RICE UNIVERSITY

Data-Driven Tools for Introductory Computer Science Education

by

Lei Tang

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Joe D. Warren, Chair
Professor of Computer Science

Scott Rixner
Professor of Computer Science and Electrical and Computer Engineering

Margaret E. Beier
Associate Professor of Psychology

Houston, Texas

August, 2018
Data-Driven Tools for Introductory Computer Science Education
by
Lei Tang

Abstract

The software industry spends a tremendous amount of effort and resources on software testing and maintenance to improve the quality of software. However, a large portion of the cost may be saved by training high-quality software developers with better Computer Science education. Skilled software developers can not only produce code of fewer bugs and better design but also identify and fix issues more effectively. Therefore, in this thesis, we researched building useful educational tools for facilitating Computer Science education, particularly in introductory programming courses.

Since understanding the code execution is the first step of writing high-quality code and software testing, in the first study, we built a web-based interactive tool to teach students necessary comprehension and analysis skills to understand the program execution. Secondly, we built an automated tool for students to interactively practice writing test cases and debugging programs. The tool gauges the test coverage of students’ test sets using a large corpus of buggy programs we collected in our previous course sessions. The tool returns the buggy programs as immediate feedback which students’ test sets failed to catch. Students need to study those returned buggy programs to gradually improve the testing coverage of their test sets. In the third project, we built a tool that automatically generates high-quality test cases to construct concise test sets for testing students’ coding assignment solutions. The tool utilizes heterogeneous historical student incorrect implementations to guide the test case search process. Its generated test cases are expected to provide better test
coverage than instructor built tests cases.

To validate the effectiveness of our tools, we conducted studies in introductory programming courses among students at Rice and online students of our Massive Open Online Courses (MOOC). The studies showed that, compared with studying traditional Computer Science curriculum, students made significant improvements in understanding basic Computer Science concepts and software testing by interacting with our educational tools.
Acknowledgments

This thesis would not have been possible without the help and support of many people during my PhD studies.

First and foremost, I express my sincere gratitude to my advisor Professor Joe Warren for his steadfast support, guidance, and encouragement. I learned a lot from Joe’s sharp observations, intellectual curiosity, and passion for research problems, which motivate me not only in graduate studies but also in my future life. He has always been my support and inspiration whenever I met with difficulties. I am truly grateful to Joe. I also would like to express my special thanks to Professor Scott Rixner for his technical advices, inspirations, and help with my research work. Scott has set an excellent example for me with his meticulous and rigorous scholarship. His constant constructive criticism improved my research and this dissertation significantly. I am also grateful to Professor Margaret E. Beier for generously agreeing to serve as part of my committee and imparting helpful suggestions on both of my dissertation and research.

Thank Professor Moshe Vardi for his inspirations and encouragement during my PhD studies. I also thank Professor Anshumali Shrivastava for his discussion with me on applying Machine Learning in Computer Science education. My special thanks go to Dr. Jan Hewitt, Dr. Tracy Volz, and Joe for their help on building my technical writing and presentation skills which I can benefit life-long. I also want to thank Professor Ron Goldman and Dr. Stephen Wong for their friendship, valuable discussions, and their courses at Rice.

I would like to also thank my close collaborator Rebecca Smith for her hard work and friendship. I am truly impressed by her excellence in research. I appreciate Yue Wang and Binghang Yuan for sharing the office with me and being good accompanies. Thank all my fellow students and friends inside and outside Rice for their friendship, encouragement, and the time we spent together, which I will treasure forever.

I would like to thank my father Ming Tang, my mother Lianxiang Zhai, and my
sister Wenting Tang, for their great love and support throughout the years. And I could not have finished this thesis and study without them. I owe them a lot. I thank my brothers and sisters in Christ for their constant prayers and support. Last but not least, I thank my Lord Jesus Christ almighty God for guiding me towards PhD and leading me on this journey with His abundant love and grace. May all the glory be to Him alone.
Contents

Abstract .......................................................... ii
Acknowledgments ................................................. iv
List of Illustrations ........................................... ix

1 Introduction ..................................................... 1
   1.1 Motivation .................................................. 1
   1.2 Research Overview ....................................... 4
       1.2.1 Improving Students’ Understanding of Code Execution ..... 5
       1.2.2 Improving Students’ Test Case Generation Skills .......... 7
       1.2.3 Generating Test Cases for Grading Students’ Programs .... 8
   1.3 Contributions ............................................. 10
   1.4 Outline of Thesis ......................................... 11

2 Related Work .................................................. 13
   2.1 Novice Programmer Education ............................... 13
   2.2 Teaching Software Testing ................................. 14
   2.3 Educational Tools in Computer Science Classes ............ 18
   2.4 Data-Driven Teaching ...................................... 22

3 Improving Students’ Understanding of Code Execution 26
   3.1 Related Works and Background ............................. 26
   3.2 A Tool for Teaching How Programs Are Executed .......... 33
       3.2.1 Student View ........................................... 35
       3.2.2 Instructor View ....................................... 40
3.2.3 Architecture .............................................. 46
3.3 Study Using iTeach .............................................. 48
  3.3.1 Study Method .............................................. 48
  3.3.2 Scoring Method .......................................... 54
  3.3.3 Two Study Groups ....................................... 54
3.4 Study Results .................................................. 56
  3.4.1 Understanding of Code Execution ......................... 56
  3.4.2 Confidence to Complete Course .......................... 61
  3.4.3 Assignment Evaluation ................................... 63
  3.4.4 Summary .................................................. 64
3.5 Conclusion ...................................................... 67

4 Improving Students’ Test Cases Generation Skills .... 69
  4.1 Background and Related Works ............................. 70
  4.2 Tool Design Overview ...................................... 74
  4.3 Infrastructure ............................................... 76
    4.3.1 Solution Corpus Constructor ............................ 76
    4.3.2 Bug Identifier .......................................... 77
    4.3.3 Progression Scheduler ................................... 77
    4.3.4 Student Interface ...................................... 78
  4.4 Methodology .................................................. 79
  4.5 Application and Evaluation .................................. 82
  4.6 Conclusion .................................................. 86

5 Generating Test Cases for Grading Students’ Programs ... 88
  5.1 Background and Related Works ............................. 89
  5.2 Tool Design Overview ...................................... 92
  5.3 Base Test Set Generation ................................... 95
    5.3.1 Specifying the Domain for Test Cases .................. 95
5.3.2 Auto-Generating Test Cases ........................................... 96
5.3.3 Assigning Test Case Complexity .................................... 97
5.3.4 Configuration Example .................................................. 97
5.4 Testing Student Solutions .................................................. 100
5.5 Concise Test Set Generation .............................................. 102
  5.5.1 Approximately Minimal Test Sets .................................... 102
  5.5.2 Gradated Complexity Test Sets ...................................... 103
5.6 Application and Evaluation .............................................. 106
  5.6.1 Comparison with Expert Tests ...................................... 106
  5.6.2 Method Sensitivity Analysis ........................................ 108
5.7 Conclusion and Future Work ............................................ 110

6 Conclusion and Future Work ............................................. 112
  6.1 Teaching Program Execution .......................................... 114
  6.2 Teaching Test Case Generation ........................................ 114
  6.3 Building a Test Case Generator ....................................... 115
  6.4 Summary ..................................................................... 115
  6.5 Future Work .................................................................. 116

Bibliography ................................................................. 117
Illustrations

1.1 Software Development Life Cycle [1] (SDLC) ........................................ 2

2.1 Illustration of Typical Software Testing Process Using Test Cases . . 17

3.1 jGRASP and UUhistle Code Execution Visualization Tools .......... 30
3.2 Online Python Code Execution Visualizer: Online Python Tutor and  
    CodeSkulptor VizMode .......................................................... 31
3.3 iTeach Tool Quiz Page .............................................................. 36
3.4 iTeach Tool Quiz Feedback Pages .............................................. 38
3.5 iTeach Tool Problem Construction Process ................................. 39
3.6 iTeach Tool Instructor Administration Dashboard .......................... 40
3.7 iTeach Tool Problem Management Panel .................................... 41
3.8 Overview of iTeach tool Quiz and Problem Structure .................... 41
3.9 Problem Authoring Page ............................................................ 42
3.10 iTeach Tool Problem Sample Choice Panel ................................. 43
3.11 Synthesis a Quiz From Existing Problems ................................. 44
3.12 iTeach tool Quiz Admin Page ..................................................... 44
3.13 Automated Mutant Program Generation ..................................... 45
3.14 System Architecture of iTeach tool ........................................... 47
3.15 iTeach Tool Study Quiz 1 ......................................................... 49
3.16 iTeach Tool Study Quiz 2 ......................................................... 50
3.17 iTeach Tool Study Quiz 3 ......................................................... 51
3.18 Distribution of Student Study Years ........................................ 54
3.19 Distribution of Student Computer Science Background ............. 55
3.20 Distribution of students’ understanding of code execution BEFORE the study. Experimental group average score: 0.59; Control group average score: 0.875 ........................................ 57
3.21 Distribution of students’ understanding of code execution AFTER study. Experimental group average score: 1.93; Control group average score: 1.725 ........................................ 58
3.22 Distribution of students’ understanding of code execution after study among students initially with “no understanding” at all. ........... 59
3.23 Distribution of students’ understanding of code execution after study among students initially with “a little understanding”.......... 59
3.24 Distribution of students’ confidence level on completing the course before and after study in Experimental group. Average score before: 1.556; after: 1.926 ........................................ 61
3.25 Distribution of students’ confidence level on completing the course before and after study in Control group. Average score before: 2.025; after: 2.000 ........................................ 62
3.26 Circles Assignment Difficulty Level. Control group average score: 0.8; Experimental group average score: 1.04 ........................................ 63
3.27 Circles Assignment Work Time. Control group average score: 1.875; Experimental group average score: 2.4 ........................................ 64

4.1 System Infrastructure .............................................................. 75
4.2 Sample Function Specification .................................................. 79
4.3 Student User Interface ............................................................ 80
4.4 Assessment: Final Scores ....................................................... 83
5.1 FEAT Structure and Workflow ............................................. 93
5.2 Sample Config File .......................................................... 98
5.3 Comparison of test case complexities in two concise test sets for the
    problem expected_value .................................................... 106
5.4 FEAT's coverage with respect to $|S|$ ................................. 109
Chapter 1

Introduction

The software industry spends a tremendous amount of effort and resources on software testing and maintenance throughout the Software Development Life Cycle (SDLC) to improve the quality of software [2]. Software companies usually need to conduct various types of software testing during the entire SDLC to ensure the quality of software (Figure 1.1). However, all software engineers, not only software testing engineers, are expected to be responsible for the quality of software products. Many studies [3] have shown that lack of solid Computer Sciences education undermines the programmers’ program quality and software testing practice, which will increase future cost for the software industry and entire society such as costs on software reimplementation and fixing software defects. For example, in the year of 2017, the consequences of software failure alone caused at least $1.7 trillion US dollar losses worldwide [4]. In this thesis, we studied how to improve Computer Science education, especially integration of software testing training in introductory Computer Science courses, and envision that this is the key to changing future programmers’ mindset for building high-quality software.

1.1 Motivation

Studies [5, 6] have shown that early introductory Computer Science Education plays an important role in Computer Science students’ entire career life. Good early ed-
Education helps students to learn correct concepts and good habits which will benefit further study and work. Conversely, any missing aspect in early education may handicap students’ performance in the future. However, compared with the high demand for skilled software developers to ensure the quality of software, inadequate training of basic skills, such as software debugging and testing, hasn’t been conducted in current Computer Science education, especially in early Computer Science courses. Instead, most institutes adopt an “implementation-first, test-last” approach that requires students to focus on solving problems and implementing functionalities first. Since instructors place little or no emphasis on systematically teaching debugging and
testing methods and skills in early programming courses, novice programmers usually struggle to locate issues and fix bugs in their own code. Although many Computer Science educators and researchers have studied this problem and suggested changes to the current Computer Science curriculum to reflect the importance of software testing skills, the improvement has never been as easy as expected due to many realistic limitations.

One such suggestion is to add new courses on the topic of software testing into the current Computer Science curriculum. However, due to the broadness of modern Computer Science, current CS curricula already contains too many courses such that both instructors and students have little time to spend on the new topics. Another limitation is that, to learn to debug and test, students need to have basic programming skills to start writing code. Hence, in many introductory programming courses, instructors are mainly focusing on teaching students to write programs to solve problems or implement required functions and have to postpone software testing training to more advanced programming courses. This “test-last” strategy may mislead students to reduce their awareness of the importance of debugging and testing in software development. Students don’t see the necessity of doing software testing through the experiences of completing tasks without doing testing and seeing immediate consequences. The misconception students build at the early stage may make them think of software testing as an unnecessary extra work, which will hinder them from performing serious software testing in future work.

Therefore, researchers [3, 7, 8, 9] have studied integrating testing, like test-driven development (TDD), into CS courses. However, as we stated before, these efforts face some common roadblocks in practice. Edwards [10] summarizes them as five major challenges as follows:
Novice programmers are not ready for testing until they learned basic programming knowledge.

Instructors do not have enough lecture hours to teach a new standalone topic on software testing.

It’s hard for course staff to grade the correctness and completeness of students’ test cases.*

Students need fast, informative feedback on their test cases and other testing practice.

Students need to see the value of software testing.

Like Henrik pointed out [11], we should not treat software testing as a separate topic in Computer Science education. Instead, it should be integrated into all programming activities. Fundamental curriculum and course material changes may need to be made to overcome these five challenges since it is unlikely for students to self-develop systematic testing skills without the help of instructors.

1.2 Research Overview

In this thesis, we study how to effectively teach basic Computer Science concepts, and integrate software testing in early Computer Science education with the help of well-designed educational tools. Our tools were built with the five challenges (Section 1.1)

* Test Case: A test case is a pair of input and its expected result used by a software tester to verify whether the target software or one of its features is working as it is designed to do by comparing the actual test result with the input and the expected test result. For more information about test cases, refer to Section 2.2.
in mind and aim at providing a good working solution for teaching basic software concepts and software testing as well as suggestions for future Computer Science course design. The test-first educational strategy helps students acquire necessary comprehension and analysis skills needed to write reliable code and software. We aim at teaching students better understand their own code, write effective and good test set to discover potential code pitfalls, and providing them with frequent, direct, and informative feedback to guide them improve their code or test set. We hope these efforts can foster a new culture among students that they need to be responsible for their own code, and be able to understand, explain, and be confident about their own code through thorough testing and debugging. We conducted the following three studies to verify our ideas:

1.2.1 Improving Students’ Understanding of Code Execution

The first step of writing good code is to be able to understand program execution, including a good understanding of basic program concepts and how program run-time systems execute programs. However, despite instructors’ great efforts and intentions, student programmers often lack comprehension and analysis skills for programming. A student may believe that once his or her program runs correctly on the first few runs and on some sample data, it must be correct. When a program misbehaves, students tend to add print statements and keep modifying code to check the results until they fix the issue. Advanced integrated development environment (IDEs) can provide rich debugging information hence making students less motivated to carefully read programs to understand code logic and predict code behaviors. This kind of “Trial-and-Error” method provides slow feedback and usually produces poor code in terms of both design and implementation.
In the first project, we build a programming training tool to teach basic programming concepts and program execution process for novice programmers who have very little or no coding background. The educational tool provides code execution state visualization, requires mental engagement to learn actively, and enables students to interact with the tool to gain immediate result and feedback. The goal of the tool is to perform as a teaching assistant that allows students to study at their own pace and it can provide a theoretically unlimited amount of exercises to students to help them learn the concepts taught in regular lecture hours.

Instructors need to decide what programming concepts to teach and in a proper cognitive order. To enable students to make steady progress, instructors usually need to build exercise problems to cover each basic and more advanced program concept. Each exercise problem comes with its own program consisting of a piece of hidden initialization code, a visible code snippet used to explain the desired programming concept, and a set of simple example code snippets for each incorrect answers to help correct students’ misconceptions.

On the other hand, students first need to read a graphical program execution state diagram to get an initial state of code execution generated by the hidden initialization code. After that, students need to mentally run the visible code snippets of the exercise programs to understand the logic and target program concept by themselves without the help of a compiler or any grader. In the third step, they are required to match their mental representation of code execution state with the graphical representations provided by our tool in a multiple-choice problem form. The tool will then either congrats students on their correct choice or provide them with the example code if they made an incorrect choice.

Moreover, the tool can generate a large number of variant exercise problems based
on a small set of template programs wrote by teaching staff through applying multiple stage randomization operations on each template programs if necessary. The complexity of template programs increases along with the coding knowledge taught in class.

As the first step of learning software testing, students don’t need to write any test case in this study since they still need to study the basic programming concepts (Section 1.1 Challenge 1). Here students are expected to read and comprehend source code, envision how a sequence of statements will behave, and predict how a change to the code will result in a change in behavior. Students need explicit, continually reinforced practice in hypothesizing about the behavior of their programs and then experimentally verifying (or invalidating) their hypotheses.

1.2.2 Improving Students’ Test Case Generation Skills

As students had enough training on reading, writing, and understanding programs. We built an interactive learning system for teaching software debugging and testing skills. We need to teach students to write high-quality test cases to detect defects in their own code. However, it usually requires a lot of effort and is very tedious for teaching staff to check the quality of students’ test cases and give them feedback or teaching them how to debug their own code (Section 1.1: Challenge 2 & 3).

