TOOLS TO UNDERSTAND HOW STUDENTS LEARN

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Lisa Yan
June 2019

This dissertation is online at: http://purl.stanford.edu/zd634rz5708

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Nick McKeown, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Chris Piech, Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Balaji Prabhakar

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mehran Sahami

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

As classroom sizes grow, instructor workload also increases. Despite innovations in technology to scale education, little has been done to improve upon the most critical component of student learning: unsupervised work on assignments. In computer science education, *learning process*—the way in which students design, debug, and explore programming assignments—is instrumental to performance and mastery. Yet few studies have defined assignment-centric metrics to measure learning process, much less design systems that transform the way we think about unsupervised work today.

My work explores how to improve student assignment work so that both the teacher and learner benefit. While many tools analyze only a student's final submission, I focus on a paradigm to collect in-depth snapshots of in-progress student work. I first discuss the complexity of characterizing progress on a programming assignment with an abstraction called *milestones*, and I show how we can use machine learning methods to visualize how students work through an open-ended graphics-based assignment. Next, I present a tool, Pensieve, which organizes snapshots of student work so that teachers see a student's problem-solving approach. This tool facilitates sit-down student-teacher conversations, where teachers can give more in-depth feedback to each individual student. Thirdly, I present TMOSS, a tool to detect excessive collaboration—that is, when a student heavily relies on peer or online resources—at any point during unsupervised work on an assignment. For both Pensieve and TMOSS, I discuss pedagogical and cultural impacts on students as well as the classroom at large.

This work points to a new paradigm for supporting learners and a path forward for designing new types of assignments that enhance the student experience. I close by discussing a graduating networking classroom project to reproduce existing research, which prepares students for research and industry careers in networking.

Acknowledgments

This dissertation would not have been possible without the support of many people; the acknowledgments in this section cover a very small subset of the individuals who have seen me through this journey. First and foremost, I am deeply indebted to my advisers, Prof. Nick McKeown and Prof. Chris Piech. From Nick, I learned how to think critically, create a compelling narrative, and present my ideas to a broad audience. From Chris, I learned how to transform positivity into passion for researching computer science education and evolving as a teacher. From both of my advisers, I have received countless, valuable lessons on how to be a good person. Thank you.

I am extremely grateful to Prof. Mehran Sahami, Prof. Balaji Prabhakar, and Prof. Emma Brunskill, for being part of my dissertation committee and bringing valuable insights and new perspectives to my work. In the early years of my PhD, I had the pleasure of collaborating on networking projects with Prof. George Varghese, Prof. Mohammad Alizadeh, and Lavanya Jose, from whom I learned about community, clarity, and dedication—all values that I brought into building systems for computer science classrooms. A large portion of my early discussions on education would not have been possible without the members of the Lytics Lab, who showed me the expansive research opportunities at the intersection of education, technology, and society. I would like to thank my collaborator Annie Hu, who taught me that enthusiasm and amicability go a long way when designing classroom tools for teachers. I would also like to thank the Stanford Graduate Fellowship and the National Science Foundation for supporting my doctoral studies.

The McKeown group has been a wonderful, plentiful source of discussions, ideas, and perspectives. A special acknowledgment goes to Bruce Spang, who painstakingly read through early drafts of my dissertation, and another to Stephen Ibanez, who kept both our lab servers and our coffee machine running. I would also like to thank other members of the McKeown group, both old and new: Serhat Arslan, Sean Choi, Eyal Cidon, Bryce

Cronkite-Ratcliff, David Erickson, Jenny Hong, Te-Yuan Huang, Catalin Voss, Kok Kiong Yap, Yiannis Yiakoumis, James Zeng, Prof. Guru Parulkar, and Renata Teixeira. I sincerely appreciate the help of Rachel, Chris, Lancy, and Andi, who both organized the McKeown group and fed all of us very well.

My PhD experience would not be complete without the many friends I met at Stanford: my Gates 3A officemates—who always greeted me no matter what time of day I came in—and Ana, Kathy, Julia, Leo, Angad, Tim, John, Michael, Daniel, Cliff, Steve, Reuben, and Charlotte—my friends who taught me work-life balance. And of course, I must give a special shout-out to "main chat"—Tim, Humphrey, Eric, and Wei—who were always there to listen and mark my messages as "read."

I would like to dedicate this dissertation to my family, who has given me unconditional love and encouragement throughout this adventure. My parents, Jennifer and Ran-Hong, have spent many hours traveling to Stanford to cheer me on—and many more years raising me to be the person I am today. I want to thank my older sister, Rose, for taking the time to be a copyeditor for this dissertation, and my younger sister, Sarah, for always reminding me to stay open-minded. Finally, I would like to thank my partner, Yusuke, for showing me endless support through your phone calls, video chats, and text messages, even when an ocean separated us.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Technology is transforming the way we think about education. Worldwide, the enrollment in tertiary education has more than doubled since 2000 [4], and undergraduate classrooms across the United States now include electronic clickers, online class forums and submission systems, and video-streamed lecture recordings.

Computer Science (CS) as a field of study has grown tremendously in the past decade; the number of CS majors declared in United States institutions has more than tripled since 2006 [3]. The subsequent boom in tools supporting these classrooms can be explained with three observations pertaining to this surge in university computer science education. The first is that programming literacy is widely regarded as essential for job marketability [1, 35]; many students are likely to take at least one CS course in their undergraduate study, even if they do not major in CS [3]. Second, the quantitative nature of CS lends itself easily to computer-assisted assessment, course organization, and classroom management systems. And third, the growth of teaching and tenure-track faculty numbers in the field has risen by only 50% and 20%, respectively, which has resulted in soaring student-teacher ratios and high management overhead. These three trends make the instructor's job much more difficult: instructors must maintain the quality of education despite burgeoning classroom sizes, even when the student body grows more diverse.

The majority of new technologies that instructors are using forgo the latter objective of teaching different students in favor of the former objective of teaching many students. Many classroom tools broadcast teaching to many students at once—e.g., creating short lecture videos, developing fully autogradable programming assignments, and answering questions on online student forums. However, simply expediting these student-teacher transactions

does not improve the student learning experience. For example, while Massive Open Online Courses (MOOCs) and fully online higher degree programs have the potential to reach millions of students, the learning outcomes in such programs pale in comparison to in-person classrooms [60, 80]. It is unclear whether the multitude of tools in our classrooms is creating a disjointed learning experience that seeks only to deliver course material efficiently, rather than to teach students effectively.

