furthermore, P satisfies the entire testcase set T and is deemed **correct** if it satisfies all the testcases t of the set T.

Thus, in addition to the student program, PR-CSF^2 also takes as input a set of testcases T which can be hand-written or automatically generated. In my research, all of the testcase sets were compiled manually. A key desirable feature of the testcase set is to closely describe the desired behavior of the program as I emphasize in Section 4.2. I call the failing testcase subset as the *negative testcases*, and it encodes the defects under repair. Conversely, I call the passing testcase subset as the *positive testcases*, and it encodes functional and non-functional program requirements. Both types of testcases are typically required to guide the search to an adequate repair, except for a few cases which I discuss next.

8.3.3 Problem Statement

Formally, the problem of repairing student programs is as follows: given a student program P and a testcase set T such that P does not satisfy T, let *M(P)* be the set of all programs that can be obtained by modifying the program using a specific set of rules which I describe in Section 8.3.4. The goal is to synthesize program S, such that S satisfies T and S is in *M(P)*.

However, not all student program repairs can be captured by the test score as pointed out earlier. Therefore, PR-CSF^2 can be configured for *program repair mode* or *feedback mode*. The only difference between the two is that in *feedback mode*, PR-CSF^2 tries to further simplify students' programs beyond functional correctness. For example, even if a student's program passes all the testcases, their program may still have stylistic issues such as using a loop to check a condition once and then breaking out of the loop at the end of the loop. While this does not impact the functional correctness of the program, it is a bad coding practice that may be connected to misconceptions and can negatively impact students' programming in the future.

In *program repair mode* at least one *negative testcase* is needed. Moreover, for the majority of the types of program repairs, except for input/output (IO) repairs, at least one, and typically several *positive testcases* are needed. For the datasets I use in my research, students were required to read input, output results, and signal bad input by using an IO module provided to them by the teacher. Especially for earlier assignments, many students had issues understanding how to properly use the IO module and for that reason, their code was not able to pass any testcases. To address those cases, the search algorithm first does a search phase focused on specifically repairing IO issues in the student's program. This search phase is guided by the number and type of expected inputs, expected outputs, and expected bad input signals. At the end of this search phase, if the student program passes at least one testcase, the search algorithm proceeds to search for the other types of repairs, otherwise, it terminates.

8.3.4 Program Representation

Program Abstraction. Any programming language has an explicit context-free grammar (CFG) that can be used to parse source code into an AST (abstract syntax tree). The AST represents the abstract syntactic structure of source code. An AST is a tree where each non-leaf node corresponds to a non-terminal in the CFG specifying structural information, for example, a *while* statement or an *if* statement. Each leaf node corresponds to a terminal, for example, variable names and operators, in the CFG encoding of the program's text. Figure 8.5 shows the AST corresponding to the reference code shown in Figure 8.1.

In PR-CSF^2, program edits are performed on the AST of the program as further

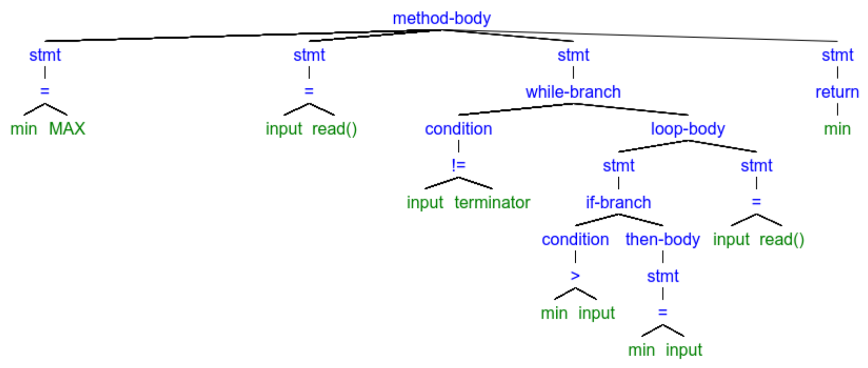


Figure 8.5: AST corresponding to the reference solution code shown in Figure 8.1

described in Section 8.3.6. Therefore, each program is represented as a tuple of:

- An AST including all of the statements and expressions in the program.
- A test score representing the weighted sum of the passed testcases.
- A progress score representing the sum of the progress scores for each statement. The progress score for each statement is calculated using the formula from the Tarantula fault localization system [62].

The test score is used to check if a candidate fix is correct as described in Section 8.3.2. Both scores are used in deciding what is the best location to make modifications in a program next, and in determining if a partially modified program is on the path of finding the correct program as explained in the next section.

8.3.5 Fault localization and Fitness Function

The search space of modified versions of a program is very large. An effective search algorithm needs to avoid areas in the search space that do not contain a correct program and focus on search paths that are most likely to productively lead to a correct program. PR-CSF^2's search algorithm prunes the search space by leveraging fault localization techniques and a fitness function to decide if a modified program is on the path towards a correct program.

Fault Localization. PR-CSF^2's search algorithm prioritizes search areas based on information from the faulty and successful executions of the student program. For each execution, it marks the statements on its path as successful or faulty based on the outcome of the execution. Then, based on the successful and faulty score of each statement, it calculates the progress score using the formula used by the Tarantula fault localization system [62]. Similar to previous techniques [71], it uses the progress score as an indicator for where the defect might be in the student program. Locations with a lower progress score are considered most likely to be responsible for the defect. Therefore, expressions and statements are picked for modification in order of their progress score. The expressions and statements with lower progress scores are modified first. However, prioritization alone does not reduce the amount of candidate modified programs to search through. The search algorithm needs a way to decide which search paths to ignore in the future, a process I describe next.

Fitness Function. PR-CSF^2's search algorithm focuses on paths that are more likely to lead to a correct program by using a multi-tier fitness function. The fitness function combines the type of the modification, the test score, and overall progress score to decide whether to keep a modified program for further modifications. A modified program is considered fit for further modifications if its test score is higher than the score of the initial student program. Additionally, based on the type of modification, the search algorithm decides whether to keep other modified programs. Modifications that change the program structure are going to results in lower test scores in many cases. The search algorithm keeps all of the programs modified using program structure edits. Subsequent modified programs' test scores are compared to the test score of the previous program on which the last modification was performed. Lastly, some modifications may not be reflected in the test score in the partially modified programs, but they may reflect later on the search paths after other edits have been performed to the program. For these cases, the search algorithm uses the overall progress score of the program to decide whether to keep the modified program.

Next, I describe how PR-CSF^2 generates modifications to student programs and how it searches for the correct program in the space of modified versions of the start program.

8.3.6 Generating Modifications to Student Programs

PR-CSF^2 allows for two types of modifications, which I refer to as **statement-level modifications** and **program structure modification**. Statement-level modifications are similar to the rule-based edits and syntax transformations that have been explored by previous works [37, 21]. More specifically, the **statement level edits** that PR-CSF^2 explores are:

- **IO read, output and bad input calls (*inputMissing, inputBadType, outputMissing, outputBadType, addBadInput*)** can be replaced with method calls of the same type, or added to the program.

- **Integer and Double Variable Types** can be interchanged. This change is needed to accommodate the input and output type changes. For example, if the student program reads an integer value and stores it into an integer variable, changing it to read a double value and to store it in the same variable entails changing the variable type to a double.

- **Math.abs (*absolute_value*) and Math.ceil (*rounding*)** can be applied to variables, literals, and expressions. I selected these methods because of their expressive power.*Math.abs* transforms an integer into a natural number, and *Math.ceil* transforms a decimal into a whole number.

- **Algebraic, Logical and Comparison Operands (*ops*)** can be replaced with operands of the same type. Additionally, an expression with a division operand can be modified to a double division by adding the double cast to one of its sides.

- **Variables (*vars*) and Literals (*literals*)** can be replaced with other variable, literals or any expressions in the same domain from students' program

- **Expressions (*exprs*)** can be replaced with any other expression in the same domain from students' program or with expressions auto-generated using expressions from student's program and operands in the same domain.

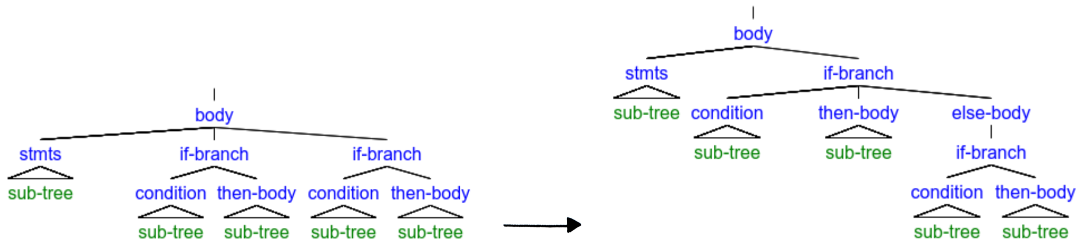


Figure 8.6: AST transformation corresponding to the *if_else_if* program structure edit.

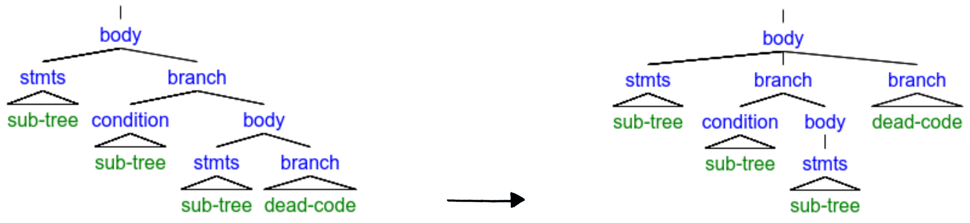


Figure 8.7: AST transformation corresponding to the *dead_code* program structure edit.

Statement-level edits *ops, vars, literals, and exprs* can be applied to the right-hand side of assignment statements and conditional expressions. When edits are applied to the right-hand side of the assignment statements, the correction is labeled as **correct_formula**. When edits are applied to a conditional operand inside a conditional expression, the correction is labeled as **correct_boundary**. For all the other changes to the literals, variables, and logical operands of a conditional expression, the correction is labeled as **correct_condition**.

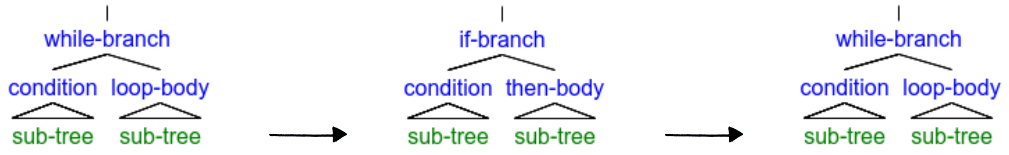


Figure 8.8: AST transformation corresponding to the *while_to_if* and *if_to_while* program structure edit.

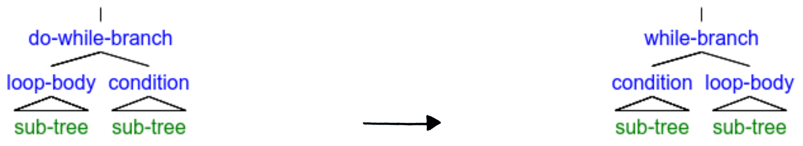


Figure 8.9: AST transformation corresponding to the *do_while* program structure edit.

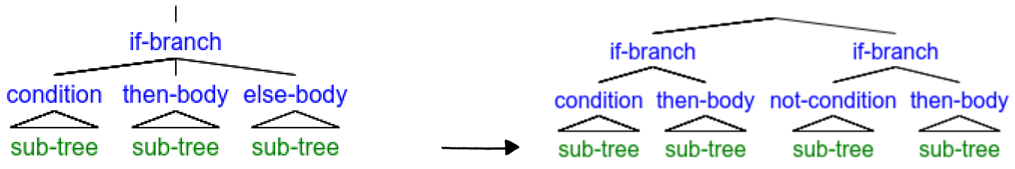


Figure 8.10: AST transformation corresponding to the *if_else_split* program structure edit.