Therefore, we built an interactive learning tool to give students rapid and informative feedback to help them practice their debugging skills and improve the quality of their test cases for a coding assignment (Section 1.1: Challenge 4). This interactive tool is built upon a large number of erroneous student solutions selected from previous course sessions for the same set of coding problems. So when students in the current teaching session write test cases for a particular coding problem, the tool
uses a small subset of erroneous programs for the same coding problem from previous sessions. The corpus of erroneous programs is carefully selected and ranked based on program complexity and how buggy they are. Then students interact with the tool as follows:

1. Students submit their own test set for the problem
2. The tool evaluates the completeness of their test set by running the submitted tests on internal erroneous programs
3. The tool returns the simplest erroneous program that the student’s test set failed to catch as feedback
4. Students need to read the erroneous program, debug, and identify the issue of the returned program.
5. Students update their test set to also catch the tool returned erroneous program
6. Students repeat the process until they are satisfied

Students iteratively perform those interactions with our tool to improve the quality of their test set. During this process, students not only learn how to write good test cases but also need to learn how to read, understand program execution, and debug programs.

1.2.3 Generating Test Cases for Grading Students’ Programs

With the previous two exercises, we expect students to be able to understand code execution and write good test cases. However, there might still be hidden bugs in the code, or their test set coverage is not perfect like in the real world where bugs
are impossible to avoid. We build a test case generator to automatically generate test cases that can be used in machine graders to assess the correctness of students’ programs and as prompt feedback for students. As in many other machine graders used by various Automatic Programming Assessment systems, internal test cases play a critical role and determine the quality of the machine graders.

Traditionally, test cases are created by human experts or using machine auto-generation methods based on the problem definition and sample solutions. Unfortunately, the human approach cannot anticipate the numerous ways that programmers can construct erroneous solutions. The machine auto-generation methods are complex, problem-specific, and time-consuming. In this work, the internal test cases are carefully calculated with the guidance of millions of erroneous student programs inspired by crowd-sourcing ideas [12]. The selected concise test set contains a minimum number of test cases yet can provide better coverage than test sets built by experts.

We then used the tool precomputed test cases of increasing complexities in a machine grader to assess the correctness of a student’s code. During the grading process, the first test cases failed the student’s code will be returned to that student as feedback for debugging code before resubmission. Students repeat this process to iteratively improve the quality of their code.

We use this study to verify students’ improvements on debugging and testing skills. In the meanwhile, we teach students to treat software testing as a serious practice since there may still be unexpected issues with your code even after careful testing the could cause serious consequences. Secondly, our grader return test cases to students in order of complexity and in an interactive manner, this approach can gradually familiarize students with the complexity of the problem and increase their confidence by tackling returned test cases one by one.
These three works were just a start on teaching software testing. More advanced pedagogical methods, educational tools, course material changes, assignment re-design are needed to help students first foster a culture of writing reliable code with thorough tests. We envision these training will benefit our students in their entire coding life.

1.3 Contributions

The contributions of this thesis are as follows:

- We built a web-based educational tool that teaches students to understand code execution. The tool can provide several benefits at the same time compared with traditional methods and other existing tools, such as reducing students’ cognitive load, combining multiple techniques to present knowledge, providing immediate feedback, promoting active thinking, and supporting self-paced learning.

- For teaching students to write high-quality test cases, we built an interactive tool that provides meaningful feedback to students and improves their abilities in three aspects. Since we built the tool based on a large number of heterogeneous student incorrect implementations of coding assignments, the tool can provide better testing coverage to judge a student’s test cases’ quality with more variant erroneous implementations as feedback for the student.

- As test cases play a vital role in automated unit testing tools, to better assess students’ code correctness and provide them with proper test cases to help them make steady progress, we built a software toolchain that generates high-quality test cases based on a data-driven method. The tool automates the process of
generating test cases with very little human effort and runs fast. Its generated
test sets are fairly concise, providing better test coverage than the expert built
test sets with a minimal number of test cases.

We built the previous three educational tools following two of Bergins Pedagog-
ical Patterns [13], namely Early Bird and Spiral. We believe basic programming
knowledge and software testing are core disciplines that must be presented early in
every aspect of computer science teaching and reinforced often. Therefore, these three
projects serve as exploratory research on how to systematically adopting advanced
educational tools to improve Computer Science education.

In this thesis, we utilized historical student study data to help build our educa-
tional tools and improve Computer Science education, particularly code execution
and unit testing, and have received promising results in the studies. However, we
only scratched the surface of a tremendous amount of student data available on the
internet. Combining with other techniques, such as Machine Learning and Data Min-
ing, data-driven methods can provide much more and bigger benefits in Computer
Science education.

1.4 Outline of Thesis

The rest of the thesis will be organized as follows:

• In Chapter Two, we will introduce the background and previous works of early
  Computer Science education related to each of our three studies. We will also
  briefly introduced our own previous work in the introductory programming
courses.
• Chapter Three discusses the design and usage of our first multiple-choice educational tool of teaching basic program concepts and code execution. We will also introduce the user study we conducted in one of our introductory programming courses that demonstrates the effectiveness of the proposed tool.

• In Chapter Four, we introduce our interactive system for teaching students to write good test cases. Students’ feedback and statistic results in our user study show that our tool not only can teach students to write high-quality test cases but also improve their general comprehension and coding skills.

• Chapter Five introduces the data-driven test case generation for building high-quality machine grader for grading student programs. The machine grader internal test case generation process, its usage, and performance will be discussed in detail.

• We finalize the thesis with Chapter Six in which we summarize our work, experiences, and discuss future works for the three studies in this thesis.
Chapter 2

Related Work

2.1 Novice Programmer Education

Learning programming is considered a difficult task [14, 15] because programs are less human-understandable and include abstract concepts, complex logic, and dynamic process. To understand the fundamental concepts of programming and be able to write programs to solve real-world problems can be challenging to novice programmers and cause high drop-out rate [14]. Hence, Computer Science educators have been passionately working on building better Computer Science curriculum and good teaching methods to help students overcome various difficulties in learning programming, especially in early programming courses [16, 5, 6, 17, 18].

Many Computer Science educators claim that “learning through practice” can help students to better understand fundamental concepts through practices. Eckerdal [6] pointed out that students may have many difficulties in both conceptual and practical learning. The reason is that the misconceptions of fundamental knowledge built during students’ early Computer Science education undermine students’ ability to construct correct programs to solve problems. They need to spend a large amount of time debugging and fixing issues and may still fail eventually.

Mead et al. [19] conducted psychological and experimental studies on the development of programming skills and propose to teach programming based on the cognitive “Anchor Graph”, which is built on series of “Anchor Concept” introduced by Meyer
and Land [20, 21]. These Anchor Concepts are fundamental Computer Science knowledge that advanced studies are based upon, which can also perform as measurable milestones of programming skills. Many novice programmers suffer the painful process of having continuously to correct the misconceptions they’ve built in their early programming age [22]. Therefore, more attention should be paid to teaching the Computer Science fundamentals in introductory programming courses.

However, learning and teaching abstract concepts and the complex logic of computer programs can be very tedious and intimidate to both instructors and students. Mastering new concepts require a lot of exercises. Active Learning, or learning by doing, was adopted by many researchers [23, 24], which encourages students to actively doing tasks, exercises to gain skills rather than being merely passive listeners. Therefore, instructors teach building games [25, 26] to engage students and boost their interest, interactive programming environment [27, 28] to help them practice programming skills, machine graders [29] to give students prompt feedback, program visualization tools [30, 31, 32, 33] to better understand code execution etc.

2.2 Teaching Software Testing

Software testing plays a critical role in ensuring software quality and accounts for a large part of software development costs in software life-cycle (Figure 1.1) [34]. However, inexperienced software engineers tend to view automatic testing as a waste of time and as an activity completely separate from programming [35]. One reason for this mindset is due to the misconceptions developed in early Computer Science classes. Given limited course time and resources, teachers in Computer Science classes tend to design course exercises and programming assignments that make the students focus more on functionality requirements and deadlines than on keeping the program
quality high [36]. This could have a negative impact on their later careers and could be a sign that improvements in software engineering education are needed when it comes to testing. Researchers identified this gap between university graduates and industry expectations and proposed that more testing training should be added to Computer Science curricula [3].

Many researchers [37, 38, 39, 40] have been actively working on finding the best approaches to teach software testing so that students gain practical skills as well as theoretical knowledge of software testing techniques. These studies are usually on the following questions [37, 41]:

- What causes bugs to occur?
- What types of bugs occur?
- What is the debugging process?
- How can we improve the learning and teaching of debugging?
- How to make the students recognize the relevance of the testing activity?
- How to motivate the students on using testing ideas in their projects?

Although software testing courses are commonly taught as part of software engineering curricula, software testing and debugging skills continue to be both difficult for novice programmers to learn and challenging for computer science educators to teach [41]. Students frequently see testing only as something that happens at the end of the development process because instructors usually place much more emphasis on implementation and reinforce this concept throughout the early Computer Science education.
Barbosa et al. [8] pointed out that the teaching of software testing should begin as early as possible so an adequate culture of testing can be created. The proposed idea was to require the students to start thinking about testing as early as possible, by including testing-related practices in all phases of the development process. Pears et al. [18] also pointed out that an introductory computing curriculum should make an explicit choice to emphasize a particular theme that will recur throughout the course, such as software testing and debugging in this case. However, educators do notice that there are several difficulties in teaching software testing caused by multiple factors, such as [42]:

- It is challenging to effectively incorporate software testing into an already over-packed curriculum.
- Ad-hoc efforts to teach testing generally happen too late in the students’ career, after bad habits have already been developed.
- The efforts lack the necessary institutional consistency and support to be effective.

Many educators tried various ways to teach students testing and debugging, like test-driven development (TDD) [43, 44, 45], mutation analysis [46], random testing technique [47], fuzzy logic based approach [48] and so on. However, as listed in Figure 1.1, the typical comprehensive testing work in an SDLC consists of various types of testing techniques with different purposes in particular SDLC phases. Not surprisingly, most CS curricula teach testing techniques used in early-stages during an SDLC like unit testing [49] and integration testing [50] etc., as the more advanced testing techniques, such as system testing [51], end-to-end testing, or even stress testing [52], etc., are not proper to teach in common CS courses due to the limited
scale of the course projects in the class and limited lecture hours. However, since early-stage testing techniques and more advanced testing techniques usually share the same cognitive models of reasoning on performing effective debugging and testing, we can still emphasize teaching the basic testing techniques in early programming courses as they will benefit students’ further testing skills, for example, through teaching unit testing.

![Diagram of testing processes](image)

**Figure 2.1**: Illustration of Typical Software Testing Process Using Test Cases

Figure 2.1 illustrates the typical testing process using test cases. Any testing practice can be abstracted as a tester (a program or system etc.) runs a set of test cases against test target (program, software, system, function, feature, etc.). Normally, a test case contains a set of given variable values or testing conditions as the input or condition for a test target. Other than that, a test case should contain expect outputs or target responses corresponding to given input variables
and conditions. Tester runs each test case against the test target and compares the actual and expected test target outputs or responses. Most testing techniques can be divided into two major categories — black-box testing and white-box testing [53], sometimes gray-box testing as well, where white-box testing is performed based on a good understanding of the internal structure of the test target; black-box testing is performed based on the test target specifications only; similarly, gray-box testing may be conducted if we only have limited knowledge of the internal details of the test target.

As long as students know how to perform white-box and black-box testing and build test cases for testing basic program functions, features. They can also run their correct cognitive models to test various more complex test targets in similar manners. Therefore, in our thesis, we will mainly focus on teaching students writing good test cases to perform thorough unit testing, using both black-box and white-box testing techniques.

2.3 Educational Tools in Computer Science Classes

Educational tools are usually developed to meet particular needs in teaching activities. Tools provide richer information, more ways for students to study the class content to practice active learning. Another motivation for tool development projects is to reduce or simplify teachers’ workload. Well-designed educational tools outperform traditional instructor in several aspects since they can:

- help many students simultaneously
- provide instant feedback to students
- support self-paced, active learning
• has richer ways to convey information

• can be personalized and so on

Considering the advantages, many research projects have been devoted to tool development [54]. Here we mainly look at two types of popular educational tools.

**Program Visualization Tool**

Human brains are very powerful for understanding and processing visual information. On the other hand, most programming concepts, algorithms, and data structures are abstract concepts with no obvious graphical form. Moreover, programs and algorithms are dynamic artifacts; capturing their essentials is a challenging task for novices. Good visualization tools can help engage and motivate students to study complex knowledge through more enjoyable ways. Not surprisingly, much research has been devoted to the visual presentation of the structure and operation of programs and algorithms [55, 56].

Some educators use visualization tools to teach basic programming principles. The tools map the basic calculation and logic into visual effects so students can have a better understanding of the internal process [57, 58, 59]. For example, Jussi [60] used LOGO language and Turtle graphics in programming class and witnessed an increased passing rate and decrease drop-out rate. Similar results were shown in Cooper’s work [59], which shows providing a visual interpretation of abstract concepts help engage students in increasing their interest, knowledge, and skills. Tang *et al.* [27] built a web-based programming environment with a simplified in-browser GUI library for interactive Python programming and a browser-based tool for visualizing the execution of event-driven Python programs. The tool enables students to build their own graphical games written in Python and attracts a lot of online students to take
this programming course.

Code visualization tools focus on visualization of static structures or display the
dynamic aspects of program execution, and are often associated with programming
environments, discussed below. Tools that reflect code-level aspects of program beh-
avior, showing execution proceeding statement by statement and visualizing the
stack frame and the contents of variables. For example, Guo [30] developed an
embeddable web-based program visualization tools named “Online Python Tutor”
that has been widely integrated into various learning platforms and online tutorials.
The tool provides line-by-line execution visualization showing the detailed internal
programming execution information graphically including variable values, call stack
information, references and so on.

Other visualization tools like algorithm animation tools [61, 62] capture the dy-
namic behavior of code in different ways. Algorithm simulation allows a teacher to
demonstrate algorithm execution directly, allows novice programmers to construct
algorithms and visualizations graphically.

**Automatic Programming Assessment Tool**

Teaching and learning rely on assessment to measure performance [63]. Various
programming assessment tools have been developed for different teaching activities
in Computer Science classes for over forty years [64]. Especially in recent years, when
online programming courses become popular, there are usually a large number of stu-
dents in the class where grading assignments can become unmanageable. In this situ-
ation, automated assessment tools can tremendously reduce the educators’ workload
and in the meanwhile provide timely and high-quality feedback to students [63, 65].
Therefore, Universities and companies have built automated systems for judging pro-
gramming competitions, such as UVa [66], TopCoder [67], and Google Code Jam [68].
Educators have created Leetcode [69], Rosalind [70] and Project Euler [71] to teach programming skills and algorithms through problem-solving exercises. Both students and instructors may benefit from these programming assessment tools.

Automatic assessment tools can help evaluate various student programming skills. Most tools are used to check the correctness of program execution (e.g., [72, 73]). Typically, this is done by comparing a students program output with results provided by the teachers gold-standard solution program against a set of carefully selected internal test cases. Educators usually need to spend a lot of time in designing the internal test cases etc.. These may require educators extra efforts to make these tools reliable and useful hence become a common obstacle to their adoption in Computer Science classes [74]. Therefore, many researchers have worked on test case generation [75] rather than manual test case construction.

Many educators now include software testing activities in programming assignments. However, students often consider software testing to be boring. They may only focus on output correctness on provided sample data and do less testing on their own [10]. On the other hand, hand-grading students’ test cases required a tremendous amount of efforts for instructors and are rather tedious. Goldwasser et al. [76] proposed requiring students to turn in tests along with their own programs, and then running every students tests against every others program. Similarly, Edwards et al. [77] claimed that running each students tests against every other students solutions is the most effective predictor of the underlying bug revealing capability of a test suite. This all-pairs testing method was adopted to make testing as a fun and competitive activity and allow students to learn from each other. However, this method also suffers some critical drawbacks like inconsistent feedback from peer students, poor test coverage, slow feedback, etc.
To overcome these disadvantages and provide consistent and instant feedback to students, researchers have developed useful tools [78, 79, 8] to automatically evaluate not only programming assignments but also their own test cases for the assignment. These tools are aimed at providing adequate feedback to evaluate the learners’ performance concerning programming and testing. Mutation analysis is another popular approach to evaluate the test set quality in many automatic assessment tools [80, 81, 82]. DeMillo et al. helped popularize mutation testing [82], in which students simultaneously submit an implementation and set of test cases; the submitted implementation is mutated in various ways, and the mutants are used to determine how capable the test suite is regarding finding bugs. However, their approach does not assess students consistently, as the mutants presented to each student are highly personalized.

Most of these tools share the same feature that they can provide students immediate feedback for improvement. However, the feature of instant feedback may encourage students to follow a trial-and-error approach to learn programming and writing test cases. This problem can be solved by limiting the number of submissions permitted, or by specifying a minimum delay between subsequent submissions [18]. Moreover, instead of directly returning students the tests they failed on, returning other indirect information could be more useful to train students’ skills.