This dissertation focuses on a critical component of student learning: *unsupervised work*—that is, working on weekly assignments without explicit instructor guidance. With the current paradigm shift from traditional, paper-based lecture classrooms to paperless, interactive ones, instructors have more data on students and can consequently give better, more specific feedback. However, instructor evaluation of unsupervised work is often still based solely on final submissions. The instructor today has limited understanding of a student's *learning process*, defined in this work as a student's activities during unsupervised work; therefore, despite studied correlations between learning process and student performance [23, 79, 171, 185], the instructor cannot give adequate feedback in this dimension. Furthermore, there is little incentive to adopt new technologies that would drastically change the existing structure of unsupervised work, as many tool adoptions are costly investments to an already time- and resource-constrained instructor.

The tools introduced here provide a window into the learning process by collecting not just a student's final submission, but also their entire path towards assignment completion. Compared to prior work, these tools have two characteristics that facilitate their adoption in today's CS classrooms: First, they are designed to scale insight into different strategies of learning across large numbers of students; and second, they support an evolving set of assignments that cater to a diverse population of learners.

1.1 Tools to facilitate learning

While many tools in computer science education facilitate instruction and enhance learning *inside* the classroom, few are designed to fully support the student experience during unsupervised work *outside* the classroom. Table 1.1 compares how three different computer science classrooms use instructional and learning tools. Courses A and B are CS1 (introductory computer science) courses at an R1 (very high research activity) institution [147]

	Course A [147]	Course B [43]	Course C [116]
Course	Intro to CS (CS1)	Intro to CS (CS1)	Deep Learning
Institution type	R1	Liberal arts	R1
# Students	300	120	260
Course organization tools			
Pre-recorded lecture videos			✓
Textbook	(✓)	(✓)	
Course notes/slides	✓	✓	✓
Online gradebook	✓	✓	✓
Recorded lectures	✓		✓
Online class forum	✓	(✓)	✓
Instruction methods			
Flipped classroom			✓
Small-group classroom	✓	✓	
Pair-programming labs [113]		✓	
Course project			✓
Office hours	✓	✓	✓
Weekly assignment/feedback tools			
Fully autograded			✓
Human feedback post-submit	✓	✓	
Extensible assignments	✓	✓	
Exam grading software [153]	✓	✓	✓

✓: Tool used in class; (✓): Tool provided, but not often used.

Table 1.1: Educational tools used in three different university CS courses.

and liberal arts college [43], respectively, and Course C is a graduate course in Deep Learning [116] based on a Massive Open Online Course (MOOC) [5]. All institutions are located in the United States.

The target student group for each of these courses influences the instructor's decision on which tools to use and how. Because Courses A and B are geared towards students with no programming experience, they heavily use small-group classrooms for student discussion; Course B goes one step further by limiting lecture sections to 30 students to further encourage student-instructor interaction. In contrast, Course C is designed for advanced computer science students and thus uses its flipped classroom design to discuss practical, real-life applications and techniques during lecture time. Each course's assignment design also reflects the respective student learning goals: In Courses A and B, teaching assistants give detailed human feedback and award extra credit for students who go beyond the base assignment requirements, while Course C has fully autograded weekly assignments in favor

	Course A [147]	Course B [43]
Lecture	2 hours	2.5 hours†
Small-group discussion	1 hour	
Programming labs	N/A	2 hours
On-task assignment time	10–15 hours	6–8 hours
Office hours	1 hour	1–2 hours
Content review	0–1 hour	1 hour
Total	14–20 hours	12.5–15.5 hours

†Lecture is held in small groups.

Table 1.2: Estimated weekly time (in hours) spent on a course, broken down by activity, in two CS1 courses in the United States.

of requiring a larger, open-ended, multi-week group project.

Unsupervised assignment work is an integral component of a student's experience. Table 1.2 shows the breakdown of course commitment time, in hours per week, that a student invests in Courses A and B, as reported by their respective instructors. Students spend more than 50% of their time on unsupervised work. This balance is not unique to the two courses surveyed; unsupervised work is essential to learning because it provides a student-centered, project-based learning environment [23], where students have a place to experiment, explore, and make mistakes. Furthermore, assigning the activity requires little additional overhead as class sizes grow. Nevertheless, the level and detail of feedback on student performance that instructors can give does not scale well, and it often depends heavily on the assignment's original design.

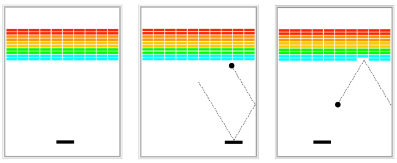
Despite the innovations listed above, the student experience with the assignment itself has been slow to change. In many classrooms, unsupervised work looks largely the same as it did a decade ago, with some administrative efficiencies: students receive an assignment handout, work on the assignment—perhaps with some help from an office hour instructor and autograder testing—and submit a final solution. Instructors receive a batch of final submissions, submit all solutions to an autograder, and read through student solutions to give more in-depth feedback if necessary. This stagnant process contrasts with how instruction in the classroom has evolved from a lecture-based, passive experience to an active learning experience rife with peer-to-peer discussion. A student's unsupervised work time is still treated as a black box whose output is a single final submission, from which instructors are expected to derive feedback on the student's problem-solving process.

Nevertheless, changing key components of the classroom is a costly investment. For example, an instructor could spend over 10 hours creating a short, 7-minute pre-recorded lecture video to replace what may amount to 15 minutes of in-person lecture [157]. When we design course tools that supplement unsupervised work environments, we must address the tension between designing engaging projects—that students feel motivated to complete—and scaling instructor feedback—that reveals in-depth understanding of our students.

Many assignments in introductory courses are designed to be interactive and open-ended. Breakout, a classic Atari game assignment (Figure 1.1) [118] is used widely in many institutions across the world [2, 57, 62, 147, 182] because its graphics-based nature appeals both to novices, who can clearly visualize what they have programmed so far, and to experienced students, who can extend the assignment solution scope for extra credit. However, the more open-ended an assignment, the harder it is to scale the grading process. The Breakout assignment is purposefully designed to be high-level so that students can build their own project, meaning that it is nearly impossible to design reliable low-level unit tests for an autograder. Course A includes one such Breakout assignment (discussed more in Section 1.2) and supports students by holding one-on-one, student-teacher conversations reflecting on the assignment, but it is unclear how valuable these conversations are without insight into how the student reached their final answer. In advanced CS courses, many assignments are supported with autograder tests that students can use to continuously evaluate their work [14, 97, 106], but this restricts assignments to well-scoped, rigid problems. Course C features a multi-week group project so that students can engage in hands-on deep learning research—yet how would this research project be scoped for a course in networking, where many research systems require months to build?

1.2 Understanding the learning process

If we gain a better understanding of how students learn through individual assignment work, then we can determine when and how to support different strategies of learning. The amount of unsupervised work will continue to increase as more and more learners enter the classroom. While it is widely believed that the best method of instruction is one-on-one tutoring [24], many institutions of higher education simply do not enough human instructors to adequately support the burgeoning student demand, so unsupervised work will persist as a key learning activity. With the lofty goal of approximating one-on-one tutoring, one



(a) Website assignment graphic.