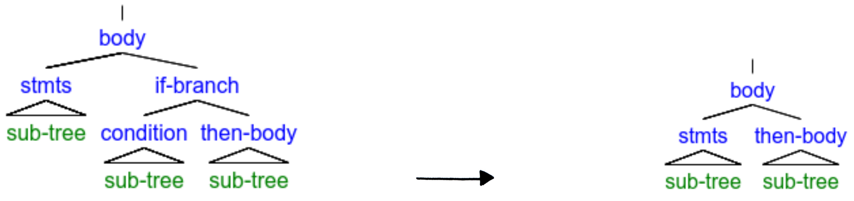


Figure 8.11: AST transformation corresponding to the *remove_branching_if* program structure edit.

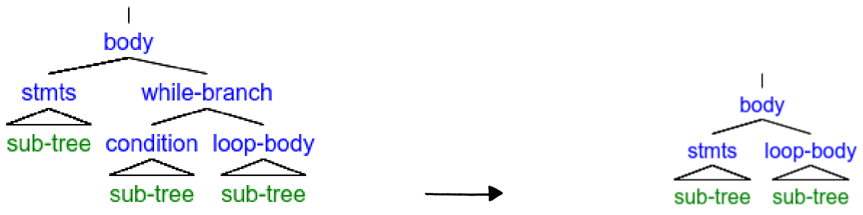


Figure 8.12: AST transformation corresponding to the *remove_branching_while* program structure edit.

Allowing for additional program structure edits is what differentiate PR-CSF^2 from previous program repair solutions. In PR-CSF^2 any statement can be **deleted or reordered** within the block that they are in. **Delete** iteratively picks a combination of statements inside a block and removes them until it exhausts all combinations. **Reorder** moves all the statements inside of a block by repetitively permuting their indexes within the block. Additionally, the structure of the program can be further modified using the following **program structure edits**:

- **Successive if statements can be combined into an if-else-if structure (*if_else_if*).** This edit is an assignment-specific edit which points to misconceptions about the control flow of the program. For this edit, sub-trees corresponding to *if* branches to the right of an *if* branch node are moved to its *else* body in the AST as shown in Figure 8.6.

- **Dead code can be moved to other blocks in the program (*dead_code*).** In some situations, parts of the code are placed in the program in such a way that it never gets executed. Moving these parts outside of their parent block may allow them to be executed and bring the program closer to being correct. This edit takes the structure sub-tree containing the dead code (*if* or loop structure) and moves it in all possible locations in the body of other structures in the AST. An example of an AST transformation corresponding to this type of program edit is shown in Figure 8.7.

- **Loops can be transformed to ifs and vice versa (*while_to_if* and *if_to_while*).** Often students use while loops and if statements interchangeably regardless of the program descriptions which points to misconceptions about translating word problems into code. In the AST, *if* and *while* structures are represented similarly and so performing these edits entails only changing the type of the branching node and the type of the body node as shown in Figure 8.8.

- **Do-while statements can be transformed to while loops (*do_to_while*).** Some student programs use a do-while when a while loop structure is more appropriate which points to misconceptions regarding when the condition of the loop gets checked. In the AST, this edit involves moving the condition sub-tree before the loop's body sub-tree as shown in Figure 8.9.

- **If-else statements can be split into two ifs (*if_else_split*).** This issue is related to other control flow issues; the repair breaks the dependency between the *then* and the *else* blocks. For this edit, a new sub-tree is added to the right of the *if* branch node representing an *if* branch that has as condition the negated condition of the former *if* branch and the *else* body sub-tree as its *then* body sub-tree as shown in Figure 8.10.

- **If statement conditions can be removed (*remove_branching_if*).** This issue is similar to the dead code issue. In some cases, additional unnecessary checks, for example, checking if a value is positive, may hinder the functionality of the code and point to misconceptions about the assignment. This edit necessitates the removal of the branch parent node along with the condition sub-tree as shown in Figure 8.11.

- **Loop conditions with break statements can be removed (*remove_branch -ing_while*).** Many student programs use a loop, often with a *true* condition, even when reading only one value from the IO. They also use a break at the end of the loop to break out of the loop after one iteration. I believe this issue points to misconceptions about reading input or about the assignment description. The AST transformation corresponding to this program edit is shown in Figure 8.12.

Allowing for these kinds of edits can dramatically change the semantics of a program and cause an explosion of the search space. Moreover, some of the edits are complements of each other, and allowing for two complement repairs to be applied on the same statement can cause a statement to be changed back and forth. To prevent it from happening, PR-CSF^2 does not apply a new edit to a statement or expression that was previously edited as I further explain in the next section.

8.3.7 Search Algorithm for Fixing Student Programs

Algorithm 1: Student Program Repair Search Algorithm

```
Require: $P$, student program
Require: $T$, testcases set that describes the behavior of the expected program
Ensure: $F$, partially or completely repaired program
1: $F = P$
2: add $P$ to $C(P)$
3: repeat
4:   $stmts = getNextMostSuspectStmts(P)$
5:   $candidates = allCandWithStruct(C(P)) \cup maxCandWithStmt(C(P))$
6:   for $candidate \in candidates$ do
7:     for $edit \in StructEdits = (delete, reorder, if\_else\_if, etc)$ do
8:       apply $edit$ to $stmts$ in $candidate$
9:       if $correct(candidate')$ then
10:        return $F = candidate'$
11:      end if
12:      if $fit(candidate')$ then
13:        add $candidate'$ to $C(P)$
14:      end if
15:    end for
16:  end for
17:  for $candidate \in candidates$ do
18:    for $edit \in StmtEdits = (ops, vars, literals, exprs)$ do
19:      for $stmt \in stmts$ do
20:        $candidate'$ apply $edit$ to $stmt$ in $candidate$
21:        if $correct(candidate')$ then
22:          return $F = candidate'$
23:        end if
24:        if $fit(candidate')$ then
25:          add $candidate'$ to $C(P)$
26:        end if
27:      end for
28:    end for
29:  end for
30:  $F = candidateWithMaxScore(C(P))$
31: until $candidates$ is empty or $stmts$ is empty
32: return $F$
```

The core algorithm for finding corrections to student programs is described in Figure 1. It uses a priority queue $C(P)$ with the candidate fixes ordered based on their type and their test score. The priority queue $C(P)$ is seeded with the student program P (Line 2), and then the algorithm iteratively applies edits to P or its previously generated variants (candidate fixes) until it finds a correct program or there are no more candidate programs or no more suspect statements to edit (Lines 2 to 32).

At every iteration step, the search algorithm first picks the next most suspect locations in the program where to perform edits (Line 4). Then it selects the previously generated variants that are the best candidates for identifying the corrections (Line 5). From the set of programs with statement edits, only those with the maximum score are allowed to be further explored as a way to prune the search space. All programs with structure edits are allowed to be further explored because changing the program's structure generally does not immediately impact the test score positively, except for delete and reorder edits.

Throughout the search, the algorithm checks programs with structure edits (Lines 6 to 17) interleaved with programs with statement edits (Lines 18 to 31). For every new candidate program obtained by applying a specific edit rule, the algorithm first checks if it's correct by running the program with all the testcases (Lines 9 and 22). If the candidate is correct the algorithm returns it and terminates. If the candidate is not correct, the algorithm then checks if the program is fit for further exploration using the fitness function described in Section 8.3.5 (Lines 13 and 26), and saves it if it's deemed fit.

8.4 Implementation

PR-CSF^2 was implemented in Java. It uses the JavaParser library [96] to generate the Abstract Syntax Tree (AST) of the program and to perform modifications to programs. Each program repair edit is implemented using the *"visitor pattern"* in JavaParser library. The visitor pattern is a design pattern commonly used in the parser of a compiler. It provides a versatile way of traversing the AST and "visiting" each node in the AST. When visiting a node, the code first decides if the node is a target node for the program repair and then it performs the necessary modifications locally.

Lastly, PR-CSF^2 uses the JUnit framework [97] for testing each program. JUnit provides a RunListener class to monitor JUnit tests during runtime. PR-CSF^2 extends this class to capture information about the performance of the program during testing, such as passed and failed testcases. This information is used to guide the search for the correct program.

8.5 Discussion: Modifications to PR-CSF^2

Additional Repair Patterns. Although PR-CSF^2 can identify additional program repairs and provide feedback about the structuring of student programs, the overall set of repairs is not complete for all assignments. More research on different assignments and datatsets of student programs is needed to compile all the program repairs patterns and how to identify them in student programs.

Partial Correctness. After every search phase, the search algorithm saves the program with the highest score. For student programs that are partial solutions or have deep design flaws, PR-CSF^2 may not be able to find a program that passes all of the testcases. However, it may be able to find a repair to the partial solution and give students repair feedback on it along with information about missing tasks. Additional research is needed to assess if the partial feedback generated this way is accurate.

Search Technique. Hill-climbing procedures can get stuck in local optima and PR-CSF^2's search procedure is hill climbing in the statement-level edits phases. As a result, this search procedure can not capture program repairs that work together to improve the modified program's test score, when these repairs are not able to improve the test score on their own. For example, if a program checks if the value stored in a variable is in different ranges, changing the range boundary between two ranges involves changing the comparison for both ranges on each side of the boundary. Changing only one comparison at a time may not reflect a positive change in the test score for the modified program and therefore be discarded. Adding some randomness to the search procedure in these phases can improve the success rate of the search procedure.

Fitness Function. Related to the hill-climbing issue is the issue that using test scores as part of the fitness function is too coarse to capture repairs that do not immediately improve the test score. For that reason, PR-CSF^2 uses the sum of the progress score of each statement as a fitness function. However, it only works for modified programs with the same program structure and as long as the program edits do not negatively impact the test score. For the example in the previous paragraph, this means that the repairs will only be done in a specific order. Suppose the boundary needs to be moved slightly to the left by one. This repair may be achieved by changing the comparison operands on both sides of the boundary. The search procedure will only find the correct repair if it changes the comparison on the right in the first step which will have an overlapping effect between the two ranges, and then change the comparison on the left which will get rid of the range overlap and will correctly accomplish the boundary change. However, if there is interdependence between the code blocks executed for each range, then the overlapping effect may still negatively impact the test score and in turn the overall progress score of the program. In this case, the search procedure will still fail to find the fix. Using more continuous fitness functions such as the sigmoid function in combination with a random search may improve the success rate of the search method.

Implementation. Lastly, PR-CSF^2's search algorithm is sequential, but it can be easily configured to run in parallel. Different threads can search through various paths in the search space in parallel and share information at the end of each phase, which may reduce the time to find a correct program and increase the success rate of the program repair. I plan to explore these research directions in my future work.

Chapter 9

How does PR-CSF^2 Fare Compared to Equivalent Program Repair Techniques?

In this chapter, I show how allowing for program structure repairs improves the success rate of previous program repair approaches that only allowed for statement-level repairs or rule-based repairs [37, 38]. Then, I show the distribution of repairs for a subset of the assignments and explain what misconceptions the feedback should target for each repair. Lastly, I provide examples for which using program structure repairs lead to stylistically more desirable repairs.

I describe the benchmark set in the next section, the methodology in Section 9.2, and I report on findings of the study in Section 9.3.

9.1 Benchmarks

The benchmarks set used for this study is comprised of problems taken from the Introduction to Computer Science course (CS111) at Rutgers University offered over seven semesters between Spring 2016 and Spring 2019 inclusively. It includes all the problems from the first six weeks of the course. Below, I provide a brief description of each problem:

- *Sum* - read two integers from the Input-Output (IO) module provided by the teacher and output their sum to the IO module.
- *Multiply* - read two integers from the IO module and output their product to the IO module.
- *BuyingApples* and *BuyingTomatoes* - read two doubles from the IO module representing the cost per pound and weight in pounds and return the total cost to the

IO module. If either of the doubles is <= 0, signal bad input to the IO module.