2.4 Data-Driven Teaching

People are all familiar with traditional education, where instructors choose and organize teaching materials based on personal teaching experiences. The teaching quality largely depends on instructors’ experiences and time spent on course preparation. There is seldom personalized education and not enough individual guidance for each student in traditional classes. Students struggle to wrap their heads around new
knowledge learned in lectures; Instructors exhaust to answer question, grade homework and so on. This situation has come to a change in recent years.

The wide adoption of modern educational technologies has changed our education in many aspects. Various educational tools, especially web-based educational tools, has enabled instructors to teach a large number of students at the same time without spending too much more time so as to quickly gain teaching experiences in a relatively short amount of time. Therefore, massive open online course (MOOC) and many MOOC websites have become very popular nowadays. On the other hand, with these technology advances, students are able to perform the self-paced study at their own will, build personal study profiles to embrace personalized education provided by those tools and systems.

The widespread use of the modern educational technologies not only brings convenience to both instructors and students but also produce large volume and variety of data at a fairly high speed so as to bring education to “Big Data” [83, 84] era of personalized instruction, responsive formative assessment, actively engaged pedagogy, and collaborative learning, etc. The precious data can help us better understand student learning and help us to create and deliver high-quality teaching materials that engage students. Researchers work in the fields of educational data mining [85] and learning analytics [86] dig into the educational big data using Statistical Analysis, Data Mining (DM), Artificial Intelligence (AI), and Machine Learning (ML) techniques, etc. to greatly improve the current education system.

**Build Better Statistical Student Models**

Modern education systems and tools can produce varied types of data that are recordable and analyzable. We can build better statistical models through tracking students’ learning activities and study log data etc. Based on the *Crowd-Sourcing*
ideas [12, 87], we can infer the best cognitive models and knowledge graphs by discovering the common misconceptions, pitfalls, difficulties, gaps, etc. in students’ study process.

**Optimize Pedagogical Strategies**

Based on the cognitive models and knowledge graphs, instructors can better select course materials and improve pedagogical strategies. For example, Beck *et al.* [88] uses Reinforcement Learning technique to derive pedagogical policy. More recently, Chi *et al.* [88] studied Markov Decision Process (MDP) to improve teaching policies.

**Support Personalized and Self-paced Study**

Student study data logged by various tools, e.g., interactive study tools, enables us to establish each student’s individual model and study pattern. Based on the generic student models and the overall student statistics, we can evaluate students’ knowledge so as to recommend them the most suitable sequence of instructions to optimize the study result [89].

**Build Intelligent Teaching Systems**

Historically, most claimed intelligent tutoring systems (ITS) had been built upon human knowledge engineering and expert experiences. The teaching materials, pedagogical strategy, assessment, and feedback are all manually built by instructors. Hence, the tools usually tend to be biased and not so “intelligent” when it comes to students' individual needs. Luckily, modern data-driven methods can enable rapid development and iteration of intelligent tutoring systems [90]. Modern ITS systems can provide more effective help to students due to its better understanding of teaching aim and students.

**Intelligent Feedback Generation**

Data-driven approaches also allow us to overcome the traditional feedback/hint
generation challenge in ITS systems. Before, instructors need to derive an extensive
graph of possible states and actions so that the ITS system know what feedback/hint
to provide to help students. Rich student data enables ITS system to automatically
generate feedback/hint on demand, direct them to the correct path by providing the
most proper help at the right time. Many researchers have been working in this area,
such as Barnes et al. [91] used Markov Decision Process (MDP) to generate hints
based on historical data; Timotej et al. [92] used a data-driven approach to synthesis
sequences of program edits as hints to direct students to reach the final correct pro-
gram solution.

Since we noticed the increasingly important role data-driven teaching, particularly
the data-driven approached based education tools, may play in modern education,
we decide to study on using data-driven approaches to build useful educational tools
for teaching important Computer Science concepts.
Chapter 3

Improving Students’ Understanding of Code Execution

According to most existing Computer Science curricula, the main focus of the first several introductory programming courses is to teach students basic programming concepts and language features. After taking these introductory courses, students should be able to read, write simple programs to solve simple application problems. However, novice programmers usually find these courses quite challenging so that the dropout rates are high due to the difficulties in understanding the abstract and highly dynamic programs.

3.1 Related Works and Background

Despite Computer Science educators’ consistent efforts on the research related to programming education [5, 93], learning to program remains a challenging task to novice programmers [15, 5, 94, 95]. Programming, like driving a car, requires applying multiple types of skills rather than just a single skill. An expert programmer needs to use many skills at the same time such as coding, code inspection [96], debugging, and testing, etc. Among these essential skills, coding as the first and critical skill also consists of a set of hierarchical skills [97]. Research [98] suggested that an ideal pedagogical way is to teach coding skills from the bottom up — students will learn the basics of syntax and gradually move on to semantics, structure, etc. Since the early stages of learning about coding usually greatly depend on the superficial con-
cepts and skills, any misconception at an early stage can be a serious obstacle in further study. The common misconceptions arise from two aspects: firstly, common generic programming language concepts, syntax, and semantics, etc.; secondly, lack of understanding of code execution process inside a computer.

Based on one particular programming language like Python, C, or Java, etc., most CS1 courses start with teaching students the basic concepts including language syntax, data model, expression, statement, control flow, and basic data structures. Instructors rely on carefully built examples to help students to build a correct mental structure of all those abstract conceptual knowledge, usually called “knowledge scheme” [99, 100, 101]. People use their internal lower level schema to build hierarchical structures or networks of schema so as to internalize more advanced knowledge [99]. Through experiences in everyday life, people constantly create, revise, and use various knowledge schema to understand, reason about, and solve problems. Similarly, students need to build their own schema of programming via learning the examples and exercises provided by instructors. However, due to the limited time and effort of instructors, students are usually in lack of experiences with new concepts and build incomplete or incorrect schema. Moreover, the defects propagate in further study and start to pile up misconceptions leading to failure. Incomplete knowledge schema and insufficient confidence give student trouble to read and write code and impose a heavy cognitive load on novice programmers. Therefore, it is vital to ensure each student has enough exercise and gain immediate feedback to build clear and correct their internal knowledge schema.

Besides a good understanding of program itself, students also need to understand how does a machine represent and execute a program. The static aspect of a program is explicit and easy-to-follow, yet it is only a tip of an iceberg as for a highly dynamic
computer program. On the other hand, the hidden dynamics requires students to have vital “mental models” \([102, 103]\) of a computer runtime environment, and be able to run a mental simulation based on the mental model to reason about program execution and predict the program outcome. A mental model is an internal representation of something in the real world; it reflects people’s thought process about how things and stuff work, how they are built, how to interact with them, and how they may behave.

Due to limited course time and effort, students usually can only build incomplete mental models of a computer due to limited course time and programming practice. Unfortunately, in most CS1 courses, students are required to start writing programs after a quick introduction to programs and program execution. With a defective mental model of computer and knowledge schema of programs, to different extents, students treat a computer as a black-box so as to write and debug programs following a Trial-and-Error method due to lack of self-confidence and incapable of predicting the program executor’s behavior. Moreover, fixing an early stage formed defective mental model may require even more work than building a viable mental model in the first place.

Therefore, Computer Science educators have been researching on finding effective methods to help students build correct knowledge schema of programs as well as a representative mental model of computer. Many research studies have shown that various program visualization tools \([104, 105, 106]\) can be very helpful for students to understand the abstract program concepts, algorithms, and data structures and so on. A code visualization tool provides either static or dynamic or both aspects of program execution. For example, BlueJ \([107]\) uses UML-like diagram to provide static code representations of object-oriented concepts. However, other studies have attributes
students’ major difficulties to poor understanding of the dynamics of program execution. Kaczmarczyk et al. [108] pointed out, based on their research data, the major cause to students’ misconceptions is the relationship between language elements and underlying memory usage. Hence, visualization tools provide dynamic views of program execution considered even more helpful. Some tools like jGRASP [109], UUhistle [110], and Online Python Tutor [30] all provide dynamic view of program execution.

jGRASP is a lightweight integrated development environment (IDE) which can visualize three types of information including diagrams of code control structures, UML Class Diagram for Object-Oriented programs, and dynamic views of objects and primitive variables while users step through a program (Figure 3.1). UUhistle can visualize variables, the call stacks, the heap, etc. for a subset of Python language features and can also allow students to perform simple interactions with the program execution process (Figure 3.1). However, both tools are somewhat complex and require students to download and install the software on their own computers and require the installation of extra Java Virtual Machines (JVM). Instead, Online Python Tutor (OPT) retrieves Python program execution traces generated by real Python debugger on web server and then use JavaScript visualization code running in users’ web browsers to visualize many aspects of code execution to help novice programmers to better understand the code execution process, such as variable values, objects, references, parameter passing, call stacks, heap, return values and so on. OPT sends all user programs to run on its website backend server in order to generate code execution trace. Since its code execution visualization can be easily accessed via normal internet accesses, this tool is very easy to use and become very popular among novice programmers and CS1 instructors. Therefore, we integrated OPT’s JavaScript
visualization code and built our own web-based programming environment and code execution visualizer (VizMode) to accommodate a large number of students in our MOOC courses. In VizMode, since both code execution and visualization happen in
users’ browser, students can easily open up a new window to start writing, visualizing, and debugging their code as shown in Figure 3.2.

Figure 3.2: Online Python Code Execution Visualizer: Online Python Tutor and CodeSkulptor VizMode
Although these program visualization tools have been widely used, more research indicates that simply looking at the visualization is not good enough to successfully build a good mental model of code execution [105, 104]. A more effective way is to engage students with a more interactive tool and working on some task that encourages them to perform “active learning” [111, 112] to gain a deeper understanding. Active thinking means students need to run a mental simulation to predict how the execution proceeds instead of passively sitting in front of the screen and watching the dynamic visualization as animation. As Craik and Lockhart [113] suggested “learning and retention are related to the depth of mental processing”. Studies [114] show that students learn much better when they are interacting with the material.

Researchers have been working on building various educational tools to teach programming concepts in CS1 courses. C-doku [115] is a web-based application designed to encourage students to actively read the code to figure out proper inputs or outputs to satisfy problem requirements. However, this tool lacks necessary code execution visualization and informative feedback for students. Jeliot [116] is an application that visualizes the data and control flow of Java programs. It can also randomly generate multiple choice questions anywhere in the program when students read the code, for example, a common question might ask for the result of an expression evaluation result. However, Jeliot does not isolate basic programming concepts very well which may overwhelm students, and cannot provide informative feedback to students either. Jeliot requires students to download and install it on their own computer, which also imposes an extra burden on novice programmers. Moreover, both C-doku and Jeliot do not have good randomization mechanism to generate variant problems. In order to provide sufficient exercises to students, researchers have proposed “parameterized questions” to help reduce instructors’ problem authoring burden that, at each presen-
tation time, the parameterized questions will be instantiated with randomly generated parameters so as to generate a large or even unlimited number of similar questions about the same concepts [117]. The randomized variants of problems can also help resolve the potential plagiarism problems in programming classes. Since feedback is critical to human knowledge acquisition, researchers also work on generating informal hint or feedback to students [118].

Therefore, in this work, we build a new educational tool that engages students with performing the mental simulation of code execution with the help of code visualization to answer questions in one of our CS1 courses. Students’ answers will receive immediate feedback and hint to help them better understand the target concepts.

3.2 A Tool for Teaching How Programs Are Executed

In order to better help students to build correct program knowledge schema and mental model of computer, we designed a tool named iTeach with the following requirements and goals in mind that the tool:

- Need to be easily accessible by a large number of students and provide self-paced learning to students simultaneously

- Provide rich information to students about basic program concepts and code execution process

- Encourage students to actively learn by interacting with the tool

- Reduce cognitive load on students by gradually teaching basic concepts before more advanced tasks like writing code by themselves and solving problems
• Show tolerance, allow students to try multiple times on the same problem and provides instant, informative feedback

First of all, as a web-based tool *iTeach* is built to overcome the limitation of lecture hours and instructor efforts. Students can easily access the tool as visiting any other websites. They can work on their own individual sessions independently as much as they want, as long as they want until they completely understand a new concept.

Our tool is designed to teach students the ability to trace a program, a mental simulation process, which means analyzing code execution in mind to simulate program operations and state changes. Tracing a program is an essential ability to programmers in their everyday work life. Expert programmers usually can use “symbolic tracing” [119] to comprehend and inspect code by using only generic values while tracing the code execution process since they’ve already had profound knowledge schema of program itself as well as the program’s runtime system. On the contrary, “concrete tracing” [119] comes in handy for novice programmers to understand the low-level abstractions via using specific values in the programs to understand the execution process [120]. Therefore, in our tool, we build exercises to ask our students to perform mental simulations and concrete tracing of programs.

Since novice programmers usually consider writing programmers as a challenging task, especially when they are not familiar with basic program concepts. Therefore, we built *iTeach* to present exercises as multiple choice questions to reduce students’ cognitive burdens. With this format, students do not need to write any code by themselves. Instead, they can directly focus on studying the provided code to learn program basics.

Our tool can automatically generate the visualization of the program states before
and after the code execution. The graphical view of program state illustrates variable values, call stack frame, references and so on. Students will need to run a mental simulation to trace code execution that bridges the correct pair of initial and final state of code execution. This practice can help students tune their mental model of program runtime system.

Since novice programmers are not good at symbolic tracing, the tool will initialize each program with concrete variable values and enable students to perform concrete tracing. If a student got a wrong answer, our tool would provide instant feedback to students to explain the target concepts in a friendly way. Also, our tool will allow students to retry the same problem multiple times with varying initial states until they learned the concepts. With this tool, students learn at their own speeds and will always have immediate feedback.

In the following sessions, we will introduce the details of our tool and how students and instructors use it in CS1 introductory programming courses.

### 3.2.1 Student View

Students can easily access the *iTeach* tool by visiting the tool website [www.tooldomain.com](http://www.tooldomain.com) same as visiting any other website. Students may be able to register and login to their personal account panels to manage their study materials and progress in the future. As complimentary study materials to regular lectures, we present exercise problems to students as short quizzes consisting of several multiple-choice problems related to the programming concepts or language features taught in the most recent lectures. At the current stage, we simply let students to directly access the programming exercise problem page via URLs provided by instructors, for example, [http://www.tooldomain.com/student/quiz/5a4d1f630bd5884f516dfbab/](http://www.tooldomain.com/student/quiz/5a4d1f630bd5884f516dfbab/), where
the alphanumeric code in the URL is the unique quiz identifier.

The tool quiz page looks like in Figure 3.3 where it shows the quiz name, and the student’s progress (the second problem out of total 5 problems in the short quiz). The left panel is a graphical visualization of initial code execution state which is generated by a piece of initialization code snippet (initialization code). The middle panel displays a visible code snippet (visible code) which runs after the initialization code to produce final code execution state. The right panel contains graphical views of four different code execution final states where there is only one correct choice given the initial code execution state and the visible code and other three distractor graphical final states.

![Figure 3.3: iTeach Tool Quiz Page](image)

To solve the multiple-choice problem, students’ task summaries as follows:

1. Read the graphical state diagram in the left panel to parse the initial state for
the next visible code shown in the middle panel. For example, in the left panel in Figure 3.3, the initialization code create two variables \( x \) and \( y \) equals 4 and 7 respectively.

2. Read the visible code snippet and mentally simulate the program execution process using “concrete tracing” starting with the initial state as presented in the left panel and reach a final program execution state. The example visible code calls Function \( f0 \) on Line 10, but does not change the value of \( x \); Line 11 calls Function \( f0 \) and update \( x \) to 2; the last line pass the updated \( x \) value to Function \( f1 \) and then update \( y \) to 9. Therefore, the correct final states corresponding to the visible code should be “\( x = 2; y = 9 \)”

3. Inspect each graphical view of code execution final states in the right panel and match the mental representation of final code execution state with its graphical representation.

4. Confirm select with the tool and check the result. Proceed to next problem if correct, otherwise study the tool’s feedback and go back to Step 1 to work on the same problem again.

Students may make correct or incorrect choices. Our tool needs to give them immediate and informative feedback and hints. Figure 3.4 contains feedback for both correct and incorrect answers. If students made correct choice the tool will congrats students and allows them to move on to next problem. However, if students made the wrong choice, our tool will provide the students with the simple example/hint code to familiarize them with the basic concepts covered in quiz problems. The tool grayed out “Next Problem” button and encourage them to “Redo Problem”.
When a student made an incorrect choice, the tool showed a simple example code snippet to illustrate the concept “passing variable value \((q = 5)\) to a function call \((f0(q))\) won’t change the variable value in global scope \((q = 5)\) after the function call”

Figure 3.4: \(iTeach\) Tool Quiz Feedback Pages

In order to enable students to work on the same problem multiple times without presenting them the identical problem. We introduced two folds of randomization mechanism in the tool, as illustrated in Figure 3.5. The first type of randomization happens on the initialization code side. The initialization code contains randomizers acting as a “parameterized program” [117] so that it can generate varying initialization state with different random seeds. Our tool can generate an unlimited number of
random seeds for the randomizer in initialization code so as to generate different initial states and problems.