Order	Milestones
1	Set up bricks
2	Implement mouse tracking of paddle
3	Implement ball animation (bouncing off walls)
4	Implement collision detection and brick removal
5	Finish up (implement win/loss condition)
Optional	Extend assignment for extra credit

(b) Suggested feature implementation order in course handout.

Component	Description
Assignment #	3 (out of 7). Week 4 of a 10-week course.
Work days	9
Late days	Up to 2 extra days with penalty.
Assignment handout	Tips, tricks, and suggested milestone order (as in (b)).
Overview session	2 sessions (1-1.5 hours) within 48 hours of handout upload.
Office hours	Walk-in, 5 days a week.
Paired work	N/A. Individual work only.
Grading	90% Functionality
	10% Programming methodology

(c) Assignment components and resources.

Figure 1.1: Breakout, a weekly assignment offered in an undergraduate CS1 course.

component of this dissertation work is to understand the learning process with respect to unsupervised work in CS courses so that instructors can deliver better feedback to students.

To ground this discussion on learning process, we describe how a particular assignment, Breakout [141, 147], motivates students to consider the process of problem-solving, and how the current instructor feedback system fails to evaluate how students work through the assignment. The Breakout assignment (Figure 1.1) is the third weekly assignment in Course A [147] and is due right before the course midterm examination. Because this is the first big programming task that students must complete individually, instructors offer several guidelines and incentives for success.

To keep students from feeling overwhelmed, the assignment handout presents the Breakout problem in terms of the key features and suggests an order for achieving assignment milestones (Figure 1.1b). As soon as the assignment is released, undergraduate teaching assistants (TAs) hold overview sessions to ensure that students understand the assignment objectives and have the opportunity to ask clarifying questions. While students are working on the assignment, they can attend drop-in evening office hours where TAs can

provide debugging help. After final submission, student work is evaluated along two dimensions: *functionality*—how well the student's Breakout game operates according to the task outline—and programming methodology, or *style*—how well the student code is designed and written.

For unsupervised work—such as programming Breakout—students may interact with the assignment material in several ways. Tools for understanding learning process should acknowledge that students may have different approaches, including but not limited to:

- **Planner-oriented progression.** Carefully structuring programs over time [162]; e.g., guided by the milestones outlined in the handout, a student *planner* would tackle Milestone 2 after successfully debugging and testing Milestone 1.
- **Tinkering-oriented exploration.** In contrast to goal-oriented progress, a student *tinkerer* would program with just-in-time planning and incremental changes [20, 23, 162]
- **Floundering and struggling.** Haphazard activity that does not result in functional progress; e.g., a student conceptually does not understand animation loops for animating ball movement and has given up on debugging their current code.
- **Other assignment-related activities.** Programming with the intent of fixing the formatting of code, variable names, indentation, etc.; e.g., a student who has already achieved functional success and is instead adding comments, decomposing functions, and "cleaning up" their submission with respect to the style grade component [22].

It is well-documented that the student's learning process affects their success on the specific assignment, as well as their overall course performance [171]. Both planning and tinkering are activities associated with successful *functional progress*, defined as progression towards the assignment's functional goals (e.g., making a playable Breakout game) [19]. On the other hand, floundering, struggling, and poor time management may lead students towards plagiarism and *excessive collaboration*, where they rely heavily on peer or online code in ways that may violate the institution's course collaboration policy. If instructors can identify how students are learning, they can better support different types of productive learning and deter students from ineffective strategies.

There are many additional benefits to classroom education if instructors can leverage a deeper understanding of the learning process:

- **Identify overall trends in student learning.** Instructors should understand if the assignment is correctly addressing student learning goals, so that they can adjust the learning experience both during and after the assignment.
- **Give tailored feedback to specific students.** Given that teaching resources are scarce in large classrooms, instructors need to identify which students need additional assistance or supervision. The degree of help can range from offering autonomous tools to groups of students (e.g., planners may benefit from software for scaffolding intermediate planning stages [111]) to recommending in-person, one-on-one instructor office hours.
- **Identify ineffective learning processes.** Instructors need to identify when students are *not* learning at all but would benefit from human support over autonomous tools. For example, instructors should be able to flag students at risk of excessive collaboration and provide in-person help where needed.
- **Train students' metacognitive skills.** Inherent to the process of learning is observing how students exhibit *metacognition*, or learning how to learn (discussed more in Chapter 3). Students who exhibit stronger metacognition are better self-regulated learners, who can strategize and effectively use classroom resources for their individual learning goals [151, 160]. If instructors can develop metacognition skills in introductory computer science students, they can train these students to navigate their own learning in more advanced courses.

For the Breakout assignment, instructors only scaffold the learning process with a suggested order of functional progress (Figure 1.1b), given before the student starts unsupervised work. There are very few resources for evaluating learning process, as evaluation of student work is by and large only on final submissions. Instructors assess the final submission for functionality and programming style; they currently are not equipped to give feedback on the student's programming methodology—i.e, how the student approached the problem, debugged their solution, and followed common programming practices for success. Furthermore, feedback is an incredibly time-intensive process for instructors; it is important that any tools designed for this task should encourage students to incorporate this costly instructor feedback into work on future assignments.

The challenge to developing classroom tools to understand the learning process is therefore two-fold: (1) Identify learning activities so that instructors can give feedback to students, and (2) do so without significantly increasing instructor burden. However, existing research is difficult to translate to tools that instructors can use to provide actionable, scalable feedback to students in large classrooms. This is in part because existing research often characterizes student work patterns using low-level indicators of student progress, such as code diffs, completion times, or errors [7, 16, 23, 33, 79, 155] (discussed more in Chapter 2). While these measurements are easy to calculate for each student, they do not translate well into actionable feedback for the student and are not necessarily transferable across contexts [122, 185]. There has been some successes in modeling learning process and propagating feedback on in-progress student work [126, 127, 169, 177], but the programming languages in these research studies, albeit Turing-complete, are block-based or variable-constrained, and they would thus be intractable for more complex, open-ended assignments like Breakout, much less for any real-world assignments in advanced CS courses.

The main barrier to tractable analysis and modeling student assignment work is one of data sparsity. For many assignments, the distribution of student abstract code syntax trees in students' final submissions follows a Zipf distribution [128, 177], which has two implications. First, the task of understanding the space of just *final* submissions is akin to the task of understanding natural spoken language [123]. Second, while instructors can predict the most likely solutions for any programming assignment, there is always a non-zero probability of encountering a completely new submission, for which there is no understanding. In other words, precise, completely autonomous understanding of student assignment work is intractable; compared to natural language, which has billions of labeled examples, the number of student submissions is rarely in the millions [87], if not more often in the hundreds or thousands [126]. This problem is exacerbated if we consider saving snapshots of student progress; for a 480-student course, the average student programming Breakout submits a final submission 300 lines in length after about 240 intermediate snapshots of student work. The solution space is far too large for human instructors to review within a reasonable timeframe, much less use to develop constructive feedback to students. If we supplement human feedback with recent data-driven techniques like machine learning (e.g., for natural language modeling or image classification), these methods must be adapted to much smaller—but equally complex—datasets.