- *Gas* - read two doubles from the IO module representing the cost per gallon, and the quantity in gallons, as well as a boolean value signaling if the payment is made with a credit card or not, and return the total cost to the IO module. Credit card payments incur a 10% surcharge of the total cost. If either of the doubles is <= 0, signal bad input to the IO module.

- *TrainTicket* - read two boolean values from the IO module to signal if the buyer is a senior and if the ticket purchase is on board the train. Return the total cost of the ticket after applying the appropriate discounts and surcharge - the discount for being a senior and the surcharge for buying on board the train, and return it to the IO module.

- *Party* - read two doubles from the IO module representing the cost of a pizza and the cost of a case of soda, and five integers representing the number of people, slices per person, sodas per person, number of slices in a pizza and number of soda cans in a case. Return the total cost for the pizza and soda to the IO module.

- *PayFee* and *PayFriend* - read a double from the IO module representing a payment and return the fee associated with the payment to the IO module. The fee needs to be calculated using a multi-tier fee system provided to students.

- *LuckySevens* - read two integers from the IO module representing a range, and for all the numbers in the range count the number of times digit 7 appears in each number, and return the total count to the IO module.

- *TwoSmallest* - read a sequence of doubles from the IO module which starts and ends with a terminating value and output the two smallest values to the IO module. If a terminating value is read before two valid values, the code needs to signal bad input to the IO module and continue reading.

9.2 Methodology

For this study, I ran PR-CSF^2 on benchmarks containing code submitted by students for multiple programming assignments in two modes:

- **statement-level repair mode (SRM)** in which I disabled all the program structure repairs and only allowed for statement-level repairs described in Section 8.3.6.
- **program structure repair mode (PRM)** in which I allowed for all the repairs that applied to a specific assignment.

For each student program, I ran the search algorithm twice, once in each of the modes. Experiments were performed on a 3.4GHz Intel® Core™ i7-6700 CPU with 8 cores and 16GB RAM. For each experiment, I recorded the **number of lines of code (LOC)** of the student program, the total **number of searches (NOS)** the algorithm performed, the **total runtime (RT)** and the subset of repairs used to correct the program. After running all of the experiments for each assignment, I counted the number of submissions successfully repaired, and averaged the LOC, NOS, and RT for each mode. Then, for each type of repair, I counted how many times the repair was used to repair a program for one semester, for a subset of the assignments. Additionally, I manually analyzed a subset of the repaired programs (25 for each assignment), with a focus on those programs that were repaired differently with and without program structure repairs enabled. I present and discuss these statistics and findings of the manual analysis in the next section.

9.3 Results

In this section, I compare the success rate and performance of PR-CSF^2 configured in the two modes discussed earlier I present the distribution of the repairs for one semester for a subset of the assignments and conclude with observations about the quality of the program repairs between the two modes. For each assignment, I enabled only the program repairs specific to the code structure of the solution. For example, *PayFriend* requires the use of multiple independent *if* structures. For this assignment, I enabled the *if_else_if* program repair which transforms sequential *if* statements into an *if-else-if* structure. Reorders and deletes were enabled for all assignments.

9.3.1 Success Rates and Runtimes Statistics

	Total	**Attempted**	**PRM**		**SRM**	
			Count	%	**Count**	%
Sum	4541	**1302**	1245	**95.6%**	1226	**94.2%**
Multiply	4030	**993**	914	**92.0%**	705	**71.0%**
BuyingApples	1556	**319**	299	**93.7%**	253	**79.3%**
BuyingTomatoes	1141	**237**	213	**89.9%**	195	**82.3%**
Gas	1662	**971**	871	**89.7%**	827	**85.2%**
TrainTicket	869	**506**	469	**92.7%**	439	**86.8%**
Party	7516	**3429**	2699	**78.7%**	2576	**75.1%**
PayFee	2655	**1550**	1031	**66.5%**	975	**62.9%**
PayFriend	5484	**3421**	2444	**71.4%**	2117	**61.9%**
LuckySevens	6699	**2576**	1234	**47.9%**	1219	**47.3%**
TwoSmallest	7565	**4057**	955	**23.5%**	644	**15.9%**

Table 9.1: Table showing the total number of student programs, the number of programs attempted to be repaired (must compile), and the number of programs repaired and the percentage it represents from the total attempted for each mode

Success Rate. Table 9.1 shows the total number of student submissions, the number of student submissions attempted to be repaired (must at least compile), and the count and percentages of student programs repaired in each mode for each assignment. For some assignments, we see a significant increase when the program structure repairs are enabled, in particular assignments with a complex program structure such as PayFriend. PayFee was designed as an alternative to PayFriend to be offered in different semesters. However, we do not see an equivalent increase with PayFee as with PayFriend. Based on my manual investigation, I discovered that many of the PayFee programs contain the exact or similar code as PayFriend, which points to plagiarism across semesters. Correcting these programs would require deep problem-specific repairs which are not covered by PR-CSF^2's technique.

The success rate drops for complex assignments such as **LuckySevens** and **TwoSmallest** that require multiple non-trivial tasks to be executed one after another or in parallel. For example, **TwoSmallest** requires inspecting the numbers in a stream of inputs, extract those that are not equal to the terminating value and find the two smallest values among them. Many student programs do not attempt to implement all the tasks or have deep design flaws. PR-CSF^2 does not add functionality to student programs and does not target deep design flaws. Thus, it can not find a repair for many of the student programs submitted for these assignments.

Assignment	**Average LOC**	**Average NOS**		**RT (secs)**	
		PRM	**SRM**	**PRM**	**SRM**
Sum	11.6	7.7	7.4	1.9	1.9
Multiply	12.4	14.3	13.1	3.1	2.7
BuyingApples	16.6	13.6	13.0	3.0	2.9
BuyingTomatoes	16.2	8.0	8.1	1.9	1.9
Gas	23.3	31.1	29.7	7.2	6.8
TrainTicket	22.5	28.7	25.2	6.6	5.7
Party	32.1	487.9	480.3	85.6	86.1
PayFee	41.2	318.6	297.0	56.9	52.4
PayFriend	42.5	522.6	443.2	94.7	88.7
LuckySevens	18.0	17.3	17.5	4.0	3.9
TwoSmallest	46.1	533.0	352.2	96.7	72.3

Table 9.2: Table showing the average lines of code (LOC), the average number of searches (NOS) and the average runtime (RT) in seconds for both configuration for each assignment in the benchmark.

Runtime Stats. Table 9.2 shows the average number of lines of code, the average number of trials, and the average run time for each of the modes. As expected, there is an increase in the average number of trials and the average runtime from SRM mode to PRM mode because in PRM mode more edits are allowed. Additionally, for most assignments, the average number of trials increases with the number of lines of code. However, these trends are not consistent because the search repair algorithm can take different paths based on the mode configuration for the same program. Moreover, in PRM mode the search space increases with the structural complexity of the program which does not necessarily correlate to the number of lines of code.

9.3.2 Repairs Distribution Across Different Assignments

Tables 9.3, 9.4, 9.5, 9.6 and 9.7 show the distribution of different repairs described in Section 8.3.6 for a subset of the assignments. Note that multiple repairs can be used to correct one student program, and for that reason, their cumulative percentage for each assignment is greater than 100% of the total repair programs. For many assignments, a large percentage of the repairs are regarding reading, outputting results, and signaling bad input to the IO module. Moreover, a large percentage of the repairs across assignments are regarding correcting boundaries that are performed by modifying the conditional expression inside loops and *if* statements. An example of this repair is replacing a $<$ operand to a $<=$ operand. Lastly, these observations explain why the success rates are not significantly greater when program structure repairs are enabled since these repairs are counted as statement-level repairs.

Repair	**Programs Count**	**%**
inputMissing	9	1.9%
inputBadType	211	44.9%
outputMissing	404	86.0%
outputBadType	58	12.3%
remove_branching_if	4	0.9%
remove_branching_while	2	0.4%

Table 9.3: Table showing the distribution of repairs for one semester for Sum assignment. A total of 470 student programs were repaired by PR-CSF^2. Note that multiple repairs can be applied to the same student program.

Repair	**Programs Count**	**%**
inputMissing	7	2.3%
inputBadType	60	20.1%
outputMissing	152	50.8%
outputBadType	43	14.4%
addBadInput	19	6.4%
correct_formula	26	8.7%
correct_boundary	35	11.7%
while_to_if	46	15.4%

Table 9.4: Table showing the distribution of repairs for one semester for BuyingApples assignment. A total of 299 student programs were repaired by PR-CSF^2. Note that multiple repairs can be applied to the same student program.

For *Sum*, the two program structure repairs *remove_branch_if* and *remove_branch_while* shown in Table 9.3 are needed in cases where the program does additional checks of the input, for example, it checks if the input is greater than zero. These repairs point to misconceptions about the scope of the assignment and potentially about data types.

For *BuyingApples*, the only program structure repair in Table 9.4 is *while_to_if* which transforms a *while* loop into an *if* structure. For those cases, the buggy program reads input until it reads a non-negative value, which is not the expected behavior. The program is expected to read only one value, signal bad input if the value is negative, and terminate. Similarly to *Sum*, this repair points to misconceptions about the problem specification.

Repair	**Programs Count**	**%**
inputMissing	2	0.7%
inputBadType	161	58.8%
outputMissing	22	8.0%
outputBadType	52	19.0%
double_vs_int_div	112	40.9%
rounding	86	31.4%
correct_condition	3	1.1%
correct_formula	9	3.3%
remove_stmt_while	5	1.8%

Table 9.5: Table showing the distribution of repairs for one semester for Party assignment. A total of 274 student programs were repaired by PR-CSF^2. Note that multiple repairs can be applied to the same student program.

For *Party*, there is a variety of repairs used to correct student programs although the majority of them are statement-level repairs as shown in Table 9.5. Besides IO repairs, many of the repairs are computational and centered around rounding the result of a division between integer values. The program structure repair *remove_stmt_while*, removes *while* loops which perform additional checks that change the behavior of the program. They point to misconceptions about the problem specification.

For *PayFriend*, the most common program structure repair is *if_else_if* shown in Table 9.6. It transforms consecutive *if* statements into an *if-else-if* structure. This modification points to misconceptions about control flow, lack of proper testing or misconceptions about the problem specification. Corrective information provided as feedback to students should also address all these misconceptions.

Repair	**Programs Count**	**%**
inputMissing	15	6.1%
inputBadType	81	32.8%
outputMissing	71	28.7%
outputBadType	133	53.8%
if_else_if	52	21.1%
change_boundary	144	58.3%
correct_condition	56	22.7%
correct_formula	132	53.4%
remove_stmt_if	6	2.4%
reorder_dead_code	7	2.8%
remove_branching_while	33	13.4%

Table 9.6: Table showing the distribution of repairs for one semester for PayFriend assignment. A total of 247 student programs were repaired by PR-CSF^2. Note that multiple repairs can be applied to the same student program.

Repair	**Programs Count**	**%**
outputMissing	7	5.8%
correct_condition	68	56.7%
correct_formula	4	3.3%
remove_stmt_if	5	4.2%
reorder_read	6	5.0%
if_to_while	17	14.2%
do_to_while	14	11.7%

Table 9.7: Table showing the distribution of repairs for one semester for TwoSmallest assignment. A total of 120 student programs were repaired by PR-CSF^2. Note that multiple repairs can be applied to the same student program.

For *TwoSmallest*, the most common program structure repairs are *if_to_while* which transforms an *if* into a *while* loop and *do_to_while* which transforms a *do-while* loop into a *while* loop, as shown in Table 9.7. These repairs point to misconceptions about loops, in particular reading and control flow in loops, or misconceptions about the assignment specification. Therefore, student feedback should contain textual information about both.