Each “Program” is a combination of its “Initialization Code” and “Visible Code”. In order to generate four final states, each problem consists of four pieces of code snippets including the target visible code (in “Target Program” panel) which will be visible to student in the middle panel on quiz page (Figure 3.3) and three distractor code snippets (in “Distractor Program” panels)

Figure 3.5 : *iTeach* Tool Problem Construction Process

After that, the visible code starts with the initial state and performs more operations to reach the final state. Due to the differences between the three versions of distractor code (hidden from students) and visible code, they would yield different final states even with the same initial state. Therefore, as the second type of randomization, we shuffle the final execution state diagrams and display them in varying
order in the right panel on the quiz page (Figure 3.3). Thus, thanks to these two folds of randomization, students can see different yet essentially the same problems theoretically unlimited number times until they understand the concepts we plan to teach them.

### 3.2.2 Instructor View

Each short quiz we presented to students consists of a set of multiple-choice problems built by instructors, either manually or with the help of the *iTeach* tool. Instructors can login into the instructor dashboard (Figure 3.6) through a hidden URL and with valid credentials. Instructors can manage problems and quizzes on problem and quiz panel respectively. We will introduce seed panel later in this section.

![Instructor Dashboard](image)

In order to synthesize quizzes, instructors first need to create new multiple-choice problems on the Problem Panel as shown in Figure 3.7. Instructors can create/delete problems, edit existing problems such as renaming problem, modify initialization code or visible code snippet and so on. Also, instructors can preview how the problem will
be presented to students.

Figure 3.7: *iTeach* Tool Problem Management Panel

Figure 3.8: Overview of *iTeach* tool Quiz and Problem Structure

As shown in Figure 3.8, each multiple choice problem contains one target program and three distractor programs. All four programs differ slightly yet cover the
same program concept. In the meanwhile, instructors also need to attach a piece of example code or hint code for each distractor program as “code-revealing hints” [118] for students if they picked the wrong final state choice generated by the distractor programs.

Figure 3.9: Problem Authoring Page

Figure 3.9 shows the tool’s multiple choice problem authoring page. There are four small “choice panels” with three embedded code editors. The very first panel labeled “Target Program” is always the target program whose generated final state diagram will be used as the answer to the multiple choice problem, and all other three panels are labels as “Distractor Program” panels which will generate distractor choices. Notice among the three code editors. The third code editor in each choice
panels is used to enter example code or hint code that covers the same program concepts as in the target/distractor programs in a more simplified way.

The following Figure 3.10 is an example choice panel. In this example, the initialization code creates a variable with a random value ranging from 2 to 9. The visible code demonstrates the usage of composite function calls and uses variable and expression as function call parameters. In case students don’t understand the concept, the example code illustrated the same concept using a simpler program and a concrete value as the function parameter.

![Distractor Program 1](image)

**Figure 3.10 : iTeach Tool Problem Sample Choice Panel**

Our tool presents the study materials to students as quizzes consisting of a set of multiple-choice problems. The quizzes were synthesized from existing problems built by instructors beforehand. Instructors built quizzes on the following page as in Figure 3.11. The left panel displays all available problems. Instructors can select problems from the left panel which will then show up in the right selected problem
panel. After naming the quiz, instructors can create a new quiz and manage them in the quiz admin page (Figure 3.12).

Figure 3.11: Synthesis a Quiz From Existing Problems

Figure 3.12: *iTeach* tool Quiz Admin Page
As shown in the Figure, instructors can still add/delete/modify problems in exiting quizzes until they are satisfied with the quiz. Then instructors can release the quiz to students using the “Release Quiz” button, which will generate a public quiz URL for students.

In the current version, instructors need to manually enter target/distractor programs to build problems in order to ensure high quality of problems. We also proposed an alternative method to automate massive problem authoring with a small number of template programs (we call them “seed programs”) as shown in Figure 3.13. Instructors need to first write a set of high-quality seed programs. Ideally, each seed program needs to cover one individual program concept. Any seed program can be used to generate a new problem. Besides the seed program itself, our tool can automatically generate mutant programs by applying predefined mutation operations on the seed program, for example, using a Python library called MutPy [121]. This method eliminates tedious work for instructors and enables them to focus on building high-quality seed programs. Therefore, instructors can quickly build many problems.

![Figure 3.13: Automated Mutant Program Generation](image)

Figure 3.13: Automated Mutant Program Generation
3.2.3 Architecture

In order to provide easy access to our students, we built *iTeach* tool as a web-based tool and deployed it on Amazon EC2 machines. Considering the possible future use in large classes, like in Massive Open Online classes, we split the tool’s frontend and backend and deploy them on two separate machines since this architecture can accommodate more traffic at the same time compared to running multiple virtual servers on the same server machine. Also, the same design can be easily scaled out when there are a lot more students online, for example, using an *Apache* [122]/*Nginx* [123] reverse proxy load balancer to direct user requests to different server machines, etc. The entire system was built upon various open source software and libraries.

Since students and instructors intensively use the frontend to access the study material, we decided to use the modern frontend development toolkit and frameworks to make it more reliable, efficient, concise, and more intuitive to use. We use *Yeoman* [124] to scaffold the frontend website and use *Bower* [125] to manage various frontend development packages. We then implemented the main logic based on *AngularJS 1* [126] Model-View-Controller (MVC) framework. The GUI was developed using *Bootstrap* [127] toolkit. To automate the repetitive tasks during the entire development process, we use *Grunt* [128] JavaScript task runner to manage all the common tasks. We then deployed the frontend website on one Amazon EC2 Linux machine and used the *Apache* HTTP webserver software.

On the other side, the backend of our tool was developed using *Node.js* [129] web framework *Express* [130]. We use NoSQL database *MongoDB* [131] to store website data since it is agile, efficient, and easy to scale out. The backend was built to provide *RESTful API* [132] services to the front-end server so as to enable users to interact with the system. We run the backend on as a virtual server on port 3000 on another
EC2 machine under a regular user instead of directly under the machine system root user due to the system security concern. Thus, we can use Nginx reverse proxy to forward all HTTP requests from the default listening port 80 to port 3000 and provide API services. The Node.js application was managed by a Node.js process manager pm2 [133], which allows automatically restart our application when it crashes or even the EC2 machine restarts.

![Diagram of System Architecture of iTeach tool](image)

Figure 3.14: System Architecture of *iTeach* tool

We register one domain name on GoDaddy.com and bind the static IP address of the frontend EC machine to www.tooldomain.com and the backend machine IP address to its subdomain named api.tooldomain.com. Hence, users can access the tool GUI by visiting www.tooldomain.com and all API services were provided by backend server listening at api.tooldomain.com. Figure 3.14 shows the big pictures of the entire system.

In the following subsections, we will introduce the usage of the proposed tool and the study result analysis.
3.3 Study Using iTeach

3.3.1 Study Method

In Spring 2018, we recruited participants from students enrolled in the course COMP140: Computational Thinking at Rice University. Student level varied from freshman to senior, but most of the students in the class were Freshmen or Sophomores. Plus, most students were not Computer Science major students (76%) or “Undecided” major students (17.3%) with none or little programming knowledge before. The main purpose of the course is to teach students to design and implement algorithmic solutions in Python to solve real-world problems. In order to do so, the course dedicated its first several weeks to teaching programming fundamentals and familiarizing students with Python programming.

We ran the study in the first two weeks, during the two weeks instructors will cover various programming concepts in Python and build quizzes to help students better digest the concepts outside the lectures. There will be programming assignment modules after lectures and quizzes to evaluate students’ acceptance of new knowledge. Therefore, we deployed the tool to build three quizzes to assist instructors to teach students those concepts taught in the class lectures. Figure 3.15, Figure 3.16, and Figure 3.17 shows five target programs we used in each quiz. Each target program was built to teach one new concept. We released the first two quizzes in the first week and another one in the second week. The first assignment (named Circles) was released at the end of the first week.

The consenting students in the class were randomly assigned to the control and experimental groups. The experimental group students need to take the three quizzes built upon our tool and in the meanwhile, the control group students work on the
regular quizzes typical for the course in our previous course sessions. The two groups work in parallel.

![Program code snippet](image)

**Figure 3.15**: *iTeach* Tool Study Quiz 1

We then evaluate the effectiveness of the proposed tools using two methods. Firstly, we asked our students to complete two surveys (Table 3.1 and Table 3.2) before and after the two-week study. The pre-study survey focuses on students’ Computer Science background and self-reported confidence and understanding level
Figure 3.16: *iTeach* Tool Study Quiz 2
of programming. The post-study revisited students for their newly updated confidence and understanding level after the study as well as their rating and comments on our tool directly. The other method is to check student performance in the Circles assignment after they have worked on the first two quizzes.
<table>
<thead>
<tr>
<th>Survey Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1.</strong> What is your current academic year?</td>
</tr>
<tr>
<td>a. <em>Freshman</em> (0)   b. <em>Sophomore</em> (1)   c. <em>Junior</em> (2)   d. <em>Senior</em> (3)</td>
</tr>
<tr>
<td><strong>2.</strong> What is your current major? (Note that &quot;Undecided&quot; is an acceptable answer.)</td>
</tr>
<tr>
<td>a. <em>I have no interest in majoring in Computer Science.</em> (0)</td>
</tr>
<tr>
<td>b. <em>I have a little interest in majoring in Computer Science.</em> (1)</td>
</tr>
<tr>
<td>c. <em>I have substantial interest in majoring in Computer Science.</em> (2)</td>
</tr>
<tr>
<td>d. <em>I plan to major in Computer Science.</em> (3)</td>
</tr>
<tr>
<td><strong>3.</strong> How interested are you in majoring in Computer Science?</td>
</tr>
<tr>
<td>a. <em>None</em> (0)</td>
</tr>
<tr>
<td>b. <em>I have taught myself a little programming</em> (1)</td>
</tr>
<tr>
<td>c. <em>I have taken a programming course</em> (2)</td>
</tr>
<tr>
<td>d. <em>I have taken several programming courses</em> (3)</td>
</tr>
<tr>
<td><strong>4.</strong> What kind of programming background (if any) do you have?</td>
</tr>
<tr>
<td>a. <em>None</em> (0)</td>
</tr>
<tr>
<td>b. <em>I have taught myself a little programming</em> (1)</td>
</tr>
<tr>
<td>c. <em>I have taken a programming course</em> (2)</td>
</tr>
<tr>
<td>d. <em>I have taken several programming courses</em> (3)</td>
</tr>
<tr>
<td><strong>5.</strong> Please characterize your current level of understanding concerning how computer programs are executed.</td>
</tr>
<tr>
<td>a. <em>I have no understanding.</em> (0)   b. <em>I have a little understanding.</em> (1)</td>
</tr>
<tr>
<td>c. <em>I have some understanding.</em> (2)   d. <em>I have a complete understanding.</em> (3)</td>
</tr>
<tr>
<td><strong>6.</strong> How confident are you that you will successfully complete this course?</td>
</tr>
<tr>
<td>a. <em>I am not confident.</em> (0)   b. <em>I am modestly confident.</em> (1)</td>
</tr>
<tr>
<td>c. <em>I am confident.</em> (2)   d. <em>I am very confident.</em> (3)</td>
</tr>
</tbody>
</table>
Table 3.2 : Post-study Survey

<table>
<thead>
<tr>
<th>Survey Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. How long did you spend on the programming portion of the Module 1 Assignment (Circles) that was due last weekend?</td>
</tr>
</tbody>
</table>
| a. Less than 15 minutes (0)  
  b. 15-30 minutes (1)  
  c. 30-60 minutes (2)  
  d. 1-2 hours (3)  
  e. More than 2 hours (4)                                                                                                                             |
| 2. How difficult was the programming portion of this assignment for you?                                                                                                                                            |
| a. Not difficult at all (0)  
  b. A little difficult (1)  
  c. Moderately difficult (2)  
  d. Very difficult (3)                                                                                                                                             |
| 3. How interested are you in majoring in Computer Science?                                                                                                                                                           |
| a. I have no interest in majoring in Computer Science. (0)  
  b. I have a little interest in majoring in Computer Science. (1)  
  c. I have substantial interest in majoring in Computer Science. (2)  
  d. I plan to major in Computer Science. (3)                                                                                                              |
| 4. How confident are you that you will successfully complete this course?                                                                                                                                             |
| a. I am not confident. (0)  
  b. I am modestly confident. (1)  
  c. I am confident. (2)  
  d. I am very confident. (3)                                                                                                                                         |
| 5. How helpful were quizzes 2-4 in improving your understanding of how simple Python programs are executed?                                                                                                          |
| a. Not very helpful. (0)  
  b. A little helpful. (1)  
  c. Helpful. (2)  
  d. Very helpful. (3)                                                                                                                                             |
| 6. Your level of understanding concerning how computer programs are executed.                                                                                                                                 |
| a. I have no understanding. (0)  
  b. I have a little understanding. (1)  
  c. I have some understanding. (2)  
  d. I have a complete understanding. (3)                                                                                                                   |
| 7. For those of you that interacted with the *iTeach* tool in quizzes 2-4, please enter any comments/suggestions you have concerning the tool in the text box below.                                                        |
3.3.2 Scoring Method

To facilitate quantitative analysis and generate more intuitive interpretation, we encoded each survey question answers as numerical numbers according to their own natural order/level/scale etc. as shown in Table 3.1 and Table 3.2. For example, confidence level: not confident, modestly confident, confident, and very confident were encoded as 0, 1, 2, and 3 respectively. Hence, in the later analysis, we can calculate averaged scores for each survey questions and generate quantitative analysis result to depict the difference between the two groups or before and after the study.

![Figure 3.18: Distribution of Student Study Years](image)

3.3.3 Two Study Groups

Due to absence and late enrollment, there were 68 students in the class that satisfied the study requirements, 27 students in the experimental group and 40 students in the control group. To gain the difference before and after the study, both group students
Figure 3.19: Distribution of Student Computer Science Background

need to complete the two surveys and the Circles assignment. Moreover, students need to take all three quizzes built upon our tool to give meaningful feedback.

Based on the pre-study survey result, Figure 3.18 and Figure 3.19 show the distribution of student years and their previous Computer Science background respectively. As shown in the two figures, the control group students seem to have stronger background compared with the experimental group. We represented each question responses with numerical values as shown in Table 3.1 and Table 3.2. We calculated the average score of study years in the control group is 0.8 and only 0.59 in the experimental group. Similarly, the control group also has higher average score 1.25 as compared to 0.89 in the experimental group on the self-reported Computer Science background. However, the following studies show us exciting results that our tool helped the weaker experimental group made greater improvement and achieved even higher scores on both the surveys questions and the Circles assignment compared
with the control group.

3.4 Study Results

From the survey, we can know better about our students such as their background. We can also compare the differences between the pre- and post-study surveys so that we could look at the changes. The same set of questions were asked no matter a student is in the control group or experimental group.

Students all need to report their ratings on the difficulty levels of the Circles assignment and the time spent on completing the assignment as well. We further check their actual achieved scores on this assignment to evaluate the effectiveness of our tool.

We are also particularly interested in students’ self-reported understanding level of programming execution before and after the two-week study since our tool is specifically designed to help students better understand basic programming concepts. We expect a bigger improvement on the experimental group than on the control group.

Similarly, we also believe the first a few week’s studies is critical to students to help them build their confidence on completing the course. As students feel being left out or fall behind are more likely to drop out. Therefore, we will also check students’ confidence level improvements and the difference between the two groups after the study.

3.4.1 Understanding of Code Execution

As understanding of code execution is the programming fundamental, students need to understand the basic building blocks and concepts of programming before they can write code to solve real-world problems by themselves. It is also the major concern of
our tool. We expect students in the experimental group to gain a better understanding of code execution after using our tool.

![Figure 3.20: Distribution of students’ understanding of code execution BEFORE the study.](image)

Experimental group average score: **0.59**; Control group average score: **0.875**

We included the questions about the students’ self-reported understanding of code execution in the pre- and post-study surveys and noticed a big improvement which revealed the effectiveness of our tool. Figure 3.20 shows the distribution of students’ familiarity with code execution. Since we encoded the four levels of familiarity with numbers ranging from 0 to 3. The average score of the control group students was **0.875**, higher than **0.59** in the experimental group as the control group has a relatively stronger Computer Science background.

Both groups of students learned the same course materials except that the experimental group took quizzes built using our tool, the only difference from the control
Figure 3.21: Distribution of students’ understanding of code execution AFTER study. Experimental group average score: 1.93; Control group average score: 1.725.

The post-study survey results (Figure 3.21) demonstrated a big boost on the level of understanding of code execution among students in the experimental group. The average score of understanding level in the experimental group jumped to 1.93 from original 0.59. The experimental group’s updated average also beat the control group’s new average score of 1.725 after the study even though with a relatively weaker background.

The result was convincing that our tool was more helpful to students on improving their understanding of code execution. Moreover, we took a closer look at the subgroups of students with “no understanding” or “a little understanding” to inspect the tool’s effect on them. The results revealed us more exciting facts.

Figure 3.22 shows the distribution of level of code understanding after the study for students with “no understanding” of code execution before class. The result shows
Figure 3.22: Distribution of students’ understanding of code execution after study among students initially with “no understanding” at all.

Figure 3.23: Distribution of students’ understanding of code execution after study among students initially with “a little understanding”.
that the majority of students in the experimental group believe themselves have “some understanding” as opposed to the control group. More interestingly, for students initially with “a little understanding”, there were 25% students in the experimental group claimed they now have “complete understanding” of code execution after the study as shown in Figure 3.23, a very encouraging change. In contrast, none of the students from the control group made that claim even though they also made big progress.