The tools in this dissertation are introduced as a first step towards enabling instructors

in CS classrooms today to give better feedback on all dimensions of learning. These tools provide a data-driven approach to maximizing the human resources available at an undergraduate CS institution. With a better understanding of how unsupervised work shapes different trajectories of student learning, instructors can promote effective, self-regulated learning among the majority of students, while extending in-person help to students who need it the most.

1.3 Innovations in assignments

The design of the assignment itself is also critical to understanding how students learn and is discussed as a second component of this dissertation. To transform the process of unsupervised work means to reevaluate every facet of the activity: If we are to design tools to understand the learning process, how can we optimize the underlying learning environment for our students? The CS assignment creating this learning environment must be designed to develop student problem-solving skills yet also support scalable grading and feedback, while providing the students with a meaningful, motivational experience that prepares them for the role that computing will play in their lives.

In CS1 and CS2 courses around the world, there is a plethora of assignments that fit these goals. The holy grail of introductory assignments would be one that encourages exploration and tinkering while simultaneously supporting an efficient, fully automatic feedback pipeline. An increasing number of assignments are designed with this goal in mind [118], and many others promote friendly competition between students [70, 125, 136, 167]. In particular, open-ended, graphics-based assignments like Breakout are gaining popularity due to their ability to equalize the playing field for students with varying programming backgrounds [44, 70, 147]. For novices, the graphical visualization gives them immediate feedback on what they have implemented so far; for experienced programmers, the open-ended nature leaves room for experimentation and exploration beyond the base assignment. Despite the potential benefits to students, instructors may find these assignments arduous to grade because of the large space of potential solutions.

In more advanced courses, instructors must design assignments that prepare students for the experiences that they will face in the field. For introductory courses, assignment goals are clear: get students excited about computer science, and give assignments that connect computing to their everyday lives. On the other hand, advanced courses should contain tasks

that students will inevitably face in engineering and research. As a result, graduate courses in computing often include a larger, multi-week project with a final report write-up, so students can develop their technical expertise, ability to manage a project, and presentation and documentation skills. Designing these projects to be open-ended is perhaps even more important for advanced students, because as future engineers or academic researchers, their work will unquestionably venture into uncharted territory. Advanced students can use these course projects to practice how to navigate the unknown. It is costly for instructors to invest in relevant, timeless assignments. Not only should the ideal assignment encourage students to develop and implement a creative solution within a reasonable timeline, but also instructors should be able to grade it on a consistent scale and support the learning process with classroom resources.

1.4 Contributions and outline

This dissertation explores how to improve individual, unsupervised work as it exists in computing education courses today. We present four tools that can be implemented immediately in existing classrooms to understand the learning process and improve education. The remainder of this dissertation is organized as follows:

In Chapter 2, we introduce the PyramidSnapshot dataset along with the first of our tools: milestones, a method to identify functional progress on a large dataset of intermediate student submissions. This work is the first to provide a framework for merging expert knowledge with machine learning image classifiers in order to characterize functional progress on graphics-based assignments, which are complex but popular in introductory CS courses. With this method, researchers and instructors alike can understand how students work through the assignment, from the perspective of the individual student, as well as the class as an aggregate. The central content of this chapter was originally published in Yan et al. [180].

In Chapter 3, we present Pensieve, the first of two instructor-facing tools, which visualizes student work over time. Using this tool, instructors can use the learning process to enhance feedback and student metacognitive development on unsupervised work. We report preliminary results of how students in an offering of CS1 that used Pensieve performed better than one that did use not Pensieve. The content of this chapter was originally published in Yan et al. [178].

In Chapter 4, we present TMOSS, our second instructor-facing tool for understanding the learning process. Many existing tools only work on final submissions of student work; in contrast, TMOSS can detect when, where, and how excessive collaboration impacts unsupervised work. Using TMOSS, we were able to identify different student collaboration patterns and analyze how these patterns correlated with course performance. The content of this chapter was originally published in Yan et al. [181].

In Chapter 5, we present an assignment designed to improve the graduate networking student learning experience. The Reproducing Research Results project tasks students with reproducing key findings in existing networking research, enabling students to experience a realistic research environment while connecting them with the rest of the networking community. The content of this chapter was originally published in Yan et al. [179].

Finally, Chapter 6 summarizes the dissertation's main contributions and describes potential future research directions for supporting learning not only in today's undergraduate computing education, but also in the larger space of education.

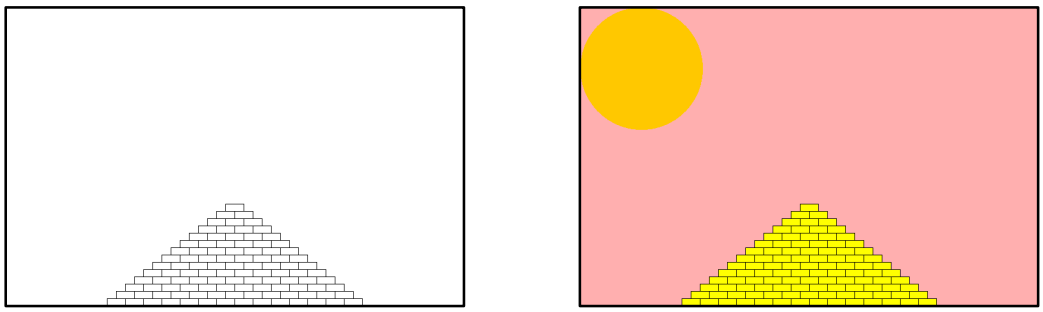
Chapter 2

Pyramid Milestones

First-time CS students learn by programming—they must design an approach, debug their code, and iteratively improve towards a final solution. For a teacher, however, a single timestamped submission per student at the end of this process is insufficient to capture all the intermediate steps a student has taken towards a solution. While a final submission gives some indication of how a student designed their solution, it reveals very little about the attempts and detours that a student may have made along the way.

Consider two students, Student A and Student B, working on a CS1 graphics-based assignment called Pyramid, where the task is to draw a Pyramid. Both submissions received full credit; furthermore, a teacher noticed that while Student A achieved only the baseline Pyramid (Figure 2.1a), Student B went above and beyond, meriting extra credit (Figure 2.1b). However, the final submission is a shallow reflection of the two students' learning process. On the midterm exam two weeks later, Student A was a top scorer, whereas Student B scored among the bottom ten in a class of 500 students. A deep dive into the students' work would reveal that after just 14 minutes, Student A had drawn their first Pyramid shape. In contrast, Student B took 5 hours and 45 minutes to reach the same milestone and spent over 60% of their total work time debugging compile-time errors.