9.3.3 Repairs Quality Comparison

Multiple program implementations can be functionally correct, but stylistically undesirable as previously observed [21, 17]. The search algorithm of PR-CSF^2 interleaves program structure modifications with statement-level modifications as described in Section 8.3.7. Therefore, when program structure repairs are enabled, the search algorithm may take a different path than when they are disabled and consequently arrive at a different program implementation. For this reason, I have manually analyzed **25 student programs** from each assignment in the subset (a total of **125 student programs**) and compared the repaired programs generated when program structure repairs were enabled and disabled.

The main difference between the repaired programs using the two different configurations is that the programs repaired in SRM configuration are sometimes stylistically undesirable compared to their counterparts repaired in PRM configuration. For example, with the SRM configuration, a statement or a structure that needs to be removed is changed so that it does not impact the program behavior negatively. For *assign* statements this could mean assigning a variable to itself. For *if* statements or loops, it could mean changing the condition to *false*. Similarly, if only the condition of an *if* statement or a loop needs to be removed, the repair with the SRM configuration would relax it to the point where it would not make sense, for example by changing it to *true*. Both kinds of repairs are stylistically undesirable and in the least can confuse students or even hinder their learning. Out of the 125 student programs I manually analyzed, I identified **22 programs (17.6%)** repaired using the SRM configuration that exhibit this kind of stylistic undesirable features.

To summarize, allowing for program structure edits expands the search space for finding corrections to student programs. Consequently, in some cases, the search procedure has to search through more program variants and spend more time looking for the correct program. However, the search algorithm can find a correction to more student programs and the repaired programs are stylistically more appropriate than their counterparts repaired with statement edits only.

Chapter 10
Conclusions

10.1 Summary

In this dissertation, I present a mixed-approach methodology that bridges the gap between autograding and the knowledge assessment of programming assignments to provide meaningful feedback to students.

The first approach asks the instructor to systematically analyze programming assignments with respect to knowledge maps to ensure course cohesion between the specific challenges posed to students by the programming assignments and the material taught in class. The methodology also outlines an approach for finding common errors in a set of submissions for an assignment and generating an error classifier and hints that can be used by an autograder to give feedback on future submissions. This process can also give instructors insight into how to adjust class material to address knowledge deficiencies. I have applied this methodology to two assignments and found some evidence suggesting that the hints provide useful feedback for many students to make progress after submitting incorrect programs.

The second approach offers a set of program repairs that map to misconceptions students may have about the structuring of their code and other programming concepts. Instructors can select the program repairs to apply to students' buggy programs based on the concepts and skills that apply to the assignment. This feature guides the repair search algorithm towards more assignment-specific repairs. Additionally, they can write feedback for each repair aimed at the misconceptions they point to. Thus, students will receive both corrective feedback on how to correct their code as well as formative feedback related to the misconceptions. Moreover, teachers will gain an understanding of the misconceptions students may have about the material taught in the course. I have applied this methodology to multiple assignments from a CS1 course and discovered that this methodology is more successful at repairing student programs than previous approaches. Through manual investigation of the programs, I also discovered situations in which the repairs using my approach were more stylistically appropriate than with previous approaches.

10.2 Impact and Future Work

The latest trend of high enrollments in CS programs and courses has generated a plethora of solutions for providing automated feedback to student programs which I summarize in Chapter 2. These solutions explore the trade-off between requirements and the quality of the feedback provided. This dissertation presents a solution that is similar to previous approaches. However, one important benefit of my approach is that it explicitly tries to bridge the teaching of abstract concepts with hands-on programming practice. Connecting concepts and skills with practice has been recognized for its positive impact in deepening students' understanding of the programming concepts [98]. Therefore, I believe this is a good research direction in this context, primarily because it resembles the process instructors follow when providing feedback manually. However, some challenges arise from my research which I discuss next, along with suggestions for future work.

Firstly, there is limited support in CS education for connecting concepts and skills with practice. By using my framework, teachers have a systematic way to fill in the gap between lecture material and hands-on practice. However, the case studies I present in my dissertation are intended as proof of concept and as a guide and are not meant to provide a comprehensive solution. The assignments used in my analysis are relatively simple assignments used in an introductory computer science course. Further research is required to understand the mistakes students make in their programs, how to identify them and what feedback works best in addressing the misconceptions associated with them for assignments from different CS courses and levels.

Secondly, although the results of my analysis and evaluation of the feedback and its impact on students and teachers suggest that it is useful for learning and teaching, the evaluation has its limitations. Focused user studies along with using metrics that measure the actual learning are required to further understand the impact the feedback provided by my proposed mixed approach has on students' learning. In parallel, a related future work direction aims to understand how teachers use the framework and the specific impact it has on their teaching.

References

[1] J. Haskel and S. Westlake, *Capitalism without capital.* Princeton University Press, 2017.

[2] National Center for Women & Information Technology, “Computing Education and Future Jobs: A Look at National, State, and Congressional District Data.” `https://www.ncwit.org/edjobsmap`.

[3] N. A. of Sciences Engineering, Medicine, *et al.*, *Assessing and responding to the growth of computer science undergraduate enrollments.* National Academies Press, 2018.

[4] C. P. for Computing and T. S. Development, “Low student engagement, manual grading pose key challenges for cs instructors,” 2019.

[5] C. E. Kulkarni, M. S. Bernstein, and S. R. Klemmer, “Peerstudio: rapid peer feedback emphasizes revision and improves performance,” in *Proceedings of the second (2015) ACM conference on learning@ scale*, pp. 75–84, 2015.

[6] H. Keuning, J. Jeuring, and B. Heeren, “Towards a Systematic Review of Automated Feedback Generation for Programming Exercises,” in *Proceedings of the 2016 Conference on Innovation and Technology in Computer Science Education*, 2016.

[7] B. S. Bloom, “The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring,” *Educational researcher*, vol. 13, no. 6, pp. 4–16, 1984.

[8] S. H. Edwards and M. A. Perez-Quinones, “Web-CAT: Automatically Grading Programming Assignments,” in *ACM SIGCSE Bulletin*, vol. 40, 2008.

[9] C. Douce, D. Livingstone, and J. Orwell, “Automatic test-based assessment of programming: A review,” *Journal on Educational Resources in Computing (JERIC)*, vol. 5, no. 3, pp. 4–es, 2005.

[10] D. Milojicic, “Autograding in the Cloud: Interview with David O’Hallaron,” *IEEE Internet Computing*, 2011.

[11] K. Rivers, E. Harpstead, and K. Koedinger, “Learning curve analysis for programming: Which concepts do students struggle with?,” in *Proceedings of the 2016 ACM Conference on International Computing Education Research*, pp. 143–151, ACM, 2016.

[12] K. Rivers and K. R. Koedinger, “Data-Driven Hint Generation in Vast Solution Spaces: A Self-Improving Python Programming Tutor,” *International Journal of Artificial Intelligence in Education*, vol. 27, 2017.

[13] J. Stamper, T. Barnes, L. Lehmann, and M. J. Croy, "The hint factory: Automatic generation of contextualized help for existing computer aided instruction," 2008.

[14] B. Paaßen, B. Hammer, T. W. Price, T. Barnes, S. Gross, and N. Pinkwart, "The continuous hint factory - providing hints in vast and sparsely populated edit distance spaces," *CoRR*, vol. abs/1708.06564, 2017.

[15] A. Gerdes, B. Heeren, J. Jeuring, and L. T. van Binsbergen, "Ask-elle: an adaptable programming tutor for haskell giving automated feedback," *International Journal of Artificial Intelligence in Education*, vol. 27, no. 1, pp. 65–100, 2017.

[16] R. Zhi, S. Marwan, Y. Dong, N. Lytle, T. W. Price, and T. Barnes, "Toward data-driven example feedback for novice programming," in *EDM*, 2019.

[17] G. Singh, S. Srikant, and V. Aggarwal, "Question Independent Grading Using Machine Learning: The Case of Computer Program Grading," in *Proceedings of the 2016 SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[18] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: Data-driven feedback generation for introductory programming exercises," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, (New York, NY, USA), pp. 481–495, ACM, 2018.

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

[20] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas, "Learning program embeddings to propagate feedback on student code," *arXiv preprint arXiv:1505.05969*, 2015.

[21] A. Head, E. Glassman, G. Soares, R. Suzuki, L. Figueredo, L. D'Antoni, and B. Hartmann, "Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis," in *Proceedings of the 2017 Conference on Learning @ Scale*, ACM, 2017.

[22] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, (New York, NY, USA), p. 86–93, Association for Computing Machinery, 2010.

[23] V. J. Shute, "Focus on formative feedback," *Review of Educational Research*, vol. 78, no. 1, pp. 153–189, 2008.

[24] S. A. Ambrose, M. W. Bridges, M. DiPietro, M. C. Lovett, and M. K. Norman, *How Learning Works: Seven Research-based Principles for Smart Teaching.* Jossey-Bass, 2010.

[25] S. Cummins, A. Stead, L. Jardine-Wright, I. Davies, A. R. Beresford, and A. Rice, "Investigating the use of hints in online problem solving," in *Proceedings of the Third (2016) ACM Conference on Learning@ Scale*, pp. 105–108, 2016.

[26] J. M. Wing, "Computational thinking," *Commun. ACM*, vol. 49, pp. 33–35, Mar. 2006.

[27] Computing Research Association, "Generation CS: Computer Science Undergraduate Enrollments Surge Since 2006." `https://cra.org/data/Generation-CS/`.

[28] P. J. Guo, "Codeopticon: Real-time, one-to-many human tutoring for computer programming," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software &#38; Technology*, UIST '15, (New York, NY, USA), pp. 599–608, ACM, 2015.

[29] J. A. Kulik and C.-L. C. Kulik, "Timing of feedback and verbal learning," *Review of educational research*, vol. 58, no. 1, pp. 79–97, 1988.

[30] J. McBroom, I. Koprinska, and K. Yacef, "A survey of automated programming hint generation - the hints framework," *ArXiv*, vol. abs/1908.11566, 2019.

[31] A. Kyrilov and D. C. Noelle, "Binary instant feedback on programming exercises can reduce student engagement and promote cheating," in *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pp. 122–126, 2015.

[32] J. B. Moghadam, R. R. Choudhury, H. Yin, and A. Fox, "Autostyle: Toward coding style feedback at scale," in *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*, L@S '15, (New York, NY, USA), pp. 261–266, ACM, 2015.

[33] F. A. Fontana, E. Mariani, A. Mornioli, R. Sormani, and A. Tonello, "An experience report on using code smells detection tools," in *2011 IEEE fourth international conference on software testing, verification and validation workshops*, pp. 450–457, IEEE, 2011.

[34] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE software*, vol. 25, no. 5, pp. 22–29, 2008.

[35] C. W. Araújo, V. Zapalowski, and I. Nunes, "Using code quality features to predict bugs in procedural software systems," in *Proceedings of the XXXII Brazilian Symposium on Software Engineering*, pp. 122–131, 2018.

[36] S. H. Edwards, N. Kandru, and M. B. Rajagopal, "Investigating static analysis errors in student java programs," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, (New York, NY, USA), pp. 65–73, ACM, 2017.

[37] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," *SIGPLAN Not.*, vol. 48, no. 6, 2013.

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

[39] R. Suzuki, G. Soares, E. Glassman, A. Head, L. D'Antoni, and B. Hartmann, "Exploring the design space of automatically synthesized hints for introductory

programming assignments," in *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '17, (New York, NY, USA), p. 2951–2958, Association for Computing Machinery, 2017.

[40] K. Rivers and K. R. Koedinger, "Automating hint generation with solution space path construction," in *International Conference on Intelligent Tutoring Systems*, pp. 329–339, Springer, 2014.

[41] K. Rivers and K. R. Koedinger, "A canonicalizing model for building programming tutors," in *International Conference on Intelligent Tutoring Systems*, pp. 591–593, Springer, 2012.