Table 3.3 : Student Groups’ Average Scores of Understanding of Code Execution Before and After the Study

<table>
<thead>
<tr>
<th>Study Groups</th>
<th>Group Types</th>
<th>Average Scores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Before</td>
</tr>
<tr>
<td>Experimental Group</td>
<td>Entire Group</td>
<td>0.590</td>
</tr>
<tr>
<td></td>
<td>NoU Group</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>ALU Group</td>
<td>1.000</td>
</tr>
<tr>
<td>Control Group</td>
<td>Entire Group</td>
<td>0.875</td>
</tr>
<tr>
<td></td>
<td>NoU Group</td>
<td>0.000</td>
</tr>
<tr>
<td></td>
<td>ALU Group</td>
<td>1.000</td>
</tr>
</tbody>
</table>

Notice: Besides the entire study groups, we also splitted the entire group into small subgroups including: NoU Group: students previously have no understanding; ALU Group: students previously have a little understanding.

Table 4.3 gives us an overview of how much improvements both experimental and control group students have made after the two-week study. As we can see in the table, based on the students’ self-reported results, not only at the entire group but also for subgroups, the students in the experimental group have made much bigger
progress and achieved higher average scores than the control group.

3.4.2 Confidence to Complete Course

Another interesting observation is about students’ confidence level change on completing the course after the study. Figure 3.24 and Figure 3.25 demonstrate the distribution of confidence level change before and after the study in both study groups.

![Distribution of students' confidence level](image)

Figure 3.24: Distribution of students’ confidence level on completing the course before and after study in Experimental group. Average score before: 1.556; after: 1.926.

As we can see in the figure, the average score of confidence level went up from 1.556 to 1.926 in the experimental group before and after the study. However, the same confidence level indication scores within the control group students even went down from 2.025 to 2.0 (Table 3.4).

Since the control group’s background was relatively stronger, it is understandable that their initial confidence level is higher than the experimental group. However,
Figure 3.25: Distribution of students’ confidence level on completing the course before and after study in Control group. Average score before: 2.025; after: 2.000.

Table 3.4: Average Scores of Confidence Level on Completing the Course

<table>
<thead>
<tr>
<th>Study Groups</th>
<th>Average Scores</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
<td>Delta</td>
<td></td>
</tr>
<tr>
<td>Experimental</td>
<td>1.556</td>
<td>1.926</td>
<td>+0.370</td>
<td></td>
</tr>
<tr>
<td>Control Group</td>
<td>2.025</td>
<td>2.000</td>
<td>-0.025</td>
<td></td>
</tr>
</tbody>
</table>

after studying the same course materials expect to take different forms of quizzes. The experimental group students who took the quizzes built upon our tool had built more confidence with only a short period of study with our tool.
3.4.3 Assignment Evaluation

In addition to all the student self-reported answers, we as instructors also want to see more concrete differences between the two groups of students reflected in either course assignments or exams. Here in our course, at the end of the first week, we set one coding assignment that requires students to practice what they’ve learned in the first week and hope for a better improvement among students in the experimental group.

![Difficulty Level of Circles Assignment](image)

**Figure 3.26**: Circles Assignment Difficulty Level. Control group average score: 0.8; Experimental group average score: 1.04

In the post-study survey, students in both groups reported their time spent on the coding assignment and their own rating on the difficulty level of it. As shown in Figure 3.26, experimental group student seem to consider the assignment a bit harder on average compared with control group. Similar readout can be seen in Figure 3.27, where control group student spent relatively less time on the assignment. One reason
64

Figure 3.27: Cicles Assignment Work Time. Control group average score: 1.875; Experimental group average score: 2.4

for this difference might be that the control group students were initially stronger than experimental group.

However, the scores of Cicles assignment says differently. Among all 27 students in the experimental group, 25 students got full score 80 (92.6%) and the average score is 79.48; 36 out of 40 students in the control group got the full score (90.0%) and a lower average score of 79.3. This result shows that the experimental outperformed the control group even if they initially had a weaker background. Our tool added to the improvement of performance among experimental group.

3.4.4 Summary

Table 3.5 summarizes our study results. Due to the pure randomized split of class, in this study, the experimental group has relatively lower average study year and less previous Computer Science knowledge compared with the control group. Their
Table 3.5: Summary Comparison of Two Study Groups

<table>
<thead>
<tr>
<th>Questions</th>
<th>Survey Time</th>
<th>Study Groups</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Experimental</td>
</tr>
<tr>
<td>1 Average Study Year</td>
<td>Before</td>
<td>0.590</td>
</tr>
<tr>
<td>2 Previous CS Knowledge</td>
<td>Before</td>
<td>0.890</td>
</tr>
<tr>
<td>3 Understanding of Code Execution</td>
<td>Before</td>
<td>0.590</td>
</tr>
<tr>
<td></td>
<td>After</td>
<td>1.930</td>
</tr>
<tr>
<td></td>
<td>Delta</td>
<td>+1.340</td>
</tr>
<tr>
<td>4 Confidence on Completing Course</td>
<td>Before</td>
<td>1.556</td>
</tr>
<tr>
<td></td>
<td>After</td>
<td>1.926</td>
</tr>
<tr>
<td></td>
<td>Delta</td>
<td>+0.370</td>
</tr>
<tr>
<td>5 Circles Assignment Difficulty Level</td>
<td>After</td>
<td>1.040</td>
</tr>
<tr>
<td>6 Time Spent on Circle Assignment</td>
<td>After</td>
<td>2.400</td>
</tr>
<tr>
<td>7 Average Scores on Circle Assignment</td>
<td>—</td>
<td>79.480</td>
</tr>
<tr>
<td>8 Ratings of Usefulness of Quizzes</td>
<td>After</td>
<td>1.704</td>
</tr>
<tr>
<td>9 Interest in Majoring CS</td>
<td>Before</td>
<td>1.074</td>
</tr>
<tr>
<td></td>
<td>After</td>
<td>0.852</td>
</tr>
<tr>
<td></td>
<td>Delta</td>
<td>-0.222</td>
</tr>
</tbody>
</table>

average numerical score on those two items were both smaller than the control group.

The student background difference explains that experimental group students have a lower level of understanding of code execution and lower confidence on completing the course before the study. However, we were glad to see a big improvement happened among experimental group students. Their confidence and understanding level both
had a large increase, catching up with and even surpassing the stronger control group.

Not surprisingly, the weaker experimental group took longer time working on their first coding assignment Circles, and considered the assignment a bit more difficult than the control group’s rating. However, the experimental students made greater progress during the first two-week study and achieved even higher average score than the control group in the coding assignment.

These results all indicate that the quizzes built upon our tool can offer our students better explanation and exercise to help them quickly learn Computer Science basics and programming concepts.

Other than the previously mentioned results, we also notice that the average rating of the usefulness of the quizzes used in the experimental group is lower than the regular quizzes used in the control group. Students’ comments to Question 7 in our tool in post-study survey shed light on the outcome. Students expressed various positive feedback, such as “very interesting way of learning”, “enjoyed the format”, “useful tool... allow me to think logically that the functions would be working”, “more interactive in terms of learning reading and understanding code”, “helpful and helps me understand the coding better” and so on. In the meanwhile, students in the experimental group also pointed out some issues that may cause them to underrate the tool. Most common issues include “tool instructions were very unclear”, “… didn’t seem to cover everything in the lectures”, and “questions could be adjusted to be more reflective of the programming we are asked to do in projects like the Circles Module”.

In general, students seemed to like the tool and proved the effectiveness of our tool in this preliminary study.
3.5 Conclusion

Programming is a fundamental part of Computer Science education. It is critical to students' success since students will need to program throughout their entire Computer Science study and in their future career life as well. However, programming is somewhat complex and is a combination of a set of skills like coding, code inspection, debugging, testing and so on. All these skills form a knowledge hierarchy, and the novice programmers need to master each of the skills in a proper order to minimize the cognitive burden and maximize the learning effect. For example, writing code should come after reading code; testing should be taught after writing code and so on. And many research works have shown that code reading is the most fundamental skill that drives all other further studies. After all, we cannot expect a student to write code and debug code even before he or she can read and understand what a program does.

Therefore, most computer science curricula heavily rely on the introductory programming courses or the CS1 courses to build up students’ various coding skills. However, the failure and drop out rate in CS1 remains high although Computer Science educators have been working on proposing new methodologies and building new tools to help students. Even though the tools have proved their effectiveness or positive effects in CS1 courses. Students still consider learning programming as difficult. Due to limited course hours, instructors often move fast and ask students to work on more complex task even before students have fully understood the basic concepts. Therefore, students suffer from the heavy cognitive burden and poor performance.

In this work, we try to teach students programming following the cognitive rules and start with the most fundamental skill – understand basic program concepts and code execution. We build a web-based educational tool that provides easy access to
a large number of students and allows them to perform self-paced learning. Our tool illustrates the code execution state using graphical visualization and requires students to perform mental simulations to mimic code execution in real code runtime system. To ensure students make steady progress, we build exercise problems for every single basic program concepts we want to cover in CS1 courses. Our tool can provide instant feedback and give students hints to help them understand the concepts. Moreover, our tool is not limited by the introductory programming course hours. Students can work on essentially the same problem yet with different initial states for theoretically unlimited times until they successfully understand those concepts. Studies in our own CS1 course shows that our tool can better improve students’ code reading, code tracing ability compared with traditional exercises and quizzes.
Chapter 4

Improving Students’ Test Cases Generation Skills

Testing is an important, time-consuming, and often difficult part of the software development process. It is therefore critical to introduce testing early in the computer science curricula and to provide students with frequent opportunities for practice and feedback. Computer Science educators find it is difficult for instructors to provide helpful feedback to help students learn the skills to generate high-quality test cases to test against their own code. The difficulties are two fold. Firstly, instructors have limited time to provide consistent, high-quality feedback to a large number of students. Also, it is difficult for instructors to correct students’ test cases or explain the deficiency of their test cases without concrete example programs. It is often too abstract for students to understand the problems with their own test cases. There were two ways to deal with this issue: 1) instructor manually generate feedback based on students’ programs to give them personalized feedback. This method is usually the most helpful methods for students. But this requires instructors to spend a lot of time with every student to provide personalized feedback; 2) Computer Science educators have been working on use machine-assisted method to help evaluate students test cases and provide feedback based on students’ programs. However, the machine assistant lacks the intelligence to predict various pitfalls and errors in a testing target program. Therefore, the machine cannot provide meaningful, concrete example programs that were not covered by student constructed test set.

To overcome the aforementioned challenges, here we present an automated system
to help introductory students learn how to test software. Students submit test cases to the system, which uses a large corpus of buggy programs to evaluate these test cases. In addition to gauging the quality of the test cases, the system immediately presents students with feedback in the form of buggy programs that nonetheless pass their tests. This enables students to understand why their test cases are deficient and give them a starting point for improvement. The system has proven effective in an introductory class: students that trained using the system were later able to write better test cases even without any feedback than those who were not. Further, students reported additional benefits such as improved ability to read code written by others and to understand multiple approaches to the same problem.

4.1 Background and Related Works

Testing is an important part of software development that can take up to half of the total development time [34]. Thus, it is critical to teach students how to test software [41, 8, 134, 39]. Computer science education focuses primarily on software development including topics such as languages, tools, and design principles and not on testing. This is particularly true at the introductory level.

Further, many students do not want to test their code, often viewing it as an unpleasant afterthought [135]. While students often derive a great deal of satisfaction from seeing a program that they wrote to solve a problem, they do not derive that same sense of satisfaction from exposing flaws in their own programs. However, there is evidence that learning to test helps students to write better code and to do so more quickly [78, 10]. Therefore, it is crucial to design and develop strategies to expose students to testing early in their computer science education [17, 3].

Therefore, we built an automated and interactive system to help students learn
how to write better test cases. The system was evaluated through a study that was conducted in an introductory computer science class. In our system, students are given a specification for a program and asked to submit a test suite for that program. Note that they do not need to write the program; rather, they must think about what inputs would stress implementation of the given specification. After submitting their tests and receiving feedback from the system, students can iteratively improve their tests and resubmit. We believe that several key elements of the system that contribute to its effectiveness.

First, students are not testing their own code. When testing their own code, students are less motivated to find bugs, as bugs expose their own failure to develop a correct program [38, 136]. When there is no personal investment in the code being tested, students are able to view testing as a rewarding activity and are driven to find as many bugs as possible. Second, the submitted tests are evaluated based on the number of buggy programs they detect from a large corpus of buggy programs, not their code coverage. This approach is motivated by the fact that techniques such as mutation analysis and all-pairs testing have been shown to be more effective than code coverage at identifying weak test sets [46, 77], and thus give students a far more accurate view of the quality of their test cases.

Finally, and perhaps most importantly, the system provides immediate feedback that not only tells students how well their tests perform but also gives clear examples of buggy implementations that their tests do not detect. It is often difficult for a student to reason about all possible edge cases in a program. Our system provides multiple example implementations with similar bugs, allowing students to understand why their test suite is not comprehensive by seeing the type(s) of bugs that their tests do not catch. This feedback not only helps students to develop better test cases,
but also gives them practice reading, understanding, and debugging code written by others. Further, it has the added benefit of exposing students to a diverse set of solution strategies to the same problem.

The system has proven to be successful at teaching introductory students how to write better test cases. After using the system, students were able to think about testing in a more comprehensive way and to develop better test cases without continuing to need to see example implementations. With no other training, students who used the system were able to score nearly a standard deviation higher on a testing assessment than students who had not a statistically significant result ($p < 0.0001$). Furthermore, over 73% of the students reported that the system improved their ability to write comprehensive test cases, to read code written by other people, and to understand multiple approaches to the same problem.

Much of the past work on teaching software testing has focused on broad structural changes to place greater emphasis on testing within computer science curricula [41, 8, 134, 78, 17, 39], touting the importance of integrating testing into the curriculum early and often. Such work is largely orthogonal to the methodological techniques presented in this work. Several other papers have proposed highly formal approaches to teaching software testing in upper-level courses which require devoting roughly half of the instructional time to teaching software testing [137, 138, 39]. While appropriate in an advanced context, these approaches are too complex for an introductory course where testing is not the primary focus.

Of the past work that has investigated techniques for integrating testing into introductory courses, the most closely-related work focuses on mutation analysis and all-pairs testing. DeMillo et al. helped popularize mutation testing [82], in which students simultaneously submit an implementation and set of test cases; the sub-
mitted implementation is mutated in various ways, and the mutants are used to determine how effective the test suite is as finding bugs. However, their approach does not assess students consistently, as the mutants presented to each student are highly personalized.

More recently, Aaltonen et al. demonstrated the superior ability of mutation analysis, as compared to code coverage, to identify weak test suites [46]. Like De-Millo et al., they used the students own implementation to generate mutants on the fly. However, they noted a flaw in their approach: students could exploit the grading system by introducing dead code into their solution, which would result in an artificially-inflated score. In contrast to our interactive framework, both Aaltonen et al. and DeMillo et al. performed mutation analysis after-the-fact to evaluate students final submissions. Another popular approach to testing in introductory courses is pairwise testing, which requires that students submit both an implementation and a test suite, and scores the test suite by running it on one or more implementation(s): traditionally, submission(s) of one [136] or more [139, 76, 140, 141] classmates. Such approaches require students to submit a final test suite “blindly” with no opportunity to interact with the implementation(s) on which their tests will be evaluated. Similar work has used alternate implementations including an instructor-provided reference implementation [79], or even a corpus of instructor-provided buggy implementations [142], posing fewer challenges to interactivity. Still, this past work largely required blind submissions and after-the-fact evaluation.

Recent work has improved pairwise testing by proving the feasibility of applying it synchronously, enabling students to see their scores progress as they develop their test cases [80]. This work required students to develop implementations and test suites simultaneously and built up a corpus of student implementations to run the
tests against as students submitted their work. However, this work faces several challenges. First, as the corpus of student implementations grows, the score associated with a given test suite will decrease, which may frustrate students. Additionally, the quality of the evaluation relies on the diversity of the student submissions, which is unpredictable. Last, the paper does not indicate that students receive any feedback beyond the score to help them complete the exercises, nor does it provide any evaluation of the effectiveness of the tool in improving the students testing capabilities.

In contrast, our system emits consistent, deterministic scores for a given test suite. Further, it ensures a diverse and thorough pool on which to evaluate students test suites by using a pre-existing corpus of programs that melds instructor-provided solutions, student solutions from prior iterations of the course, and prefabricated mutants. Additionally, we have experimentally proven the success of our system by performing a thorough evaluation its effectiveness within a controlled experimental environment.

4.2 Tool Design Overview

Many studies have cited limited time (both preparation time and instruction time) as one of the major barriers to teaching software testing [10, 143]. This motivates the use of an automated system to help students develop testing abilities. An even larger body of work demonstrates the value of active learning [23, 24, 144]. To balance these concerns, we created a system that evaluates students test suites immediately and automatically but fosters interactive learning by creating a constant cycle of formative feedback. This system was implemented for Python programs, but the pedagogical principles underlying it transcends any particular language.