For CS courses, while it is easy to generate these low-level metrics to demonstrate the difference between Student A's and Student B's learning processes, our teacher will find it much more difficult to use this information to give Student B actionable feedback on how they can improve. What would be more meaningful than these statistics is a characterization of *functional progress* over the course of a student's unsupervised work. If snapshots of student code could be mapped to *milestones*, or incremental attempts of functional progress



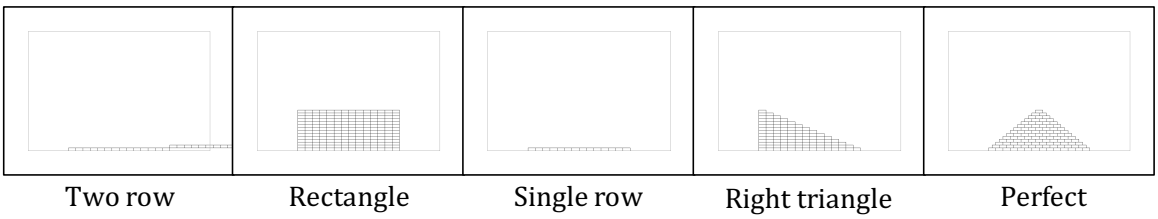
(a) Assignment goal. (b) Extra credit example.

Figure 2.1: Compiled image output of two full-credit student final submissions of the Pyramid assignment by (a) Student A and (b) Student B.

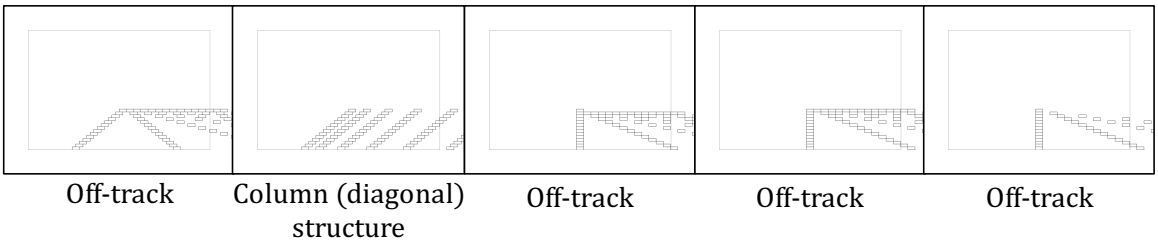
towards the assignment goal, then our teacher can differentiate between—for example—an on-track student (Figure 2.2a) and a struggling student (Figure 2.2b). Our teacher can then then discuss with both students their respective approaches in detail, as well as reflect more broadly on whether the class is adequately prepared to navigate the Pyramid assignment.

The difficulty in understanding functional progress during unsupervised work stems from the multitude of different solution paths possible. While manually inspecting the functional progress of two students may be feasible for a human instructor, it is intractable to extend this type of analysis to all 500 students in the classroom above. Unit testing and other automated assessment tools are often only designed to test functionality of the final submission, and teachers would have to design and engineer additional tests for identifying intermediate milestones. Existing tools are also difficult to tailor to *graphics-based* programming assignments, which have recently gained popularity in many CS1 courses. Due to their open-ended nature—enabling programmers of all levels to get quick, visual feedback in an exploratory environment—the large solution spaces often render unit-testing development or syntax-based code analysis insufficient for final submissions, much less intermediate code snapshots.

In this chapter, we explore how to automatically characterize functional progress on a graphics-based assignment, Pyramid. Instead of modeling student code, our scheme uses only the compiled image outputs of student work to generate *milestone labels*, which correspond to mutually exclusive steps to success. By focusing only on the compiled image output of these code snapshots, our task of understanding functional progress becomes an image classification problem, and we are able to tap into the rich field of computer vision,



(a) Milestone labels for a series of images from a typical student.



(b) Milestone labels for a series of images from a struggling student.

Figure 2.2: Two students' compiled images of functional progress on the Pyramid assignment, annotated with milestone labels.

where machine learning advances have matched [86] and, more recently, surpassed [186] human ability to detect objects from pixel input. However, to our knowledge, in spite of their effectiveness, contemporary vision classification techniques have rarely been applied to student code, as many state-of-the-art techniques are *supervised*—requiring well-labeled, plentiful data, which student assignment data often lacks.

Our case study is the Pyramid assignment, a canonical CS1 graphics-based task (Figure 2.1, Appendix A). In order to marry the two fields of CS education research and computer vision, we devise a tractable way of preparing a labeled dataset—tagging timestamped, program image output with milestone labels—for supervised machine learning. The PyramidSnapshot dataset contains timestamped, program image output of 2,633 students over 26 CS1 offerings from the same university; 84,127 of 101,636 images of intermediate work are annotated with one of 16 milestone labels mapping to functional progress. Through the method introduced in this chapter, not only can we expand our understanding of unsupervised work, but we also hope to expedite the preparation of new assignment data for functional progress analysis.

There are three main contributions of this work, which follow after a discussion of related work in Section 2.1: First, in Section 2.2 we describe an efficient method of using instructor expert knowledge to quickly characterize functional progress on a large fraction of our

dataset. We supplement these hand-labeled expert data with machine learning techniques to cover the entire dataset in Section 2.3. Second, in Section 2.4 we show how instructors can use functional progress to learn more about how different groups of students interact with the Pyramid assignment. Third, in Section 2.5 we discuss the feasibility of using computer vision techniques to grade student final submissions. This is a first step in providing scalable characterization of student progress during unsupervised work, and we published the fully labeled PyramidSnapshot dataset[1] created in this work so that researchers can extend our work to new assignments.

2.1 Related work

There is a growing body of work on automated assessment tools, which are designed to understand and give feedback to students. Many courses also employ test-driven learning, where students use a provided set of unit tests or write their own using a system like Web-CAT [49, 81]. However, unit tests in general are often brittle, time-consuming to develop, and hard to apply to graphics-based assignments, which allow for variation among correct solutions. Thus, most contemporary work aims to understand student programs based on abstract syntax tree (AST) structure [71, 108, 112, 144, 168]. While these approaches work for short programs with low complexity, Huang et al. found that implementing AST-based feedback in general is as hard a task as autonomously understanding natural language [72].

To make CS courses more accessible for a diverse set of learners, unsupervised work should encourage personal approaches to programming [162]. Some programming languages inherently encourage exploration, such as Scratch [138] and Alice [41]. For text-based programming languages, on the other hand, creativity is pushed to the assignments, which are often open-ended tasks that support visual output [93, 118, 140, 158]. Flexible assignments support both planner and tinkerer approaches to solving the problem, because such assignments allow strategic planning of milestones *and* incremental, exploratory steps towards the solution.