[42] S. Gross, B. Mokbel, B. Hammer, and N. Pinkwart, "How to select an example? a comparison of selection strategies in example-based learning," in *International Conference on Intelligent Tutoring Systems*, pp. 340–347, Springer, 2014.

[43] D. Kim, Y. Kwon, P. Liu, I. L. Kim, D. M. Perry, X. Zhang, and G. Rodriguez-Rivera, "Apex: automatic programming assignment error explanation," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 311–327, 2016.

[44] A. Mitrovic and S. Ohlsson, "Evaluation of a constraint-based tutor for a database language," 1999.

[45] J. Holland, A. Mitrovic, and B. Martin, "J-LATTE: a Constraint-based Tutor for Java," in *The 2009 International Conference on Computers in Education*, 2009.

[46] A. Mitrovic, "Fifteen years of constraint-based tutors: what we have achieved and where we are going," *User modeling and user-adapted interaction*, vol. 22, no. 1, pp. 39–72, 2012.

[47] B. P. Woolf, *Building Intelligent Interactive Tutors: Student-centered Strategies for Revolutionizing e-Learning.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[48] A. T. Corbett and J. R. Anderson, "Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 245–252, 2001.

[49] J. Huang, C. Piech, A. Nguyen, and L. Guibas, "Syntactic and Functional Variability of a Million Code Submissions in a Machine Learning MOOC," in *Proceedings of the 2013 Workshop on Massive Open Online Courses at the 16th Annual Conference on Artificial Intelligence in Education*, 2013.

[50] N. Le, F. Loll, and N. Pinkwart, "Operationalizing the continuum between well-defined and ill-defined problems for educational technology," *IEEE Transactions on Learning Technologies*, vol. 6, no. 3, pp. 258–270, 2013.

[51] N.-T. Le and N. Pinkwart, "Towards a Classification for Programming Exercises," in *Proceedings of the 2nd Workshop on AI-supported Education for Computer Science*, 2014.

[52] J. T. Folsom-Kovarik, S. Schatz, and D. Nicholson, "Plan ahead: Pricing its learner models," in *Proceedings of the 19th Behavior Representation in Modeling & Simulation (BRIMS) Conference*, pp. 47–54, 2010.

[53] T. W. Price and T. Barnes, "An exploration of data-driven hint generation in an open-ended programming problem.," in *EDM (Workshops)*, Citeseer, 2015.

[54] R. H. Hwang, J. J. Wu, Z. Y. Tsai, P. T. Yu, and C.-F. Lai, "Enhancing the programming skill in high school engineering education via flipped classroom and peer assessment," in *43rd SEFI Annual Conference 2015, SEFI 2015*, European Society for Engineering Education (SEFI), 2015.

[55] Y. Wang, W. Ai, Y. Liang, and Y. Liu, "Toward motivating participants to assess peers' work more fairly: Taking programing language learning as an example," *Journal of Educational Computing Research*, vol. 52, no. 2, pp. 180–198, 2015.

[56] E. L. Glassman, A. Lin, C. J. Cai, and R. C. Miller, "Learnersourcing personalized hints," in *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, CSCW '16, (New York, NY, USA), pp. 1626–1636, ACM, 2016.

[57] A. Nguyen, C. Piech, J. Huang, and L. Guibas, "Codewebs: Scalable homework search for massive open online programming courses," in *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, (New York, NY, USA), pp. 491–502, ACM, 2014.

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

[59] E. L. Glassman, L. Fischer, J. Scott, and R. C. Miller, "Foobaz: Variable name feedback for student code at scale," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software &#38; Technology*, UIST '15, (New York, NY, USA), pp. 609–617, ACM, 2015.

[60] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: Suggesting solutions to error messages," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, (New York, NY, USA), pp. 1019–1028, ACM, 2010.

[61] S. H. Edwards, "Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance," in *Proceedings of the international conference on education and information systems: technologies and applications EISTA*, vol. 3, Citeseer, 2003.

[62] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 273–282, 2005.

[63] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: localizing errors in counterexample traces," in *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 97–105, 2003.

[64] A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error explanation with distance metrics," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 3, pp. 229–247, 2006.

[65] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *arXiv preprint arXiv:1709.06182*, 2017.

[66] D. Gopinath, M. Z. Malik, and S. Khurshid, "Specification-based program repair using sat," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 173–188, Springer, 2011.

[67] R. Könighofer and R. Bloem, "Automated error localization and correction for imperative programs," in *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 91–100, IEEE, 2011.

[68] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

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

[70] C. Le Goues, "Automatic program repair using genetic programming," *Univ. Virginia, Charlottesville, VA, USA*, 2013.

[71] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *2010 Third International Conference on Software Testing, Verification and Validation*, pp. 65–74, IEEE, 2010.

[72] J. Hattie and H. Timperley, "The power of feedback," *Review of educational research*, vol. 77, no. 1, pp. 81–112, 2007.

[73] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller, "Overcode: Visualizing variation in student solutions to programming problems at scale," *ACM Trans. Comput.-Hum. Interact.*, vol. 22, pp. 7:1–7:35, Mar. 2015.

[74] T. W. Price, Y. Dong, and D. Lipovac, "isnap: towards intelligent tutoring in novice programming environments," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on computer science education*, pp. 483–488, 2017.

[75] P. M. Phothilimthana and S. Sridhara, "High-coverage hint generation for massive courses: Do automated hints help cs1 students?," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, pp. 182–187, 2017.

[76] T. Barnes and J. Stamper, "Toward automatic hint generation for logic proof tutoring using historical student data," in *International conference on intelligent tutoring systems*, pp. 373–382, Springer, 2008.

[77] S. H. Edwards, "Using software testing to move students from trial-and-error to reflection-in-action," *ACM SIGCSE Bulletin*, vol. 36, no. 1, pp. 26–30, 2004.

[78] L. Gusukuma, A. C. Bart, D. Kafura, and J. Ernst, "Misconception-driven feedback: Results from an experimental study," in *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pp. 160–168, 2018.

[79] S. Marwan, J. Jay Williams, and T. Price, "An evaluation of the impact of automated programming hints on performance and learning," in *Proceedings of the 2019 ACM Conference on International Computing Education Research*, pp. 61–70, ACM, 2019.

[80] M. Eagle and T. Barnes, "Evaluation of automatically generated hint feedback," in *Educational Data Mining 2013*, 2013.

[81] D. Lavbič, T. Matek, and A. Zrnec, "Recommender system for learning sql using hints," *Interactive Learning Environments*, vol. 25, no. 8, pp. 1048–1064, 2017.

[82] R. S. Baker, A. T. Corbett, K. R. Koedinger, and A. Z. Wagner, "Off-task behavior in the cognitive tutor classroom: when students game the system," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 383–390, 2004.

[83] B. Wisniewski, K. Zierer, and J. Hattie, "The power of feedback revisited: a meta-analysis of educational feedback research," *Frontiers in Psychology*, vol. 10, p. 3087, 2020.

[84] K. R. Koedinger, A. T. Corbett, and C. Perfetti, "The knowledge-learning-instruction framework: Bridging the science-practice chasm to enhance robust student learning," *Cognitive science*, vol. 36, no. 5, pp. 757–798, 2012.

[85] C. F. Lynch, K. D. Ashley, V. Aleven, and N. Pinkwart, "Defining ill-defined domains; a literature survey," in *Intelligent Tutoring Systems (ITS 2006): Workshop on Intelligent Tutoring Systems for Ill-Defined Domains*, 2006.

[86] R. E. Mayer, J. L. Dyck, and W. Vilberg, "Learning to program and learning to think: What's the connection?," *Commun. ACM*, vol. 29, pp. 605–610, July 1986.

[87] K. Brennan and M. Resnick, "New frameworks for studying and assessing the development of computational thinking," in *Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada*, pp. 1–25, 2012.

[88] R. Caceffo, S. Wolfman, K. S. Booth, and R. Azevedo, "Developing a computer science concept inventory for introductory programming," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pp. 364–369, 2016.

[89] S. A. Fincher and A. V. Robins, *The Cambridge handbook of computing education research.* Cambridge University Press, 2019.

[90] H. Hubball and N. Gold, "The scholarship of curriculum practice and undergraduate program reform: Integrating theory into practice," *New directions for teaching and learning*, vol. 2007, no. 112, pp. 5–14, 2007.

[91] J. A. Boyle, G. Haldeman, A. Tjang, M. Babes-Vroman, A. P. Centeno, and T. D. Nguyen, “Dynamic recitation: A student-focused, goal-oriented recitation management platform,” in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pp. 1269–1269, 2019.

[92] F. D. Davis, “Perceived usefulness, perceived ease of use, and user acceptance of information technology,” *MIS Quarterly*, vol. 13, no. 3, pp. 319–340, 1989.

[93] G. Haldeman, M. Babeş-Vroman, A. Tjang, and T. D. Nguyen, “Csf: Formative feedback in autograding,” *ACM Transactions on Computing Education (TOCE)*, vol. 21, no. 3, pp. 1–30, 2021.

[94] T. Jones, S. Forrest, *et al.*, “Fitness distance correlation as a measure of problem difficulty for genetic algorithms.,” in *ICGA*, vol. 95, pp. 184–192, 1995.

[95] E. Soloway and J. C. Spohrer, *Studying the novice programmer.* Psychology Press, 2013.

[96] “Javaparser.org.” `https://github.com/javaparser`. Accessed: 2021-06-17.

[97] “Junit.org.” `https://junit.org`. Accessed: 2021-06-27.

[98] N. Giacaman and G. De Ruvo, “Bridging theory and practice in programming lectures with active classroom programmer,” *IEEE Transactions on Education*, vol. 61, no. 3, pp. 177–186, 2018.

Appendix A

Complete Description of Select Assignments

PayFriend and *TwoSmallest* are two assignments I use repeatedly in my studies. Thus, in this section I include their descriptions as it was provided to the students.

Pay Friend

Write your code in the file PayFriend.java, your file has to have this exact name with P and F capitalized. You must use the IO module to read the input and to output your answer. Imagine that you work for a payment processing service called PayFriend. PayFriend charges money receivers the following fees:

The first \$100 has a flat fee of \$5. Payments over \$100 (but under \$1000) have a fee of 3% or \$6, whichever is higher. Payments of \$1,000 (but under \$10,000) and over have a fee of 1% or \$15, whichever is higher. Payments of \$10,000 and over are subject to (fees as follows): The first \$10,000 have a fee of 1% The next \$5,000 have an additional fee of 2% Anything more will demand an additional fee of 3% For example, an payment of \$40,000 would be subject to \$950 fee: 1% on the first \$10,000 (\$100 fee), 2% on the next \$5,000 (\$100 fee), and 3% on the last \$25,000 (\$750 tax).

Write a program that asks the user for the payment amount (real number) and outputs the fee owned (real number).

Example: java PayFriend 450.0

RESULT: 13.5

Two Smallest

Write your code in the file TwoSmallest.java, your file has to have this exact name with T and S capitalized. You must use the IO module to read inputs and to output your answers. Write a program that takes a set of numbers and determines which are the two smallest numbers. Ask the user for the following information, in this order: A terminating value (real number). The user will enter this value again to indicate that he or she is finished providing input. A sequence of real numbers. Keep asking for numbers until the terminating value is entered. Compute and output the smallest and second-smallest real number, in that order. It is possible for the smallest and second-smallest numbers to be the same (if the sequence contains duplicate numbers). There must be at least 2 (two) numbers in the list of numbers that is not the terminating value. If the user enters less than 2 (two) numbers, consider an error. Report the error input via IO.reportBadInput() and RE-ASK the user for the input until it is correctly entered.