At a high level, students are given a natural language specification for a function
and tasked with developing a test suite: a list of test cases, where each test case is a tuple of inputs that adheres to the specification. As students develop tests, they can submit them to the system at any time and receive feedback. This feedback is designed not only to evaluate the quality of the submitted tests but also to help students recognize subsets of the input domain that their tests do not cover. They can then iteratively improve their tests based on this feedback until they are satisfied with their results. In greater detail, the system runs the submitted test suite against a corpus of incorrect implementations of the function and provides the student with two pieces of feedback. First, they receive a score based on the number of incorrect implementations that were caught by the submitted test suite, i.e., that failed one or more test. Second, they are shown the source code for one or more incorrect programs that were not caught by their tests. Specifically, the system outputs up to three programs that share the same test signature, i.e., that exhibit the same set of bugs. Limiting the feedback to a single test signature scaffolds the students progress without immediately revealing all uncaught bugs. At the same time, providing multiple examples of implementations that exhibit that single signature exercises students pattern-matching skills, facilitating the process of identifying weaknesses in their current test suite.

As Figure 4.1 shows, our system consists of four key phases: creating a corpus of

\[ \text{Figure 4.1 : System Infrastructure} \]
implementations, identifying bugs within these implementations, developing a scaffolded progression through these implementations, and an interface through which students can interact with the scaffolded exercises as they develop their test suites.

4.3 Infrastructure

4.3.1 Solution Corpus Constructor

There are many viable means of building a corpus of implementations; we chose to combine two. First, we extracted student solutions $S$ from CodeSkulptor [27], an online interactive development environment (IDE) that automatically stores version history as students develop their code. This IDE has been used by many students across multiple MOOCs, and physically colocated classes, and thus has amassed a large corpus of student solutions to the programming exercises posed by these courses. By mining implementations directly from the IDEs version history rather than using final submissions, we capture a broader and thus more interesting for testing spectrum of the buggy and incomplete solutions.

| Functions  | $|S|$ | $|M|$ | $|A|$ | $|SE\ set|$ |
|------------|-----|-----|-----|-------|
| blackjack3 | 156 | 110 | 266 | 71    |
| format     | 2295| 235 | 2530| 76    |
| stringtime | 115 | 263 | 378 | 97    |

$|S|$: number of student solutions; $|M|$: number of mutant solutions; $|A|$: total number of solutions combining $S$ and $M$; $|SE\ set|$: number of solution sets of equivalent testing signature;.
Second, we ran mutpy \cite{81}, a mutation testing tool, on correct solutions to introduce a set of additional faulty implementations $\mathcal{M}$, giving us an aggregate set of implementations $\mathcal{A} = \mathcal{S} \cup \mathcal{M}$. As a supplement to $\mathcal{S}$, $\mathcal{M}$ serves two purposes. First, it introduces new test signatures for students to face as they develop their tests. Second, it increases the number of implementations with the same test signature, allowing the system to provide students with more detailed feedback for debugging. $\mathcal{M}$ is also effective in the absence of $\mathcal{S}$, rendering our system viable for small classes, new classes, and other scenarios in which a large corpus of student implementations may not yet exist. Table 4.1 shows the corpus sizes and number of unique signatures for the training functions (blackjack3, format) and the test function (stringtime) used in the evaluation.

4.3.2 Bug Identifier

Identifying bugs within $\mathcal{A}$ is necessary in order to identify unique test signatures and choose which programs should be displayed as feedback. For this, we used FEAT \cite{145}, an existing tool for test case generation and automated programming assessment. FEAT begins by creating an expansive “base test set through a combination of exhaustive and random generation. For the functions used in our experiment, the input domains were sufficiently constrained to use fully-exhaustive base test sets. FEAT outputs a mapping $\mathcal{D}$ associating each $\alpha \in \mathcal{A}$ with the set of test cases that it failed on. This set of failed tests is its test signature; implementations that share the same test signature typically have the same or very similar bugs.

4.3.3 Progression Scheduler

For the third component of our system, we extended FEAT with a new module that classifies and sequences programs based on their signatures. This module first creates
a mapping of each signature to the set of programs with that signature ("signature-equivalent or SE sets). It eliminates the empty test signature, as this represents programs that have no bugs. Next, it sequences the SE sets in order of descending signature size. As students work through the testing exercise, they will be provided with feedback in this order. The intuition for this is that it will be easier for students to create test suites that catch programs that fail on a larger number of cases than on those that fail in only one or two specific scenarios. Each signature-equivalent set may be arbitrarily large. However, presenting students with a large number of programs may be overwhelming. So, the module selects a maximum of three programs to represent each signature in the feedback provided to the students. To choose these three programs, it uses the radon package [1] to compute the McCabe cyclomatic complexity metric [146] for each program. The module then selects those programs that have the lowest complexity, with the goal of making the feedback maximally approachable for the students. Thus, at the end of this process, the module outputs a list of sets of programs, where each element in the outer list is a signature-equivalent set of size at most three.

4.3.4 Student Interface

To enable students to interact with these signature-equivalent sets, we used our existing auto-grading system, which requires instructors to upload a file defining how student submissions are processed and scored. We created a template file that imports the submitted test suite, runs a validation function to ensure that it meets the specification, and then runs each of the submitted tests on one representative program from each signature-equivalent set. During this process, it makes a note of the first signature that the students tests fail to catch, as that signature will be used
The `blackjack3()` function takes as its input three cards, each of which is represented as a single character from the string "23456789TJQKA". For instance, one valid tuple of inputs might be ("A", "K", "5"). In blackjack, number cards (including "T", which represents 10) are worth their value, and face cards ("JQK") are worth 10. Aces can be worth either 1 or 11; the choice between these is made such that the value of the hand is as high as possible without going over 21. (Note that there are some cases in which it is impossible to stay under 21.) Returning to the example of ("A", "K", "5"), the expected output would be 16. Your input file should contain a single Python definition:

- A list `TEST_CASES` containing at most 12 test cases for the function `blackjack3()`.
- Each test case in this list should be a list of tuples of length three whose entries are characters in "23456789TJQKA".

Figure 4.2 : Sample Function Specification

for feedback. It then computes a score based on the proportion of signatures caught. Last, this file generates feedback: the score and the source code of the program(s) in the first SE set that the submitted tests failed to catch. This scaffolds the exercise such that students receive some direction regarding which cases they're missing, but are not immediately given full white-box knowledge. For each function to be tested, the template need only be modified to import the correct SE sets and to provide a specification for the function being tested, a reference solution, and a validation function for the test suite. Optionally, the grading scale may also be customized.

### 4.4 Methodology

We evaluated the framework described in Section 3 within the fall 2016 iteration of an introductory computer science course that teaches students how to solve problems
in a computational way and how to implement their solutions in Python. This course has no prerequisites and is taken by a mix of majors and non-majors.

We ran a study consisting of a training exercise, an assessment, and two surveys. Students were randomly assigned to a control or experimental group. Both
groups completed the assessment simultaneously; however, the experimental group completed the training exercises beforehand, while the control group did not complete the training until afterward. Training the control group after-the-fact was not necessary for the study, but preserved fairness in workload and ensured that both groups reaped the benefits of the training. Neither group received any other training in testing prior to the assessment. Due to random assignment, we would expect both groups to perform equally on the assessment, all else being equal. Thus, since the only difference between groups was whether they had completed our training, inter-group differences on the assessment are likely due to that training. Last, both groups completed surveys before the training and after the test, in which they reported their level of confidence in their software testing abilities.

The training exercise was a mandatory homework assignment in which students used the framework described in Section 3 to develop test cases for two training functions. Students were given five days in which to complete these training exercises. The assessment was a mandatory in-class assignment but was not counted towards the students course grades for the sake of fairness. On the assessment, students were given a natural language specification for a function and asked to construct a list of at most 12 test cases for this function. To eliminate any potential advantages due to familiarity with the interface discussed in Section 3, the assessment was completed by hand. Students were given 15 minutes in which to complete the assessment. They were also asked to write down their partial solutions after five and ten minutes to provide greater insight into their process as they progressed through the exercise.

The pre-survey asked the students about their attitudes toward testing and confidence in their testing abilities. The post-survey asked the same questions so that the delta might show the effectiveness of the training exercises. However, we hy-
pothesized that students might not recognize their lack of testing capabilities until after they completed the training assignment, so the post-survey also included explicit questions about the effectiveness of the training.

4.5 Application and Evaluation

In fall 2016, 196 students were enrolled in the course in which we evaluated our system. 141 of these students consented to participate in the study. Students were randomly assigned to the control and experimental groups; of the consenting students, 70 were assigned to the control group and 71 to the experimental group. Due to absences, 134 students ultimately took the assessment, of which 68 belonged to the control group and 66 to the experimental group. We ran this study approximately halfway through the semester, at which point students had gained basic familiarity with Python and had likely written some test cases on their own, but had received no formal instruction on software testing within the course.

We used three pieces of data to assess the effectiveness of the training exercises. First, we compared the distributions of assessment scores of the experimental (trained) and control (untrained) groups. Second, we compared the students self-reported confidence in their abilities before and after completing the training exercises. Last, we examined the students perceptions of the effectiveness of the training exercises.

Assessments were scored out of 50, where the score was proportional to the number of unique signatures caught; Table 4.2 presents the summary statistics for each group, and Figure 4.4 shows the full distribution of scores. While both groups scores were roughly normally distributed, the trained group outperformed the untrained group by nearly a full standard deviation, with an average score of 36.53 as compared to
Table 4.2: Assessment: Summary Statistics

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Average</th>
<th>Max</th>
<th>Stdev</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Untrained</td>
<td>0</td>
<td>30.65</td>
<td>43.2</td>
<td>7.81</td>
<td>68</td>
</tr>
<tr>
<td>Trained</td>
<td>17.7</td>
<td>36.53</td>
<td>50</td>
<td>6.34</td>
<td>66</td>
</tr>
<tr>
<td>Aggregate</td>
<td>0</td>
<td>33.54</td>
<td>50</td>
<td>7.69</td>
<td>134</td>
</tr>
</tbody>
</table>

Figure 4.4: Assessment: Final Scores

30.65 for the untrained group. A one-tailed unpaired \( t \)-test yielded a \( p \)-value of \(< 0.0001\), which is regarded as statistically significant. Note that students received no feedback during assessment; the fact that the trained students performed better on the test even in the absence of feedback indicates that, while the feedback during training likely helped them to improve, they did not become dependent on it for future success in writing tests.

Interestingly, examining the scores from the five- and ten-minute checkpoints revealed that the gap between groups increased with time. As Table 4 shows, the trained group averaged a 5.00 point advantage at five minutes; by ten minutes, this had grown to 5.66 points, and by fifteen it had expanded to 5.87 points. One pos-
sible explanation is that the untrained students more quickly approached a point of diminishing returns, while the trained students were able to continue reasoning productively about the input domain.

Further, students reported increased confidence on a variety of testing-related skills. For each statement in Table 3, students rated their confidence using a four-point Likert scale (Strongly Disagree, Disagree, Agree, Strongly Agree) at the beginning and end of the study. In every case, the percentage of students who responded with Agree or Strongly Agree increased. Most notably, the percentage of students who felt confident in their ability to write comprehensive test cases increased from 34.5% to 55.2%, those who felt confident in their ability to test their own code increased from 51.4% to 71.6%, and those who felt confident in their ability to write comprehensive test cases for code written by others more than doubled, from 28.6% to 62.7%. Students also indicated notably higher confidence in their ability to reason about the input domain.

Surprisingly, a fairly small percentage of students felt improved confidence in their ability to write test cases without actually seeing the code they were testing. However, the results of the assessment, which asked students to do exactly that — develop a black-box test suite — indicates that this the student self-reported lack of confidence in post-study survey was unfounded. When asked to provide explicit feedback on the training exercises using the same four-point Likert scale, the students strongly corroborated their effectiveness. Over 73% of students agreed that the exercises improved their ability to write comprehensive test cases. Moreover, students reported auxiliary benefits of the training exercises. Over 82% of students felt increased confidence in their ability to read code written by others, and over 91% of students felt better equipped to understand multiple approaches to the same problem both hugely
valuable skills in software development.

Table 4.3: Implicit Feedback

<table>
<thead>
<tr>
<th>“I am confident in my ability to...”</th>
<th>% Agree (Pre-survey)</th>
<th>% Agree (Post-survey)</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identify distinct logical categories of input (corresponding to unique code paths) for a particular problem</td>
<td>59.3%</td>
<td>77.6%</td>
<td>+18.3%</td>
</tr>
<tr>
<td>Write comprehensive test cases</td>
<td>34.5%</td>
<td>55.2%</td>
<td>+20.7%</td>
</tr>
<tr>
<td>Write comprehensive test cases for my own code</td>
<td>51.4%</td>
<td>71.6%</td>
<td>+20.2%</td>
</tr>
<tr>
<td>Write comprehensive test cases for code written by other people</td>
<td>28.6%</td>
<td>62.7%</td>
<td>+34.1%</td>
</tr>
<tr>
<td>Write comprehensive test cases based on a description of a function, without seeing the implementation (code)</td>
<td>32.9%</td>
<td>36.6%</td>
<td>+3.7%</td>
</tr>
</tbody>
</table>

Table 4.4: Assessment: Intermediate Results

<table>
<thead>
<tr>
<th></th>
<th>Average (5 minutes)</th>
<th>Average (10 minutes)</th>
<th>Average (Final)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Untrained</td>
<td>21.80</td>
<td>26.90</td>
<td>30.65</td>
</tr>
<tr>
<td>Trained</td>
<td>26.80</td>
<td>32.55</td>
<td>36.53</td>
</tr>
<tr>
<td>Delta</td>
<td>+5.00</td>
<td>+5.66</td>
<td>+5.87</td>
</tr>
</tbody>
</table>
Table 4.5 : Explicit Feedback

<table>
<thead>
<tr>
<th>“The training exercises improved my ability to...”</th>
<th>% Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write comprehensive test cases</td>
<td>73.1%</td>
</tr>
<tr>
<td>Read code written by other people</td>
<td>82.1%</td>
</tr>
<tr>
<td>Understand multiple approaches to the same problem</td>
<td>91.0%</td>
</tr>
</tbody>
</table>

4.6 Conclusion

This chapter has presented an interactive system for helping students learn to develop better test suites. The system was evaluated within an introductory class, and has been shown to improve students testing abilities by a statistically significant degree. Given the benefits the students derived from the system, we plan to continue to use it in our introductory class in the future to teach testing.

Three key features contribute to the systems effectiveness. First, students are not testing their own code. This allows them to focus on developing test cases, rather than worrying about bugs in their own code. Second, the submitted test suites are evaluated based on how many buggy programs they detect from a large corpus of programs. This gives a more objective measure of test quality than many other metrics such as code coverage. Finally, students are given instantaneous feedback on their submitted test cases in the form of programs that contain bugs which are undetected by their tests. This enables students to reason about the quality of their test cases and to understand why they are not sufficient.

Students who trained with our system were able to construct better test cases than those who had not: on an assessment that required students to develop black-box tests for a natural language specification, the average performance of the trained
group was nearly a full standard deviation higher than that of the untrained group. This indicates that our system teaches students to think about testing in a more comprehensive way without continuing to need to see example implementations. Further, the students recognized the improvements in their abilities, reporting increased confidence in their ability to write comprehensive test cases.

The nature of the feedback system also provides additional benefits. Examining buggy programs not only helps students to develop better test cases, but also gives them practice reading, understanding, and debugging code written by others. Similarly, being exposed to these buggy programs helps students to reason about a variety of different solution strategies to the same problem. These are both extremely valuable skills for software developers to have.

Testing is a critical part of the software development process. Despite this, most computer science curricula devote little time to teaching students how to test. Therefore, it is essential to develop new systems and techniques for integrating testing into the curriculum. This work is one such system and evaluated it within an introductory course, demonstrating how the combination of automation and interactive feedback can quickly and effectively yield striking improvements in student testing capabilities.
Chapter 5

Generating Test Cases for Grading Students’ Programs

The ultimate goal of testing a program is to discover faults that cause the program failure rather than proving the correctness of the program. Some even argue that the program correctness can never be demonstrated through software testing [147]. The reasons include the complexity and the size of the input domain, the number of possible paths through the program, and wrong or incomplete specifications. Therefore, testing a program has never been an easy task for software programmers, which requires rigorous training and rich experiences.

Big software companies like Google and Microsoft etc. have established strict development process that requires their software engineers to write high-quality test cases to unit test their newly added functions. All test cases built along the development process accumulated to form a big test suite for future regression testing [148] providing confidence that new changes do not adversely affect the existing features of the software. The completeness and the test coverage of test cases directly determine the quality of programs. This practice has been proved as a big success over the years in business and has been widely adopted in the software industry.

To help our students get familiar with similar work process, we require our students to write programs according to program specifications and, in the meanwhile, write their own test cases to test against their own programs. Manually grading both students’ program implementation as well as the correctness and completeness of test
cases is usually very tedious and time-consuming. Moreover, human experts may not anticipate the numerous ways of implementations so their own gold-standard test suite might not be complete enough either in the sense of identifying defective implementations.

In this thesis, we built an automated grading system to solve this problem that can automatically generate high-quality test suite with high-completeness to test students’ submitted programs so as to indirectly assess the completeness of their own test suite. Section 1 introduces the background of the work and related literature review. Section 2 gives a quick overview of the core part of the automated grading system — the test cases machine generator as well as its features. Section 3 and Section 4 describe the details of the two major components — base test case generator and concise test set generator respectively. We will introduce the usage and evaluation results of the proposed system in Section 5. Section 6 concludes the entire chapter.