Assignment work patterns are often strong indicators of student performance in the course [98], the interplay between work and plagiarism [181], and the extent to which students are tinkering versus making forward progress [23]. It has been found that a student's

[1] Dataset available at `http://stanford.edu/~cpiech/pyramidsnapshot/challenge.html`.

interaction with syntax, compile, and runtime errors can be predictors of student performance [16, 33, 79, 155]. Ahadi et al. showed that assignment completion times, among other low-level assignment features, were predictive of exam scores [7], while Piech et al. showed that functional progress on simple, variable-free assignments can better predict exam performance [127]. In the classroom, Morrison et al. found that encouraging students to label functional subgoals contributed to better student performance, but such labeling should be well-monitored by instructors [110, 111]. Understanding functional evolution of student solutions has been explored for grid-world, simple block based assignments [126, 127, 169]. To our knowledge, our work is the first to explore functionality and student progress for assignments that use pixel-based graphical output, which are complex and more typical of computer science learning environments.

We circumvent the difficulty of AST-based functionality analysis by using *pixel-based* visual program output. In 2013, a team of researchers from DeepMind demonstrated that a convolutional neural network-based algorithm could learn to play Atari games as well as humans using only pixel output [107]. Because the Atari games used in the DeepMind paper are very similar in complexity to the outputs of assignments typical in CS1 and CS2 classes—and some of the Atari games are in fact classic homework assignments [118]—we have reason to believe that modern computer vision algorithms should be able to understand the output of our student's graphical programs from the pixel level. Despite this potential, to the best of our knowledge, the capacity for understanding graphics prior to this paper has been used mostly to play and rarely to educate. Open datasets have been integral to the early evolution of computer vision techniques [53, 86, 95]; we hope that our contribution of this dataset will help the CS education community evolve.

2.2 Preparing a dataset

The most effective machine learning classifiers today use supervised learning techniques on reliable, well-labeled canonical datasets [74, 159]. However, student assignment datasets are anything but—the biggest challenge to automatic characterization of student unsupervised work is establishing ground-truth labels. Autograder output and grading rubrics may be appropriate labels for student final submissions (elaborated further in Section 2.5); however, when characterizing in-progress student work, such schemes fall short because they are designed to assess *complete* submissions, not everything in between. In this section, we

```
public void run() {
  for(int i = 0; i < BRICKS_IN_BASE; i++) {
    // calculate row variables
    int nBricks = BRICKS_IN_BASE - i;
    int rowWidth = nBricks * BRICK_WIDTH;
    double rowY = HEIGHT-(i+1)*BRICK_HEIGHT;
    double rowX = (WIDTH - rowWidth)/2.0;
    // draw a single row
    for(int j = 0; j < nBricks; j++) {
      // add a single brick
      double x = rowX + j * BRICK_WIDTH;
      rect(x, rowY, BRICK_WIDTH, BRICK_HEIGHT);
    }
  }
}
```

Figure 2.3: Sample solution code for the Pyramid assignment.

discuss how we transformed student work Pyramid assignment into the PyramidSnapshot Dataset, a labeled dataset prepared for supervised classification tasks.

The Pyramid assignment is a sub-problem in the second week of a 10-week CS1 course at Stanford. The assignment is scoped as the students' first exposure to variables, object manipulation, and for-loop indexing. As part of their individual, take-home assignment, students use Java's ACM graphics library [142] to draw bricks on the screen and construct a pyramid shape, and they are awarded extra credit points if they can extend the correct pyramid. A teacher solution including nested loops and computation of several intermediate variables is shown in Figure 2.3, but in practice student code is much more complex.

To collect student data during their unsupervised work, we modified the Eclipse Integrated Development Environment (IDE) to support *snapshotting* of student code [127] as they work through the assignment, shown in Figure 2.4. The IDE manages a *process repository*, a Git repository that stores a snapshot of student code whenever a student compiles and runs the assignment, meaning that snapshot frequency could be on the order of minutes when a student is actively working. Then, when the student submits their final program for grading, the IDE automatically packages up the process repository and stores it on the course server.

In the CS1 course studied, the Pyramid assignment has been used for many years, with little variation in assignment scope and grading rubric. We analyzed 2,633 student submissions from as far back as 2007 (where the bulk of data was collected between 2012

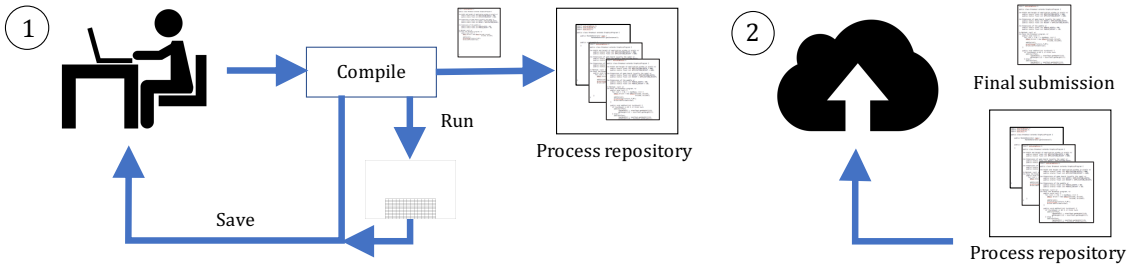
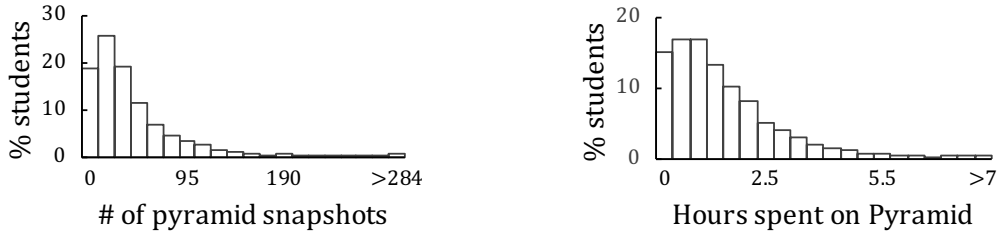


Figure 2.4: The pipeline for creating a process repository during unsupervised work.



(a) Number of snapshots per student.

(b) Hours spent on the assignment.

Figure 2.5: Pyramid work time analysis on 2633 student process repositories.

and 2014) by compiling and running all Pyramid code files in the process repositories to generate timestamped images. Figure 2.5 shows the distributions of the number of snapshots ($\mu = 52, \sigma = 59$) and hours spent ($\mu = 1.7, \sigma = 1.5$) for process repositories in our dataset. Of the 138,531 snapshots in our dataset, we successfully generated 101,636 images; 36,895 snapshots had runtime or compile errors. While the default canvas size for Pyramid is W754 $\times$ H492, we have found that a common functional error that students encounter is drawing objects off-screen; we thus add a 100px border to the graphics canvas and save our images to be W954 $\times$ H692, in color.