Example: java TwoSmallest 123 [this is the terminating value, not part of the set of numbers] 17.0 23.5 10.0 15.2 30.0 8.0 16.0 123 [this is the terminating value again, indicating that the user is done]

RESULTS TO OUTPUT (in this order): 8.0 10.0

Appendix B

Complete Statics for the Hints Efficiency Study

I previously discuss the most relevant results regarding the movement of student submissions throughout the different classes of errors in Section 7.2.1. For completion, in this section I include the percentages of all the possible pairs of submissions for each of the two assignments, and for each of the three semesters. In each table, the row label indicates the start error class and the column label indicates the end error class for a pair of code submissions. We calculate each percentage by counting the number of pairs and then diving it by the total number of pairs which started in the same error class.

Table B.1: Movement between classes of error from one submission to another for PayFriend in spring 2016; CM - does not compile, IN - failed to follow instructions, DR - failed data representation, CF - failed control flow, FM - failed translating to formulas, CN - failed translating to conditional statements, NM - both CN and FM, NO - no concept failed/ passed all tests

	CM	**IN**	**DR**	**CF**	**CN**	**FM**	**NM**	**NO**
CM	**33.3%**	4.2%	8.3%	20.8%	8.3%	4.2%	4.2%	16.7%
IN	4.4%	**36.8%**	9.6%	10.5%	7.9%	2.6%	12.3%	**14.9%**
DR	0.8%	2.1%	**57.3%**	3.4%	2.9%	1.7%	9.2%	21.8%
CF	1.5%	3.1%	0.0%	**44.6%**	13.9%	4.6%	6.2%	24.6%
CN	0.0%	3.5%	6.9%	6.9%	**37.9%**	0.0%	6.9%	**34.5%**
FM	4.4%	0.0%	0.0%	2.2%	0.0%	**28.9%**	0.0%	2.2%
NM	0.9%	1.7%	2.7%	1.8%	5.3%	5.3%	**46.9%**	**35.4%**

Table B.2: Movement between classes of error from one submission to another for PayFriend in spring 2017

	CM	**IN**	**DR**	**CF**	**CN**	**FM**	**NM**	**NO**
CM	**45.6%**	0.0%	5.1%	0.0%	10.1%	7.6%	13.9%	17.7%
IN	3.7%	**7.4%**	3.7%	3.7%	0.0%	33.3%	29.6%	**11.1%**
DR	3.8%	0.0%	**59.0%**	0.0%	10.3%	1.3%	3.8%	21.8%
CF	25.0%	0.0%	0.0%	**75.0%**	0.0%	0.0%	0.0%	0.0%
CN	9.1%	0.0%	0.0%	9.1%	**18.2%**	9.1%	18.2%	**18.2%**
FM	0.0%	0.0%	0.0%	0.0%	0.0%	**33.3%**	0.0%	66.7%
NM	1.7%	1.7%	0.0%	0.0%	3.4%	5.2%	**65.5%**	**22.4%**

Table B.3: Movement between classes of error from one submission to another for PayFriend in spring 2018

	CM	**IN**	**DR**	**CF**	**CN**	**FM**	**NM**	**NO**
CM	**33.7%**	2.2%	4.3%	8.7%	7.6%	2.2%	18.5%	21.7%
IN	14.6%	**7.3%**	0.0%	7.3%	9.8%	2.4%	26.8%	**31.7%**
DR	5.3%	0.0%	**28.9%**	5.3%	18.4%	2.6%	15.8%	21.1%
CF	2.4%	0.0%	2.4%	**38.1%**	9.5%	4.8%	16.7%	26.2%
CN	1.7%	0.0%	1.7%	5.0%	**33.3%**	1.7%	5.0%	**51.7%**
FM	0.0%	0.0%	0.0%	0.0%	2.9%	**20.6%**	20.6%	55.9%
NM	0.0%	0.0%	0.0%	3.7%	12.1%	5.6%	**29.0%**	**49.5%**

Table B.4: Movement between classes of error from one submission to another for TwoSmallest in spring 2016; CM - does not compile, IS - failed to follow instructions, IL - infinite loop, SQ - failed to read sequence, IN - failed to initialize min variables, UP - failed to update min variables, NO - no concept failed/ passed all tests

	CM	**IS**	**IL**	**IN**	**UP**	**SQ**	**SI**	**SU**	**NO**
CM	**0.0%**	33.3%	13.3%	6.7%	0.0%	6.7%	0.0%	6.7%	33.3%
IS	2.4%	**39.1%**	17.8%	1.6%	5.1%	7.5%	12.7%	1.6%	12.3%
IL	0.0%	11.3%	**58.6%**	2.7%	4.1%	2.7%	6.8%	1.8%	12.2%
IN	0.0%	5.3%	5.3%	**26.3%**	21.1%	10.5%	0.0%	0.0%	31.6%
UP	0.0%	7.3%	19.5%	0.0%	**29.3%**	0.0%	0.0%	0.0%	43.9%
SQ	1.3%	14.1%	18.0%	1.3%	0.0%	**21.8%**	6.4%	0.0%	**37.2%**
SI	0.6%	10.7%	14.1%	2.8%	6.2%	9.0%	**33.9%**	4.5%	18.1%
SU	4.2%	8.3%	12.5%	0.0%	16.7%	0.0%	20.8%	**16.7%**	20.8%

Table B.5: Movement between classes of error from one submission to another for TwoSmallest in spring 2017

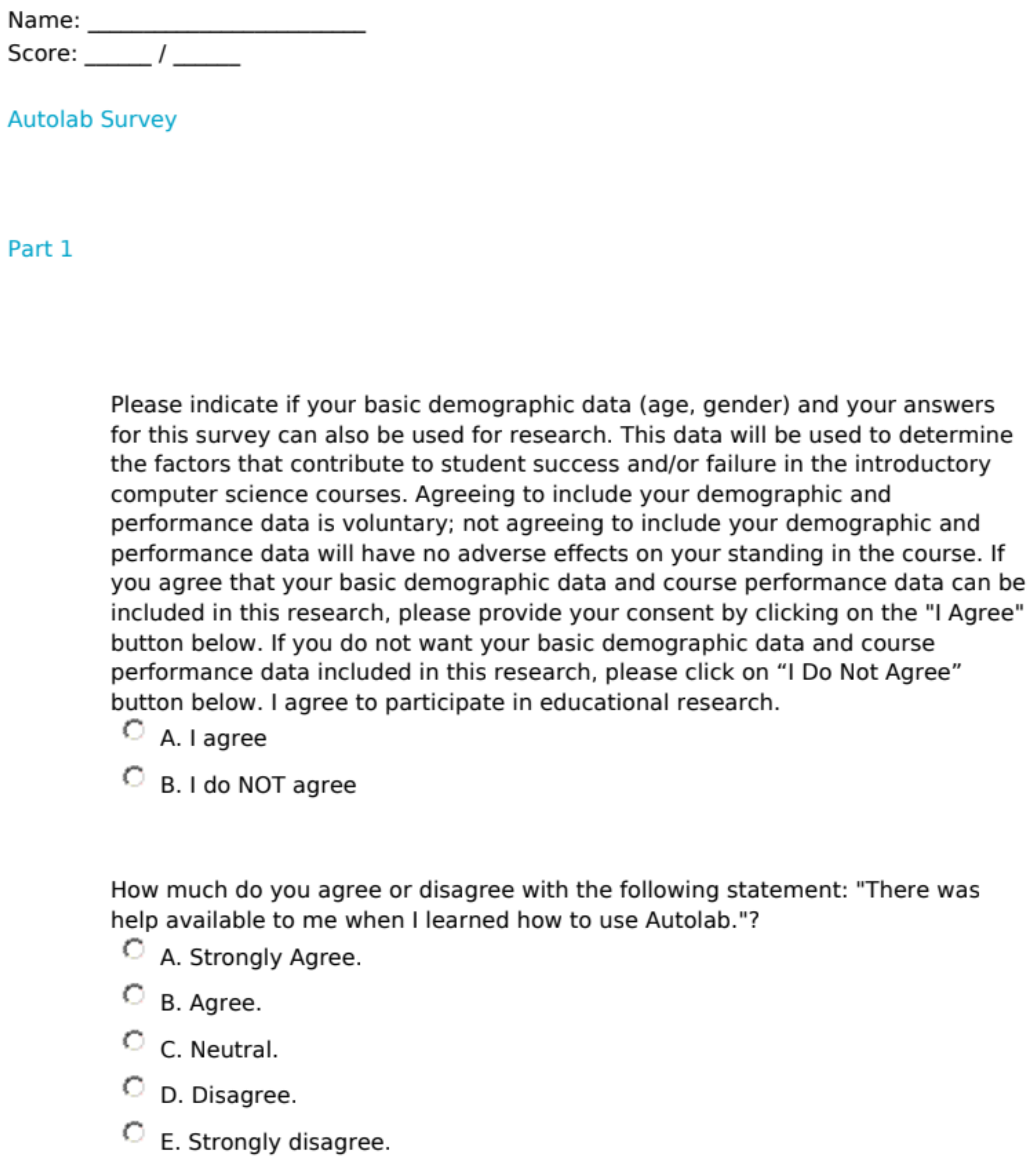
	CM	**IS**	**IL**	**IN**	**UP**	**SQ**	**SI**	**SU**	**NO**
CM	**20.7%**	0.0%	0.0%	3.4%	20.7%	13.8%	8.6%	1.7%	31.0%
IS	10.7%	**25.0%**	3.6%	0.0%	7.1%	28.6%	3.6%	3.6%	17.9%
IL	6.3%	0.0%	**25.0%**	6.3%	6.3%	31.3%	12.5%	0.0%	12.5%
IN	3.7%	0.0%	3.7%	**44.4%**	0.0%	3.7%	3.7%	3.7%	37.0%
UP	0.0%	0.0%	0.0%	8.7%	**39.1%**	4.3%	0.0%	0.0%	47.8%
SQ	6.1%	1.0%	3.0%	0.0%	4.0%	**57.6%**	14.1%	1.0%	**13.1%**
SI	2.8%	0.7%	0.7%	6.9%	5.5%	17.2%	**42.1%**	5.5%	18.6%
SU	2.7%	0.0%	0.0%	0.0%	16.2%	18.9%	10.8%	**40.5%**	10.8%

Table B.6: Movement between classes of error from one submission to another for TwoSmallest in spring 2018

	CM	IS	IL	IN	UP	SQ	SI	SU	NO
CM	**28.6%**	3.9%	3.9%	9.1%	6.5%	14.3%	9.1%	3.9%	20.8%
IS	8.6%	**14.3%**	5.7%	17.1%	5.7%	20.0%	17.1%	0.0%	11.4%
IL	20.0%	8.0%	**20.0%**	4.0%	4.0%	24.0%	8.0%	4.0%	8.0%
IN	4.0%	4.0%	0.0%	**18.0%**	16.0%	8.0%	12.0%	2.0%	36.0%
UP	2.7%	1.4%	0.0%	2.7%	**41.1%**	1.4%	2.7%	2.7%	45.2%
SQ	4.0%	0.6%	0.0%	4.0%	6.3%	**52.6%**	7.4%	1.7%	**23.4%**
SI	6.2%	1.8%	0.9%	8.8%	12.4%	13.3%	**32.7%**	5.3%	18.6%
SU	2.9%	0.0%	2.9%	5.7%	40.0%	22.9%	0.0%	**20.0%**	5.70%

Appendix C

Copy of the TAM Survey



Name: ________________________
Score: ______ / ______

Autolab Survey

Part 1

Please indicate if your basic demographic data (age, gender) and your answers for this survey can also be used for research. This data will be used to determine the factors that contribute to student success and/or failure in the introductory computer science courses. Agreeing to include your demographic and performance data is voluntary; not agreeing to include your demographic and performance data will have no adverse effects on your standing in the course. If you agree that your basic demographic data and course performance data can be included in this research, please provide your consent by clicking on the "I Agree" button below. If you do not want your basic demographic data and course performance data included in this research, please click on “I Do Not Agree” button below. I agree to participate in educational research.