5.1 Background and Related Works

Traditional program testing relies on test cases built by either human experts or using machine auto-generation methods. Existing test case generation methods can be categorized into white-box testing and black-box testing. White-box test cases are derived from the internal structure of the program [149]. Black-box test cases are derived from the program specification [150]. However, in both cases, it is difficult to achieve complete automation of the test case design [151].

Human experts are usually good at writing test cases for an existing piece of the program based on their knowledge and experiences to construct test cases for white-box testing. Therefore, this method works well in the test driven development process in the software industry as adopted in the big software companies. However,
this human method can not scale when it comes to testing large and complex systems hence usually need computer-assisted testing.

Computer-based test case generation methods can be categorized into three major categories: random-based, search-based, and data mining-based methods. Random-based methods randomly generate a large number of tests within a constrained search space [47]. Given an existing piece of code, techniques like mutation-driven selection [152] can be applied to random-based methods to eliminate unimportant test cases. However, future submissions are hard to predict, and thus cannot guide test selection in this way.

Search-based methods [153] use more advanced algorithms such as genetic algorithms [154] and particle swarm optimization [65] to directly search for high quality test cases. However, these methods are complex and computation intensive. Moreover, they are not easily generalizable, requiring problem-specific fitness function selection and tuning.

Data mining-based approaches have been proposed to reduce the number of test cases without losing coverage [155, 156, 48] by identifying hidden input-output (I/O) relations. However, these methods are typically more complex, requiring a large number of training samples and significant tuning in order to achieve accurate predictions.

One major issue with all these machine test case generation methods is that those machine auto-generation methods are usually complex, problem-specific, and time-consuming. Moreover, the machine-based test case auto-generation methods also share another issue with human-based methods – target a single existing piece of software, and are typically used to validate software that is already believed to be largely correct. Based on the existing version of the program, various kinds of code coverage criteria have been proposed to be a measurement of test-suite quality and
can be used to guide test generation process, such as control-flow coverage criteria and data-flow coverage criteria. Different from control-flow coverage criteria that are based on the control-flow graph, data-flow coverage criteria are based on the definitions and use of variables. The control-flow coverage criteria contain statement coverage, condition coverage, path coverage and branch coverage [157].

However, in our case, based on the single version of programming specification, our students may come up with numerous ways to solve or fail the task. Since both human-based and machine-based test case generation approaches needs to be carefully tuned based on the actual code implementation to achieved high test coverage, they are usually incomplete and computationally prohibitive in reality. The same problem arises in many other Automated Programming Assessment (APA) systems that have been widely used in many areas to evaluate program correctness and efficiency.

Universities and companies have built automated systems for judging programming competitions, such as UVa [158], PC$^2$ [159], TopCoder [67], and Google Code Jam [160]. Educators have created LeetCode [69], Rosalind [70], and Project Euler [161] to teach programming skills and algorithms through problem-solving exercises. Recently, many massive open online courses (MOOCs) have relied heavily on automated assessment systems for grading programming assignments [65].

The effectiveness and efficiency of an APA system are largely determined by the quality of its test cases. Manually building test cases is tedious and time-consuming, and makes it difficult to guarantee a high level of error detection. Thus, researchers have put a great deal of effort into test case auto-generation to reduce human effort and increase test quality [75, 162, 148].

Here in this thesis, we propose a fast, simple data-driven method for generating high-quality test sets for a programming problem from an existing collection of
student solutions for that problem. This proposed system demonstrates the effectiveness of online programming course assessments. The experiments showed that, when applied to large collections of such programs, the method produces concise, human-understandable test sets that provide better coverage than test sets built by experts with rich teaching experience.

5.2 Tool Design Overview

The system we proposed in this chapter is a simple and easy-to-use tool-chain for creating high-quality test sets for automated assessment systems, Feedback and Evaluation via Auto-generated Tests (FEAT). It utilizes a collection of previously-submitted student solutions to guide the test generation process, under the assumption that no error is unique in a crowd-sourcing scenario [12]. The system consists of three modules (Figure 5.1): the base test set generator, the tester, and the concise test set generator.
The first module auto-generates a large pool of test cases for a particular programming assignment, based on an instructor-provided problem specification. This pool is referred to as the base test set. The second module uses the base test set to check the correctness of a training set of student solutions against an instructor-provided reference solution. The erroneous students’ solutions collected at this stage help us harness the most common program faults and misconceptions since the crowd-sourcing approach help us “harness their human intelligence and creativity” [12]. Finally, the third module uses the testing results to construct concise test sets that are small subsets of the base test set. These concise test sets are carefully chosen such that they detect every erroneous solution that could be found using the much larger base test set while providing the efficiency needed for use in an APA system.
This proposed system has three key contributions:

- **Base test set generation:** This tool uses a simple specification for the inputs to a programming problem and describes an algorithm for creating a large set of test cases based on this specification.

- **Concise test set generation:** Given a collection of previous solutions to a programming problem, this tool performs test reduction methods for selecting concise subsets of the base test set with the property that all programs that fail on the base test set also fail on the concise test sets. Two types of concise test sets are presented: approximately minimal and gradated.

- **Comparison with expert test sets:** Finally, this tool compares the test coverage of these autogenerated concise test sets to that of expert-generated test sets, showing that the concise test sets provide greater test coverage.

The automated grading system consists of three major components: 1) The frontend that directly interacts with students; 2) The backend that tests both students’ program and test cases; 3) The internal test case generation module.

We periodically run the test case generation module to update the internal test set when more student data is available. After that, students can submit both their programs and test sets through the tool frontend to our grading system. The backend first test the correctness of students’ program using their own test sets. The program and test sets will be directly returned to students if their program cannot pass their own test set. Otherwise, the system will use the internal test set to test the completeness of students’ programs and return the first failed test set to students as the feedback for them to improve the completeness of their own test set and fix their own program.
5.3 Base Test Set Generation

This section presents the first module of the FEAT toolchain: a generic, automated approach to generating an expansive set of test cases based on a simple yet powerful means of inductively defining the inputs of the function being tested. The resultant base test set $B$ serves as the pool of candidate tests from which the two concise test sets are ultimately drawn (see Section 5.5). Further, this section presents a method for assigning a complexity heuristic to each test case. These heuristics enable the construction of gradated test sets, designed to give students actionable feedback (see Section 5.5.2).

FEAT employs a hybrid approach to the base test set generation. A fully-exhaustive test set — i.e., all valid combinations of arguments — is guaranteed to catch all incorrect programs, but generating such a test set is often impossible and nearly always computationally prohibitive.

In contrast, generating a random test set is tractable, but has a high probability of missing edge cases. Thus, $B$ is the union of two disjoint test sets: an exhaustive set covering a manageable subset of the input domain, and a random set providing partial coverage of the remaining portion of the domain.

5.3.1 Specifying the Domain for Test Cases

To generate the base test set, the user must specify four pieces of information in a config file: parameter types, argument domains for both exhaustive and random generation, and the desired number of random test cases.

Valid parameter types include all built-in Python types, such as lists, dictionaries, strings, and integers, nested arbitrarily deeply. Further, types may be classes, specified as composites of built-in types. Argument domains bound the values that
FEAT will use as arguments for each parameter, and may be continuous or discrete.

FEAT also provides three optional means of further constraining the arguments, to facilitate testing functions with arbitrarily complex specifications. First, the config file language includes keywords, such as `sorted`, for constraining a single parameter. Second, dependencies may exist between parameters. The user can express such dependencies by defining and using variables; Section 5.3.4 presents an example of such a dependency.

Keywords and dependencies are lightweight, intuitive means of expressing constraints, but they do not always suffice. Thus, the user may provide a validation function with parameters identical to those of the function being tested. This function returns `True` if the arguments are valid, and `False` otherwise. FEAT uses this function to post-process its tentative base test set, discarding invalid cases.

### 5.3.2 Auto-Generating Test Cases

Base test set generation has two phases: exhaustive generation and random generation. Exhaustive generation operates as follows. First, for each parameter, FEAT recursively generates all possible arguments. During generation, FEAT maintains metadata for each argument: a set of values for each variable that are compatible with that argument. FEAT uses this metadata to prune the search tree when it encounters disjoint sets of compatible values for the same variable across nested layers of the argument.

Once FEAT has amassed a set of consistent arguments for each parameter, it takes the cross product of these sets to generate all possible test cases, again pruning the search upon incompatibility. Last, if a validation function was provided, it uses this function to filter the set of test cases.
Random generation is simpler. While the size of the random test set is less than the target, FEAT randomly selects a value for each variable. It then randomly selects each argument subject to these variable constraints. If the resultant test case is not yet in $B$, it is added to the random test set.

### 5.3.3 Assigning Test Case Complexity

After generating $B$, FEAT assigns a floating point complexity $C[t]$ to each individual test case $t$. Complexity is defined as $C[t] = \prod C[t_i]$, where $C[t_i]$ is the complexity of the $i^{th}$ argument and $C[t_i]$ is the product of the average complexity of each nested component. Component complexity is the length for sequences and value for primitives. By default, each component of each argument is incorporated into this heuristic. However, the user may optionally specify a subset of arguments and/or components to use.

As an example, consider the list $[1, 2, 3]$. The complexity of the list itself is its length (3.0), and the complexity of its contents is the average value (2.0), for a default complexity of 6.0. Imagine that the user declares the contents irrelevant; in this case, the complexity will be 3.0.

### 5.3.4 Configuration Example

In the dice game Yahtzee, a player rolls a set of dice and then holds some subset of the dice while re-rolling the remaining dice. Consider a function `expected_value` with three inputs — the dice held (a sorted tuple of integers), the number of sides on each die (an integer), and the number of dice to be re-rolled (an integer) — and computes the expected value after the roll. Naturally, the values of the held dice may not exceed the number of sides on the dice.
Figure 5.2 : Sample Config File

Figure 5.2 shows a sample config file for this function, with syntax slightly condensed for brevity. Parameter types are expressed under the [types] header using a combination of keywords (sorted) and Python types (tuple, int), with parentheses indicating nesting. For instance, sorted tuple (int) denotes a tuple of integers sorted in ascending order.

Exhaustive and random domains ([e domain] and [r domain]) for each parameter are expressed as inclusive ranges, representing lengths for sequences and values for primitive types. For instance, the first exhaustive domain (0--3 (1--m)) stipulates that the tuple of held dice has a length on [0, 3], where each element in the tuple is an integer on [1, m].

Dependencies between parameters — in this example, the fact that values representing rolled dice may not exceed the number of sides — are captured using variables. Consider the exhaustive case. The domain for the number of sides is m, defined under the [variables] header as [1, 6], and the upper bound on the values of the rolled dice is likewise m, ensuring that no die’s value exceeds the number of sides.

Finally, the [complexity] header specifies which features of the input define its
<table>
<thead>
<tr>
<th>Index</th>
<th>Module</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2048 Game</td>
<td>merge</td>
</tr>
<tr>
<td>2</td>
<td>Graph Theory</td>
<td>compute_in_degrees</td>
</tr>
<tr>
<td>3</td>
<td>Graph Theory</td>
<td>in_degree_distribution</td>
</tr>
<tr>
<td>4</td>
<td>Graph Theory</td>
<td>make_complete_graph</td>
</tr>
<tr>
<td>5</td>
<td>DNA Alignment</td>
<td>build_scoring_matrix</td>
</tr>
<tr>
<td>6</td>
<td>Yahtzee</td>
<td>expected_value</td>
</tr>
<tr>
<td>7</td>
<td>Yahtzee</td>
<td>gen_all_holds</td>
</tr>
<tr>
<td>8</td>
<td>Yahtzee</td>
<td>score</td>
</tr>
</tbody>
</table>

Table 5.1: Program Modules and Functions

Complexity. Here, True (False); True; True indicates that increasing the number of dice held, number of sides, and number of dice rolled increases the complexity of the test case, but changing the particular values of the held dice does not affect the complexity.

In this study, we run FEAT tool on eight functions extracted from eight different programming assignment modules from our online programming courses. Table 5.1 shows the names of the coding assignment module names and the actual functions we used in our study. We built config files for all of the eight functions. The generated base test set sizes for each problem are listed in 5.2.
<table>
<thead>
<tr>
<th>Problem Name</th>
<th>Exhaustive</th>
<th>Random</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge</td>
<td>781</td>
<td>219</td>
<td>1000</td>
</tr>
<tr>
<td>compute_in_degrees</td>
<td>285</td>
<td>215</td>
<td>500</td>
</tr>
<tr>
<td>in_degree_distribution</td>
<td>285</td>
<td>215</td>
<td>500</td>
</tr>
<tr>
<td>make_complete_graph</td>
<td>30</td>
<td>20</td>
<td>50</td>
</tr>
<tr>
<td>build_scoring_matrix</td>
<td>864</td>
<td>136</td>
<td>1000</td>
</tr>
<tr>
<td>expected_value</td>
<td>627</td>
<td>373</td>
<td>1000</td>
</tr>
<tr>
<td>gen_all_holds</td>
<td>462</td>
<td>538</td>
<td>1000</td>
</tr>
<tr>
<td>score</td>
<td>461</td>
<td>539</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 5.2: Machine Generated Base Test Set Sizes

5.4 Testing Student Solutions

Once the base test set has been created, student solutions can be tested. Tests are first to run on a reference solution to acquire the expected results, and then run on a corpus of student solutions, $S$, to ascertain correctness.

Recall that the goal of running the base test set on $S$ is not merely to check these particular solutions. Rather, these solutions serve as a training set to identify high-quality test cases — those that trip up many erroneous solutions — and to derive the concise test sets (see Section 5.5). The concise test sets can then be used to efficiently test future solutions.

The tester maintains a mapping $D$ of each solution, $S_i$, to the set of test cases that it failed on, $B_i$ (i.e., $B_i = D[S_i]$). This data enables selection of a concise subset
of $B$ without sacrificing coverage of any known incorrect solutions.

FEAT was used to generate and run base test sets for eight problems from four different programming assignments. Table 5.2 shows the inputs ($|B|$, $|S|$) and outputs (number of incorrect solutions identified, runtime) of the tester.

The number of student solutions tested ranged from 3639–73156; between 20–78% of those solutions proved incorrect. The size of the base test set was typically configured to 500–1000 test cases, with one exception: `generate_complete_graph`, which takes as its input an integer and generates a complete graph with that many nodes. In this case, the simplicity of the parameter types led to diminishing returns as the size of $B$ was increased.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Total Num</th>
<th>Incorrect Num</th>
<th>Runtime (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>merge</code></td>
<td>55523</td>
<td>39617</td>
<td>12:16</td>
</tr>
<tr>
<td><code>compute_in_degrees</code></td>
<td>5145</td>
<td>1577</td>
<td>0:16</td>
</tr>
<tr>
<td><code>in_degree_distribution</code></td>
<td>5109</td>
<td>2805</td>
<td>0:15</td>
</tr>
<tr>
<td><code>make_complete_graph</code></td>
<td>5120</td>
<td>1094</td>
<td>4:27</td>
</tr>
<tr>
<td><code>build_scoring_matrix</code></td>
<td>3374</td>
<td>1090</td>
<td>12:59</td>
</tr>
<tr>
<td><code>expected_value</code></td>
<td>65217</td>
<td>37733</td>
<td>14:52</td>
</tr>
<tr>
<td><code>gen_all_holds</code></td>
<td>57758</td>
<td>35597</td>
<td>51:15</td>
</tr>
<tr>
<td><code>score</code></td>
<td>65652</td>
<td>26336</td>
<td>4:45</td>
</tr>
</tbody>
</table>

Using such an extensive training set leads to strong coverage, but demands trade-offs in time. Runtime varied greatly, influenced by $|B|$ and $|S|$; the minimum was 15 minutes, and the maximum was 51 hours. This non-trivial runtime motivates the use
of a data-driven approach like FEAT, as performing semantic analysis on each training solution would likely prove computationally prohibitive. Further, Section 5.6.2 will show that $|S|$ can be reduced substantially while maintaining over 95% of the original coverage.

5.5 Concise Test Set Generation

The base test set is designed to provide broad coverage but is ill-suited for direct use in APA systems as its large size would slow the feedback cycle. This section describes two algorithms for extracting concise test sets from $B$. The first algorithm constructs an approximately minimal test set $M$ with the property that every solution in $S$ which fails on some element in $B$ also fails on some element in $M$. The second algorithm generates a gradated test set $G$ that is similar in size to $M$, but favors lower complexity.

5.5.1 Approximately Minimal Test Sets

As stated in Section 5.4, the tester computes a subset $B_i$ of $B$ for which the solution $S_i$ disagrees with the reference solution. The goal of concise test set generation is to compute a subset $M$ of $B$ that contains at least one test case from each non-empty $B_i$. In other words, $M$ has the property that $B_i \cap M \neq \emptyset$ over all $B_i$.

This problem corresponds to the classical hitting set problem, which is known to be NP-hard. Fortunately, there exists a simple greedy methods utilizing GRE heuristics [148] for computing a hitting set whose size is guaranteed to be within $\log(|B|)$ of the optimal size.