2.2.1 Dataset complexity

For a graphics-based assignment like Pyramid, using image output to represent the functionality of student work is less complex—and therefore easier to work with—than using a text-based representation like ASTs. Like many other coding assignment datasets that have been analyzed in the past, the frequency distributions of both the Pyramid assignment's code ASTs and the image output files follow Zipf's law, the same distribution as natural language. This insight implies that the exponent s of the Zipf fit can be used as a measure

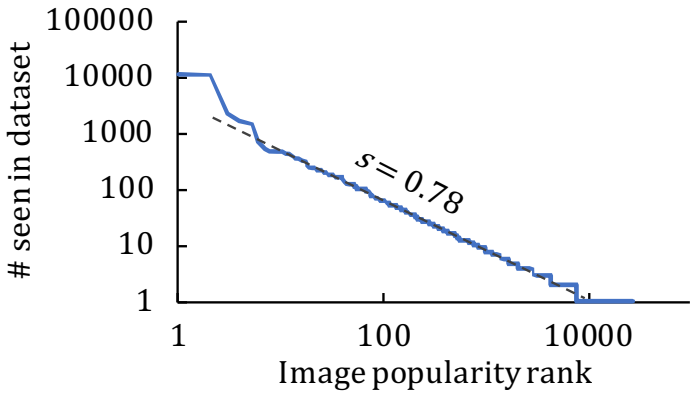


Figure 2.6: Rank-frequency distribution of the PyramidSnapshot images.

of the complexity of a programming dataset, where a higher exponent means that a dataset has a higher probability of observing the same solution more than once and is thus less complex [123]. The frequency distribution of Pyramid ASTs fits to a Zipf exponent $s = 0.57$, which is notably more complex than logistic regression implementations ($s = 1.82$), CS1 first week homeworks ($s = 2.67$), and pre-CS1 (CS0) coding challenges [128, 177]. On the other hand, the Pyramid *image outputs* have a Zipf distribution with exponent $s = 0.78$ (Figure 2.6), suggesting that modern tools for understanding images may be more effective than an AST-based approach for understanding functionality of intermediate solutions.

2.2.2 Labeling milestones efficiently

Next, we discuss how to design labels of functional progress for our PyramidSnapshot image dataset. While Figure 2.1 shows that student activity is highly variable, nevertheless we want to quantify the functional behavior of students so that we can visualize progress over time in Section 2.4.

We label the PyramidSnapshot dataset with *milestone labels*; that is, each image is categorized into one of our 16 visual categories of intermediate work. Figure 2.7 shows an example image for each of these milestones. These milestone labels were decided after looking through the top 100 most popular images. It is important to note that the term *milestones* does not imply that a student must progress through all 16 milestones to complete the assignment; rather, they represent different approaches to the assignment. For example, Milestone 2 (Single row) and Milestone 3 (Diagonal) are different implementations of a single loop of bricks. After consulting with an instructor, the recommended approach—in other

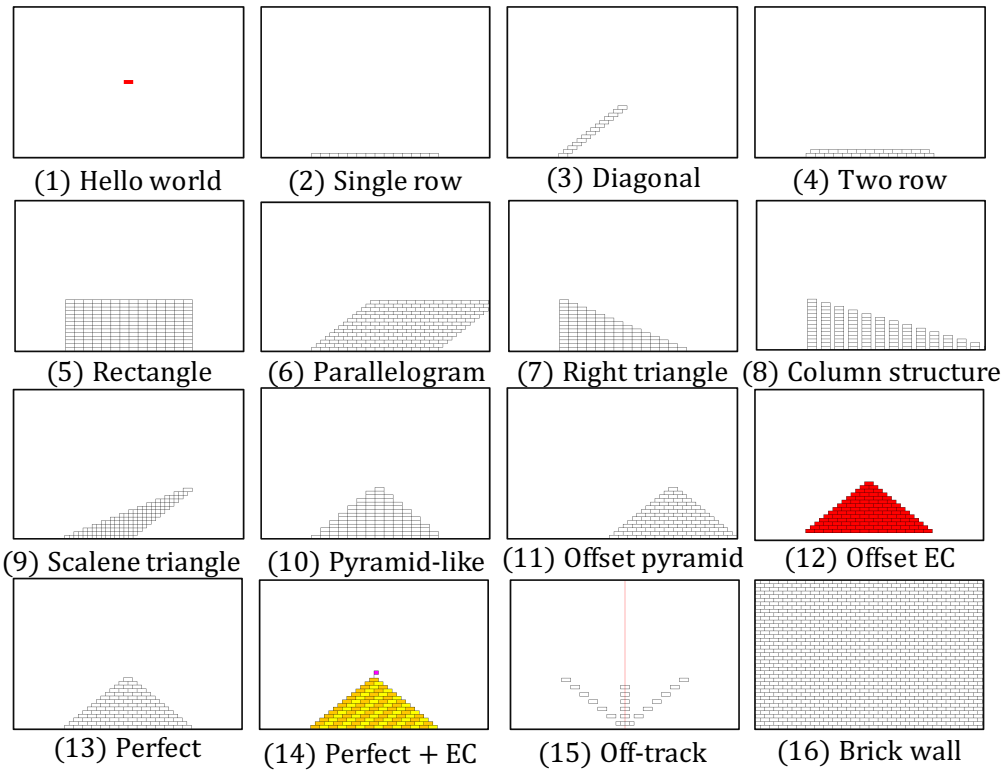


Figure 2.7: Examples from the 16 milestone category labels in the PyramidSnapshot dataset; EC stands for Extra Credit.

words, the one most suggested by instructors—for this assignment is to draw horizontal rows; Milestone 3 (Diagonal) and Milestone 8 (Column Structure) therefore pertain to single and nested loops, respectively, for both diagonals and columns, which are considered more difficult approaches to debug. Milestone 15 (Off-track) is marked for any image that does not fit into the other 15 categories.

Based on instructor advice, in Table 2.1 we group the 16-dimension space into five meaningful stages of knowledge, organized based on whether students are working on a single loop (Stage 1), a nested loop (Stage 2), adjusting brick offset within the nested loop (Stage 3), or enhancing a completed assignment and going beyond what is expected (Stage 4). The remaining milestones are grouped as "Other/Off-track." We note that higher-level abstracted knowledge stages are more meaningful for a human grader as such stages will not differentiate between two milestones associated with the same level of student cognitive understanding.