○ A. I agree

○ B. I do NOT agree

How much do you agree or disagree with the following statement: "There was help available to me when I learned how to use Autolab."?

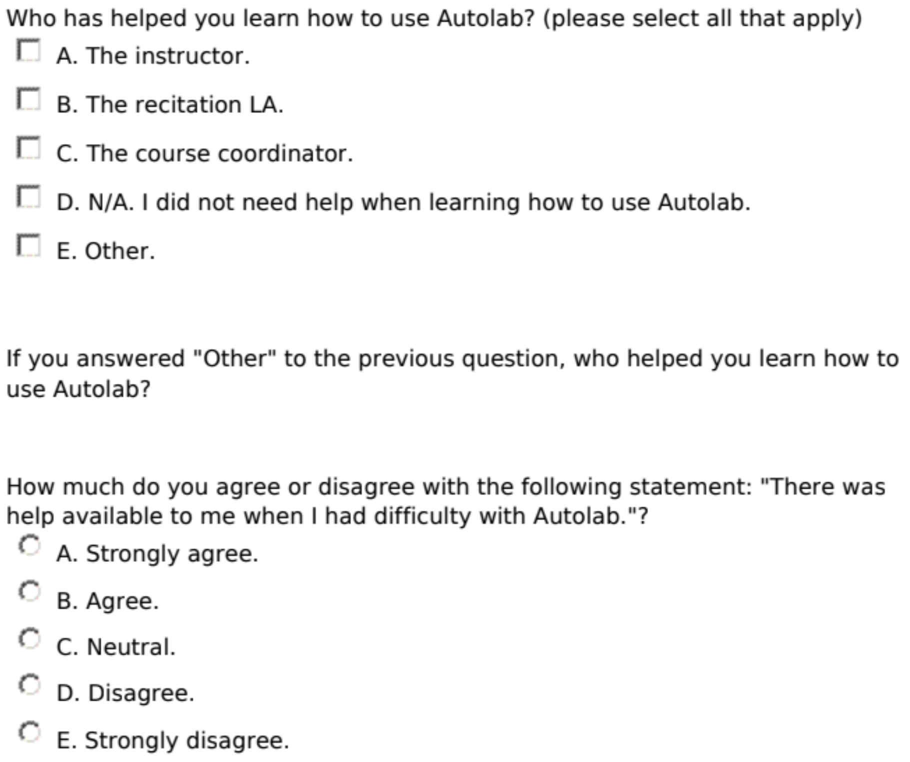
○ A. Strongly Agree.

○ B. Agree.

○ C. Neutral.

○ D. Disagree.

○ E. Strongly disagree.



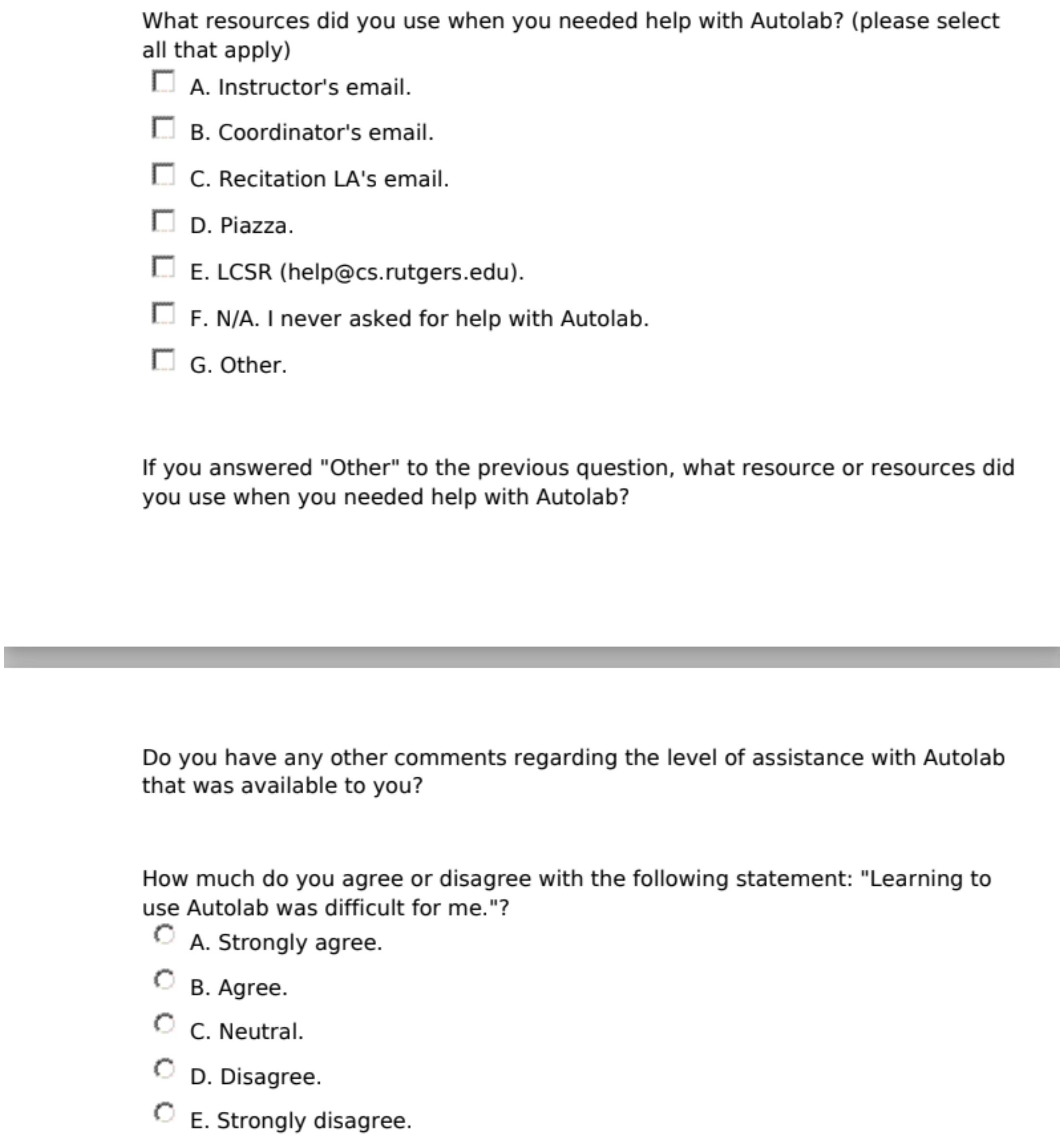
Who has helped you learn how to use Autolab? (please select all that apply)

- ☐ A. The instructor.
- ☐ B. The recitation LA.
- ☐ C. The course coordinator.
- ☐ D. N/A. I did not need help when learning how to use Autolab.
- ☐ E. Other.

If you answered "Other" to the previous question, who helped you learn how to use Autolab?

How much do you agree or disagree with the following statement: "There was help available to me when I had difficulty with Autolab."?

- ○ A. Strongly agree.
- ○ B. Agree.
- ○ C. Neutral.
- ○ D. Disagree.
- ○ E. Strongly disagree.



What resources did you use when you needed help with Autolab? (please select all that apply)

☐ A. Instructor's email.

☐ B. Coordinator's email.

☐ C. Recitation LA's email.

☐ D. Piazza.

☐ E. LCSR (help@cs.rutgers.edu).

☐ F. N/A. I never asked for help with Autolab.

☐ G. Other.

If you answered "Other" to the previous question, what resource or resources did you use when you needed help with Autolab?

Do you have any other comments regarding the level of assistance with Autolab that was available to you?

How much do you agree or disagree with the following statement: "Learning to use Autolab was difficult for me."?

○ A. Strongly agree.

○ B. Agree.

○ C. Neutral.

○ D. Disagree.

○ E. Strongly disagree.

How much do you agree or disagree with the following statement: "It was easy to submit my programming assignments using Autolab."?

○ A. Strongly agree.

○ B. Agree.

○ C. Neutral.

○ D. Disagree.

○ E. Strongly disagree.

How much do you agree or disagree with the following statement: "The interaction with Autolab was confusing."?

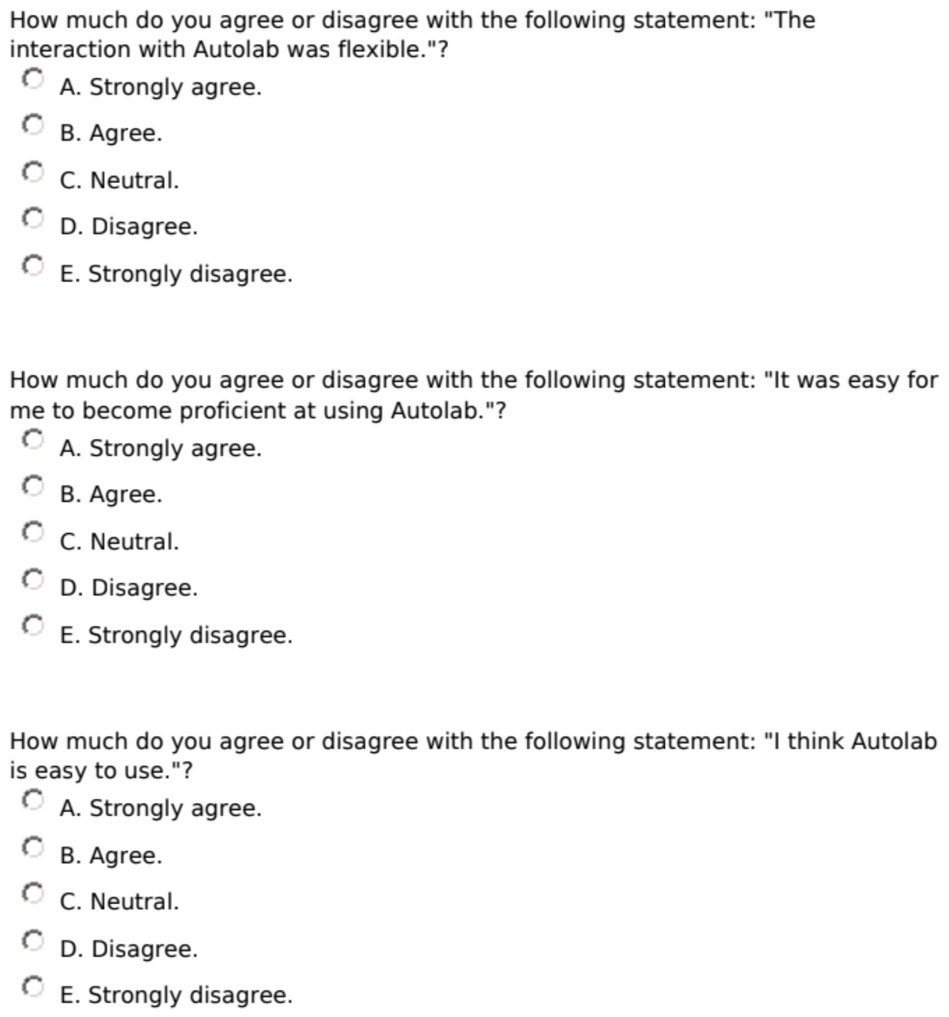
○ A. Strongly agree.

○ B. Agree.

○ C. Neutral.

○ D. Disagree.

○ E. Strongly disagree.



How much do you agree or disagree with the following statement: "The interaction with Autolab was flexible."?

○ A. Strongly agree.

○ B. Agree.

○ C. Neutral.

○ D. Disagree.

○ E. Strongly disagree.

How much do you agree or disagree with the following statement: "It was easy for me to become proficient at using Autolab."?

○ A. Strongly agree.

○ B. Agree.

○ C. Neutral.

○ D. Disagree.

○ E. Strongly disagree.

How much do you agree or disagree with the following statement: "I think Autolab is easy to use."?