FEAT maintains a family $F$ of the sets $B_i$ that is dynamically updated as the algorithm proceeds. The GRE heuristic is coverage, where the coverage of a test case
t with respect to F is defined as the number of sets in F that contain t. M is then constructed using a three-step iterative strategy:

1. Compute the test case t that has maximal coverage;
2. Add this test case t to M;
3. Remove those sets in F that contain t.

This process continues until F is empty. Since each entry in F corresponds to one student solution, this guarantees that M has the same coverage as B. The pseudo-code for this algorithm is as follows:

```
Algorithm 5.5.1: APPROXMINIMALTESTSET(D, B, S)

M ← ∅, F ← ∅
for each S_i ∈ S
do
  B_i ← D[S_i]
  F ← F ∪ {B_i}
while F ≠ ∅
do
  t ← arg max_{t∈B} |{B_i ∈ F | t ∈ B_i}|
  M ← M ∪ {t}
  F ← F \ {B_i ∈ F | t ∈ B_i}
return (M)
```

5.5.2 Gradated Complexity Test Sets

Since Algorithm 5.5.1 repeatedly chooses test cases with maximal coverage, the resultant M is significantly smaller than B. However, it has one major drawback: the test cases in M tend to have high complexity, which is inconsistent with good testing
Table 5.4: Test Set Size Comparison

<table>
<thead>
<tr>
<th>Function</th>
<th>Base Test Set</th>
<th>Concise Test Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Minimal</td>
</tr>
<tr>
<td>merge</td>
<td>1000</td>
<td>31</td>
</tr>
<tr>
<td>compute_in_degrees</td>
<td>500</td>
<td>4</td>
</tr>
<tr>
<td>in_degree_distribution</td>
<td>500</td>
<td>5</td>
</tr>
<tr>
<td>make_complete_graph</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>build_scoring_matrix</td>
<td>1000</td>
<td>2</td>
</tr>
<tr>
<td>expected_value</td>
<td>1000</td>
<td>12</td>
</tr>
<tr>
<td>gen_all_holds</td>
<td>1000</td>
<td>6</td>
</tr>
<tr>
<td>score</td>
<td>1000</td>
<td>9</td>
</tr>
</tbody>
</table>

practice. If used in an APA system, it may cause the system to report that a user failed a complex test case when a simpler example would be more valuable to the learning process.

One possible solution to this problem is to run the test cases in $M$ in order of their complexity to ensure that a user solution fails on the simplest test in $M$ first. However, this approach can fall victim to the situation where Algorithm 5.5.1 selects few simple tests.

A better solution is to balance the coverage of a test case versus its complexity when choosing a new test case. Algorithm 5.5.2 computes the gradated test set $G$ by assigning a score to each test case: the ratio of its current coverage to the square of its complexity. In each iteration, Algorithm 5.5.2 selects the test case $t$ with the highest score. Ties are broken by choosing the test case with lower complexity. After selecting $t$, the algorithm updates set family $F$ by removing those test cases sets that
have been covered by $t$. The pseudo-code below outlines the process:

\begin{algorithm}
\caption{GradatedTestSet($D, B, S, C$)}
$G \leftarrow \emptyset$, $F \leftarrow \emptyset$

\For{each $S_i \in S$} {
  $B_i \leftarrow D[S_i]$
  $F \leftarrow F \cup \{B_i\}$
}

\While{$F \neq \emptyset$} {
  $t \leftarrow \arg\max_{i \in B}(\{B_i \in F | t \in B_i\} / (C[t])^2)$
  $G \leftarrow G \cup \{t\}$
  $F \leftarrow F \setminus \{B_i \in F | t \in B_i\}$
}

\Return $(G)$
\end{algorithm}

Table 5.4 shows the size of $M$ and $G$ for the same eight problems introduced in Section 5.4; in each case, both concise test sets are substantially smaller than the corresponding base test set. While each $|G|$ is greater than or equal to the corresponding $|M|$, the gradated test sets benefit from gradual growth in complexity. Figure 5.3 compares the complexities of the tests in $M$ and $G$ for the problem \texttt{expected.value}. $M$ contains complex test cases with random ordering, shown as $M$-\texttt{Original} and sorted as $M$-\texttt{Sorted}. In contrast, $G$ achieves the same coverage as $M$ with notably less complexity.

The auto-generated gradated test sets were deployed in a programming MOOC. The APA system for that MOOC tests student’s programs against sorted tests cases in $G$ from easy to complex. The system provides a partial credit to erroneous programs based on which tests in $G$ that they fail, rather than making a binary correct/incorrect judgment. Following the test-driven development principle [45], the system also
returns the first — and therefore simplest — failed test case to the student, aiding the student in debugging their code. Students repeat this practice until they are satisfied with their scores, practicing their coding and debugging skills in the meanwhile.

### 5.6 Application and Evaluation

#### 5.6.1 Comparison with Expert Tests

The gradated concise test sets computed by Algorithm 5.5.2 were deployed in the second session of the authors’ programming MOOC. Student response was positive, with no reports of incorrect solutions passing the machine grader’s tests. Additionally, this section presents a more methodological analysis of the coverage of the concise test sets versus the instructor-created test sets used in the first session of the MOOC. All
student solutions for the eight problems studied in previous sections were analyzed.

Table 5.5 reports the results of comparing the coverage of the expert test sets and the auto-generated concise test sets. Recall that both $M$ and $G$ have the same coverage as their common base test set $B$. Thus, the “C” in the headings of this table simultaneously represents the results from $B$, $M$, and $G$. The four columns in the table indicate, respectively, the number of student solutions that pass both test sets, fail both test sets, pass the expert and fail the concise, and fail the expert and pass the concise.

Most importantly, note that the entries in Column 4 are very small — often zero — compared to those of the other columns. This reflects the fact that the concise test set caught almost all of the errors that the expert test caught. The expert test set occasionally caught a few incorrect solutions that were not identified by the concise test sets; however, this was rare, as this situation only occurs when a solution does not fail any of the tests in the base set. In this situation, the expert test set usually included tests that exploited problem knowledge that was not encoded in the configuration file provided to FEAT.

Another important observation is that, for many problems, the values in column three were large in comparison to column two. This indicates that the expert test set failed to detect a non-trivial fraction of the programs marked incorrect by the concise test set. For example, the expert test set for score allowed almost two-thirds of the solutions marked incorrect by the concise test set to pass. This indicates that the expert test set had substantial deficiencies, correlating with the student complaints from the first session.
<table>
<thead>
<tr>
<th>Function</th>
<th>√ E *</th>
<th>× E</th>
<th>√ E</th>
<th>× E</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge</td>
<td>15860</td>
<td>37776</td>
<td>1838</td>
<td>25</td>
</tr>
<tr>
<td>compute_in_degrees</td>
<td>3562</td>
<td>1555</td>
<td>21</td>
<td>6</td>
</tr>
<tr>
<td>in_degree_distribution</td>
<td>2301</td>
<td>2742</td>
<td>61</td>
<td>3</td>
</tr>
<tr>
<td>make_complete_graph</td>
<td>4025</td>
<td>865</td>
<td>229</td>
<td>0</td>
</tr>
<tr>
<td>build_scoring_matrix</td>
<td>2283</td>
<td>1030</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>expected_value</td>
<td>27256</td>
<td>34573</td>
<td>2923</td>
<td>3</td>
</tr>
<tr>
<td>gen_all_holds</td>
<td>11424</td>
<td>31576</td>
<td>669</td>
<td>0</td>
</tr>
<tr>
<td>score</td>
<td>39316</td>
<td>9320</td>
<td>17016</td>
<td>0</td>
</tr>
</tbody>
</table>

* Note: √ E means “correct on expert test set” and × C means “incorrect on concise test set” and so on so forth

### 5.6.2 Method Sensitivity Analysis

Since FEAT selects test cases based on their results on the training set, the error detection ability of the resultant concise test sets on future programs is driven by the number and quality of programs in this training set. Too few or too similar programs can bias the test selection, leading to false positive verdicts on future submissions. On the other hand, a large training set will slow the testing phase and may be difficult to come by. Thus, it is necessary to study the relationship between training set size and test set coverage in order to navigate these tradeoffs.

FEAT’s sensitivity to training set size was evaluated on the same eight problems analyzed in Sections 5.4 and 5.5. For each problem, 1000 solutions were randomly generated and evaluated.
selected to serve as the test pool $P$. From this pool, the percentage of programs randomly selected to serve as the training set $S$ was gradually increased from $0.78125\%$ – $100\%$. The resultant concise test sets were then used to evaluate all programs in $P$. To account for the randomness in the selection of $S$, this process was repeated five times for each problem. Figure 5.4 shows the average coverage for each training set size.

Intuitively, the percentage of incorrect programs in $P$ identified by FEAT’s concise test sets increases with $|S|$. More surprisingly, for all eight problems, using a mere
12.5% of the programs in $P$ achieved over 95% test coverage of $P$. In other words, the test coverage of the generated concise test sets does not increase linearly with respect to the number of training programs. Rather, it increases quickly when $|S|$ is small and then plateaus. This suggests that FEAT can generate concise test sets with good coverage even using a reasonably small number of training programs.

These results also show that more challenging problems, such as `merge` (1-M, 1-G) — for which 71% of student solutions were identified as incorrect (see Table 5.5) — tend to require a larger training set in order achieve high coverage, as there is a wider variety of ways in which students may err. As shown in Figure 5.4, the coverage of `merge` increases more gradually than that of easier problems. Yet, coverage still increases nonlinearly and eventually reaches a plateau.

### 5.7 Conclusion and Future Work

In this work, we introduced a data-driven approach to generating high-quality concise test sets for feedback and assessment in programming courses, as well as an implementation of this method in the FEAT toolchain. FEAT incorporates concise test set generation algorithms for producing near minimal test sets and slightly larger gradated test sets with simpler cases. The gradated test sets are designed to be student-friendly for feedback and evaluation. With simpler tests, the students can more easily understand and debug problems with their programs. Both algorithms take advantage of a diverse pool of student solutions, resulting in superior coverage as compared to expert-generated tests.

For the problems studied, 1.3–64.6% of the known incorrect programs were identified as incorrect by the auto-generated test sets but erroneously deemed correct by the expert test set. Moreover, only 0–0.4% of the known incorrect programs were
identified as incorrect by the expert test set but erroneously deemed correct by the generated test sets. This highlights the advantages of utilizing student submissions to generate tests.

FEAT exploits a relatively large base test set in order to find incorrect programs. The fact that a few programs were identified as incorrect by the expert tests but not the auto-generated concise tests means that students made errors that were not exercised by any tests in the base test set. To increase coverage, the base test set could be expanded, either by adding more random test cases or by more targeted selection of additional test cases. One method to identify valuable additional test cases would be to incorporate semantic analysis of the reference solution or student solutions. The challenge would be to optimize the analysis so that it is feasible to apply to a large collection of solutions.

Another interesting question for future study is the relationship between the problem specification and the size of the resulting concise test set. For some problems, only a small number of tests were needed; for others, the size of the size of the concise test set was larger, indicating that there were more ways for a student to err. Quantifying this relationship may shed some light on the “difficulty” of various programming problems.
Chapter 6

Conclusion and Future Work

High-quality software developers are the most valuable resources in the software industry and determine the quality of software. And the quality of software is the key to software companies’ success. To ensure software quality, software companies conduct various types of software testing work, which account for a large part of software development costs in the software life-cycle (Figure 1.1). Big companies build sophisticated software testing infrastructure and hire dedicated software development engineers in testing (SDET role) to continuously work on software testing. Furthermore, besides the SDETs, all other software engineers are expected to come with professional testing skills so that they can perform white-box unit testing or test-driven development to ensure the quality of their own code by identifying software defects as they develop code. Therefore, Computer Science education, especially introductory courses, are recognized as the foundation for building the future of the software industry [5, 6].

Unfortunately, even though people understand the importance of Computer Science education, researchers have identified huge gaps that exist between university graduates and industry expectations. Therefore, more and more Computer Science educators have proposed many new pedagogical strategies and educational tools to facilitate teaching software testing. However, educators have realized that there are several difficulties to overcome in teaching software testing, such as: 1) the existing CS curricula has limited lecture hours; 2) instructors and students have limited time;
3) ad-hoc efforts do not make big differences without institutional consistency; 4) institutions need good teaching materials and effective educational techniques.

Recent years have witnessed the widespread adoption of modern educational techniques in institutional and online education. The advances in technologies have been used to build intelligent, interactive educational systems to engage students and provide personalized, self-paced, or even collaborative learning experiences. These educational tools, such as intelligent tutoring systems, visualization tools, virtual classrooms, and machine graders, have been widely used in teaching writing, electrical engineering, chemistry, biology, math, and programming. Unfortunately, the existing educational tools are either too complicated for novice students to understand and use, or require a substantial amount of instructors’ time to select, organize teaching materials, author exercise problems and grade students’ exercises.

Therefore, in this thesis, we built three educational tools specifically for teaching basic Computer Science concepts such as code execution and unit testing, which we consider to be the basis for students’ further study. There are several advantages of our tools. First of all, our tools are built based on data-driven approaches, which enables rapid development and improves students’ learning experiences in many ways. Our tools provide easy access to students via the internet, deliver study materials in rich formats to engage students to interact with the tools actively. Students can also perform self-paced learning using our tools as the tools are tolerant of mistakes which allow students to revisit the same problems or try multiple times. By analyzing historical student data, the proposed tools can generate high-quality hints to help students make progress and reduce their cognitive load. Last but not least, since the tools are built upon a large amount of real student data, based upon Crowd-Sourcing ideas, our studies have shown that the proposed tools are more effective compared with
traditional methods. We envision that using our tools not only ensures enough study hours and learning result for learners but also greatly reduces instructors’ burden on preparation, individual coaching, and grading.

6.1 Teaching Program Execution

Since reading and writing programs are the fundamental skills to learn further Computer Science knowledge and software development, including software testing. Our first tool focuses on teaching students to understand program execution. To better engage students and reduce their cognitive load, our tool converts the abstract program execution states to corresponding graphical representations and only requires students to make the correct selection among provided multiple choices to match programs and execution states. Sample short study code will be presented as hints if students make incorrect selections. The tool can automatically generate similar problems to ensure students understand the target concepts we plan to teach.

6.2 Teaching Test Case Generation

Being able to understand and write test cases for both black-box testing and white-box testing is considered to be the fundamental skill for performing software testing tasks. However, CS educators suffer from lack of proper teaching materials to teach students to write test cases. Also, grading the quality and testing coverage of student test cases are very tedious and remain a big challenge. We proposed an interactive learning system that requires students to write test cases to test against a large corpus of erroneous student programs from previous course sessions. An incorrect program from the corpus may be returned to a student to inspect if the submitted test cases fail to identify the erroneous program. The student needs to revise his/her test cases to
repeatedly improve the testing coverage in order to discover all erroneous programs. This way, students learn how to debug and write test cases for both black-box and white-box testing. The effectiveness of the proposed tool has been verified in our introductory courses.

6.3 Building a Test Case Generator

Bugs are inevitable in any software systems, so are they in students code. Many educators have been working on building high-quality machine graders to avoid tedious grading efforts and in the meanwhile providing proper feedback to students. However, finding high-quality test cases for building a machine grader is a very challenging task. Its abstract version of the problem, usually called Set Cover Problem or Hitting Set Problem, is proved to be one of the NP-complete problems in computational complexity theory. Test cases built by human experts always suffer from incomplete testing coverage issue. Here in our work, we built a test case generator using a data-driven approach to automatically generate high-quality test cases used in machine graders. We use students’ incorrect solutions collected from previous course sessions to guide us to find the most important test cases. The machine generated test cases have been proved to outperform expert built test cases and have much better testing coverage in the studies we conducted in CS1 programming courses.

6.4 Summary

In this thesis, we studied how to facilitate early Computer Science education by building three educational tools using data-driven approaches and web technologies. Our research showed that with carefully designed educational tools, we are able to overcome several common challenges in introductory programming courses and help
students achieve better performance. Our user study results indicate that the web-based educational tools built upon data-driven approaches can make learning basic Computer Science concepts, such as code execution and software testing, become less tedious, more interactive, and more effective.

6.5 Future Work

Currently, in our first tool, instructors still need to manually select and organize teaching materials to author exercise problems for the tool. One future direction would be how to automatically author the exercise problem and customized the training process based on each students’ performance using data-driven approaches. In the future, we hope to mine the common misconceptions among novice programmers and automatically infer program concepts relationship graph by studying historical student data. The information can be imported to synthesize exercise problems automatically and guide the tool to adaptively select the next problem to customize learning experiences for students based on their progress and comparison with fellow students.

For the second and third tools, in the future, we are going to build a complete data collection, analysis, processing, and tool build/update pipeline to automate the entire process and may open source the projects to benefit the Computer Science education community.

In this thesis, we’ve shown that utilizing modern technologies to build educational tools can facilitate Computer Science education and improve students’ performance. But we only scratched the surface of numerous possibilities only in introductory programming courses. We hope the exploratory work in this thesis can be a good start of improving Computer Science education by taking advantage of modern techniques and the tremendous amount of data in this Big Data era.
Bibliography


[40] J. J. Li and P. Morreale, “Enhancing cs1 curriculum with testing concepts: a


[121] K. Halas, “Mutpy 0.4. 0 mutation testing tool for python 3. x.”