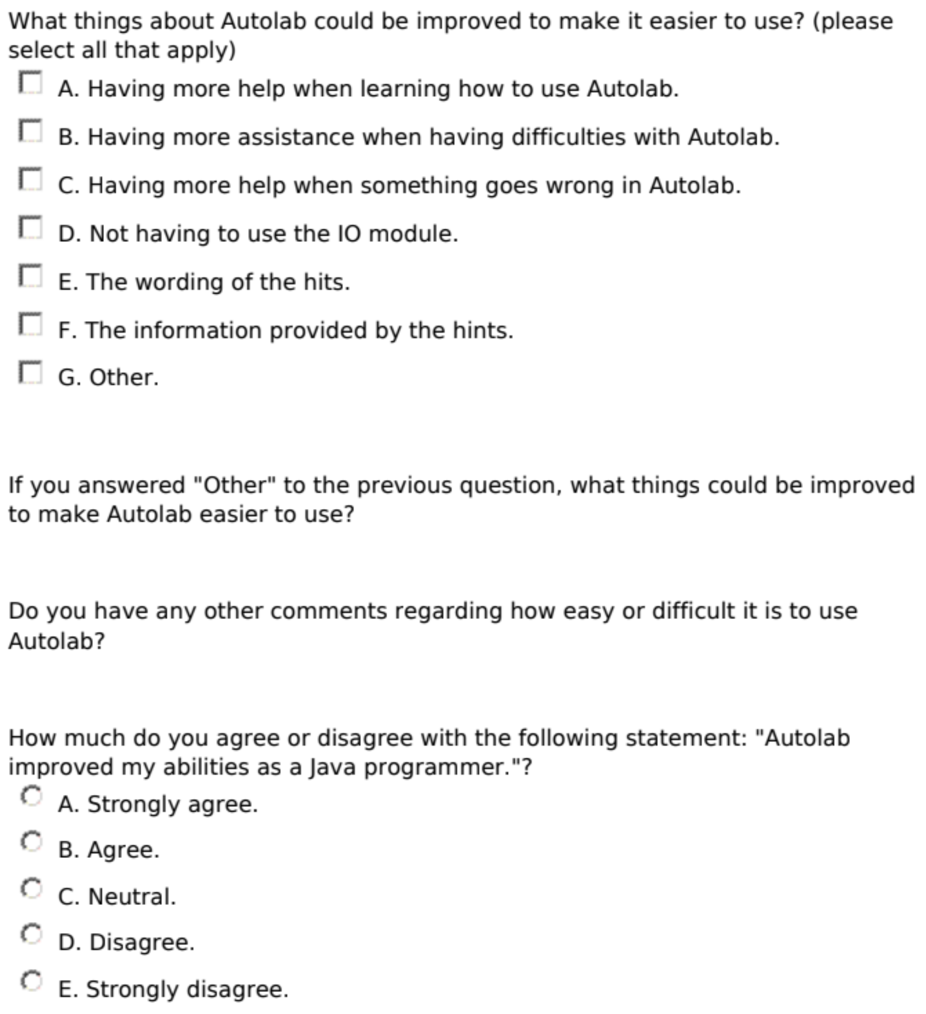
○ A. Strongly agree.

○ B. Agree.

○ C. Neutral.

○ D. Disagree.

○ E. Strongly disagree.



What things about Autolab could be improved to make it easier to use? (please select all that apply)

☐ A. Having more help when learning how to use Autolab.

☐ B. Having more assistance when having difficulties with Autolab.

☐ C. Having more help when something goes wrong in Autolab.

☐ D. Not having to use the IO module.

☐ E. The wording of the hits.

☐ F. The information provided by the hints.

☐ G. Other.

If you answered "Other" to the previous question, what things could be improved to make Autolab easier to use?

Do you have any other comments regarding how easy or difficult it is to use Autolab?

How much do you agree or disagree with the following statement: "Autolab improved my abilities as a Java programmer."?

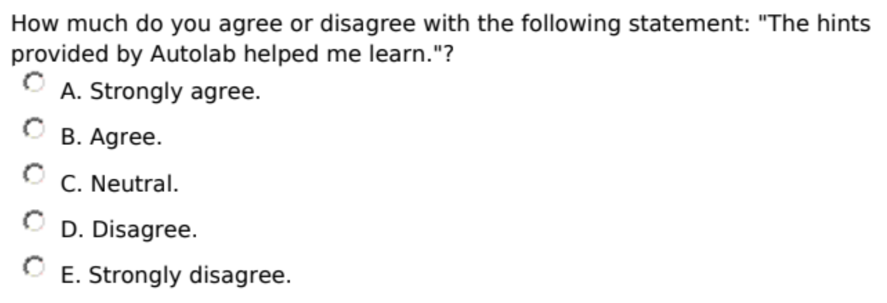
○ A. Strongly agree.

○ B. Agree.

○ C. Neutral.

○ D. Disagree.

○ E. Strongly disagree.



How much do you agree or disagree with the following statement: "The hints provided by Autolab helped me learn."?

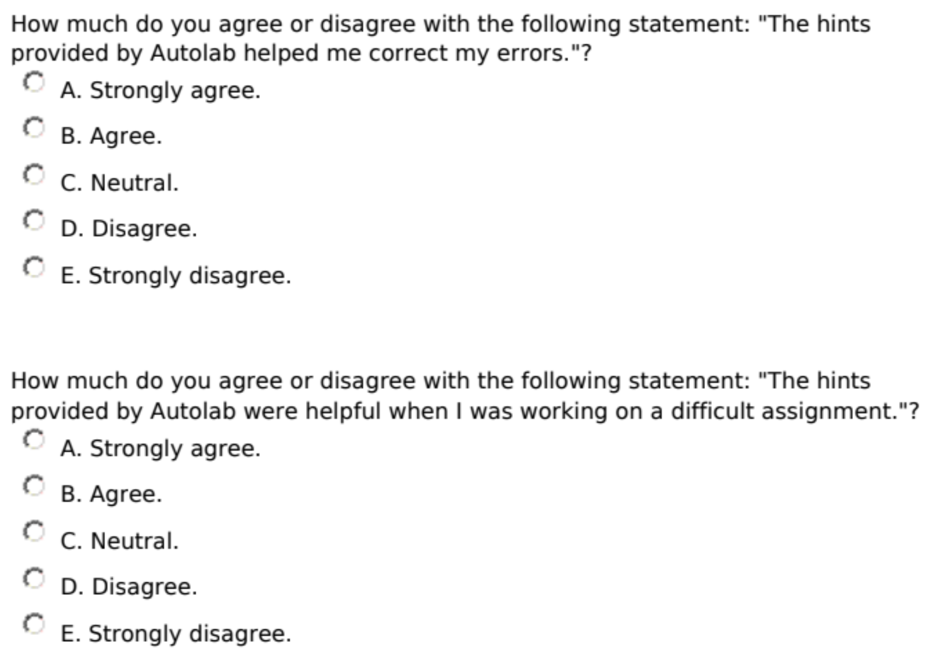
○ A. Strongly agree.

○ B. Agree.

○ C. Neutral.

○ D. Disagree.

○ E. Strongly disagree.



How much do you agree or disagree with the following statement: "The hints provided by Autolab helped me correct my errors."?

○ A. Strongly agree.

○ B. Agree.

○ C. Neutral.

○ D. Disagree.

○ E. Strongly disagree.

How much do you agree or disagree with the following statement: "The hints provided by Autolab were helpful when I was working on a difficult assignment."?

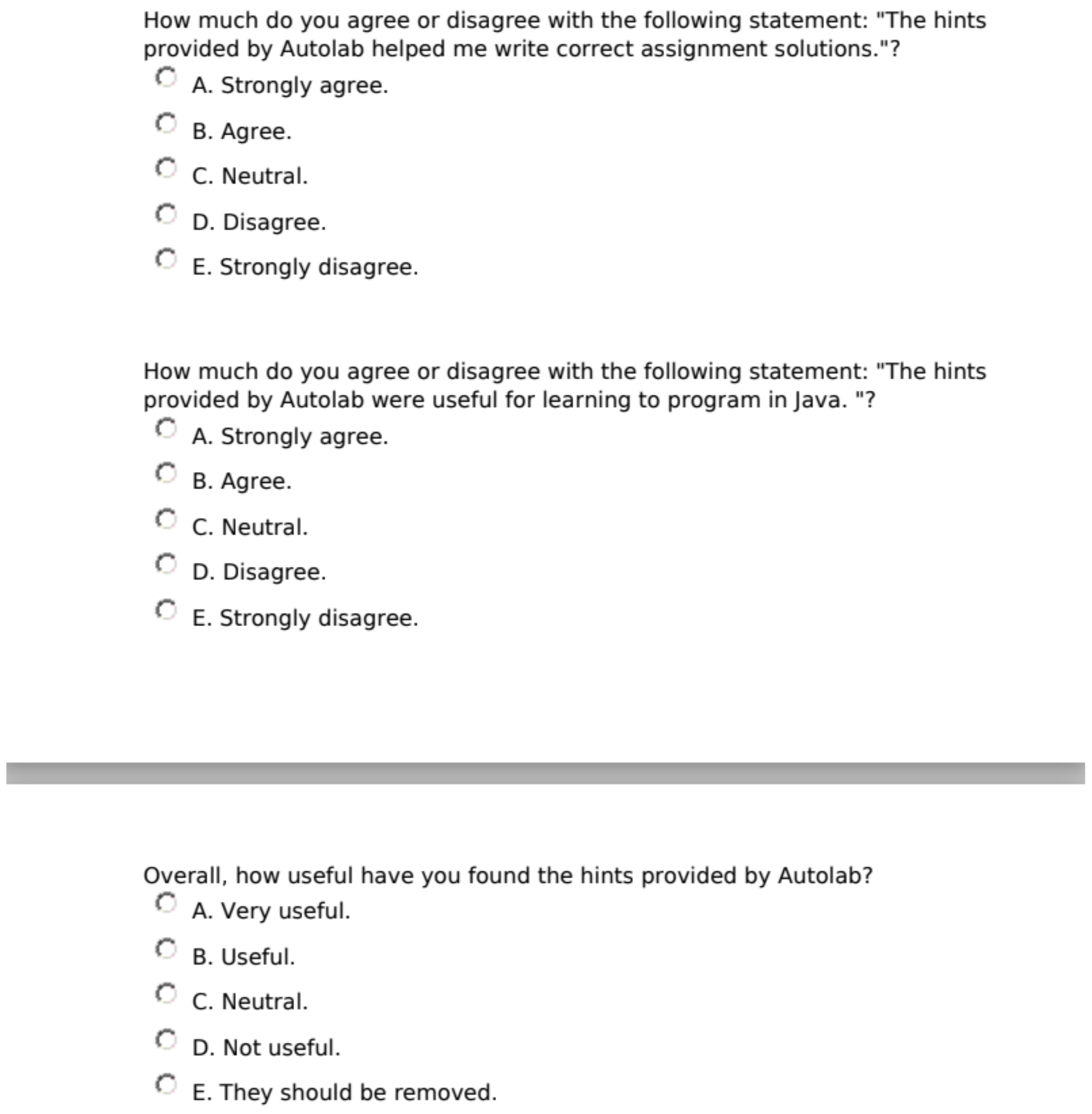
○ A. Strongly agree.

○ B. Agree.

○ C. Neutral.

○ D. Disagree.

○ E. Strongly disagree.



How much do you agree or disagree with the following statement: "The hints provided by Autolab helped me write correct assignment solutions."?

- A. Strongly agree.
- B. Agree.
- C. Neutral.
- D. Disagree.
- E. Strongly disagree.

How much do you agree or disagree with the following statement: "The hints provided by Autolab were useful for learning to program in Java. "?

- A. Strongly agree.
- B. Agree.
- C. Neutral.
- D. Disagree.
- E. Strongly disagree.

Overall, how useful have you found the hints provided by Autolab?

- A. Very useful.
- B. Useful.
- C. Neutral.
- D. Not useful.
- E. They should be removed.