	Stage description	Milestones
1	Single row	(2) Single row, (3) Diagonal, (4) Two row
2	Nested loop	(5) Rectangle, (7) Right triangle
3	Adjusting nested offset	(6) Paralellogram, (9) Scalene triangle, (10) Pyramid-like, (11) Offset pyramid
4	Adding final details	(12) Offset EC, (13) Perfect, (14) Perfect + EC
–	Other/Off-track	(1) Hello world, (8) Column structure, (15) Off-track, (16) Brick wall

Table 2.1: Knowledge stages are groups of milestones.

	# Snapshots	Unique images	Effort score
Total	138531	27220	5.1
Labeled	84127	12077	7.0
Unlabeled	17509	15143	1.2
Error	36895	–	–

Table 2.2: Label coverage of the PyramidSnapshot dataset.

It would be an insurmountable task for a researcher to label over 101,000 images in our PyramidSnapshot dataset. However, we observe from the distribution of our data (Figure 2.6) that only 27,220 (27%) of these are unique. Furthermore, the perfect pyramid (Milestone 13) in our dataset is the most frequent and occurs over 11,000 times across all student work, and the ten most popular images cover 20,219 images (20%) in our 101,636 image dataset. One can therefore label unique images to milestones in order of popularity in the dataset; a single researcher labeled 12,077 unique images with corresponding milestones in 20 hours over three days, covering 84,127 images (83%) of the actual dataset.

Table 2.2 shows our label coverage of the dataset, where the *effort score* represents the gain of labeling a unique image as the frequency of that image appearing in the overall dataset. The average effort score of unique images labeled was about 7 repeated images; on the other hand, each remaining unlabeled unique image appeared on average 1.2 times in the dataset, meaning that we would not have gained much with this labeling strategy had we continued into the tail of the distribution. Figure 2.8 shows how our labeling strategy covers many of the images within each student process repository. With the effort strategy, we have ensured approximately 87.5% of student process repositories are at least 75% labeled.

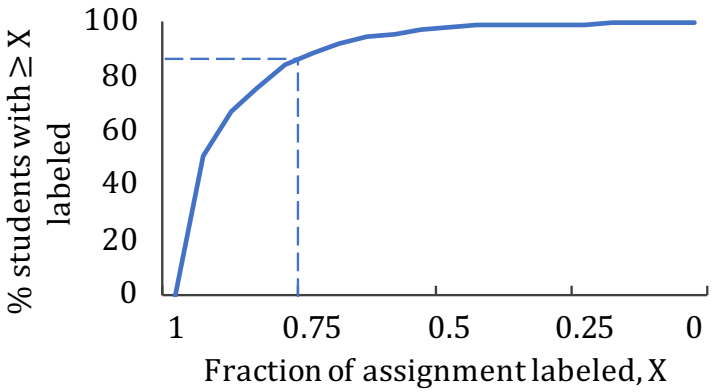
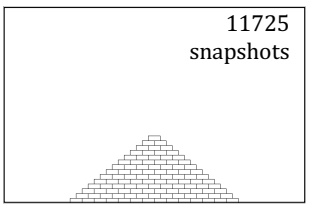
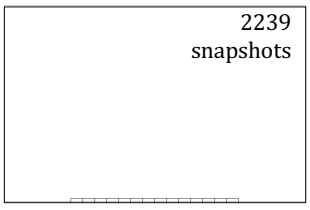


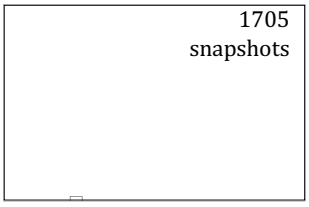
Figure 2.8: CDF of the fraction of process repositories labeled with the effort strategy in Section 2.2.



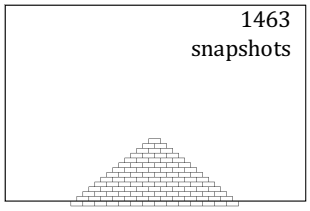
(a) Perfect pyramid (Milestone 13).



(b) Single row (Milestone 2)



(c) Hello world (Milestone 1)



(d) Offset Pyramid (Milestone 11)

Milestone		Unique (%)	Effort score
1	Hello world	14.25	6.6
2	Single row	11.79	6.4
3	Diagonal	8.81	4.4
4	Two row	2.91	4.5
5	Rectangle	4.96	5.2
6	Parallelogram	3.55	4.4
7	Right triangle	7.71	4.5
8	Column structure	1.27	2.3
9	Scalene triangle	3.59	3.2
10	Pyramid-like	8.74	3.4
11	Offset pyramid	6.85	9.0
12	Offset EC	2.16	2.4
13	Perfect	4.39	31.4
14	Perfect + EC	7.14	3.1
15	Off-track	11.49	2.6
16	Brick wall	0.40	6.2

(e) Milestone label coverage.

Figure 2.9: Labeling the PyramidSnapshot dataset: (a-d) Top four most popular images in the dataset; (e) Distribution of milestone labels over dataset.

If we analyze the coverage of the milestone labels in Figure 2.9, we observe that three of the top four most popular images are what an instructor would expect a student to complete, while the fourth most popular image represents the most common incorrect solution. Figure 2.9e shows the effort score breakdown by milestone. While Milestone 1 has the largest share of unique images, Milestone 13 yields the highest effort score because the most popular image of a perfect pyramid is an order of magnitude more common. Different students will rarely share extra credit image outputs, and naturally Milestones 12 and 14 have among the lowest effort scores.

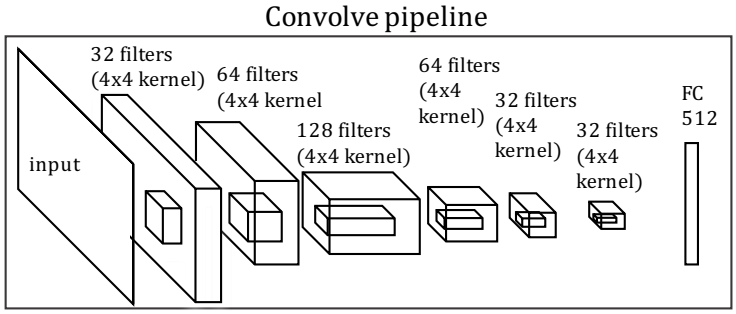
2.3 Milestone classification

In this section, we explore automatic classification methods for classifying intermediate snapshots by milestone. We realize that the popularity-based labeling scheme from the previous section presents a scalability challenge for other assignments, even if we use image frequency to reduce labeling effort. Instructors who use this labeling scheme in the classroom will probably label closer to tens or hundreds of images—and not the thousands labeled in this work—and subsequently rely on machine classification for the remainder of the dataset. We therefore pay special care to analyze how the accuracy of a classification model works as we vary N, the number of most popular images that we label, defined as our *training set* size.

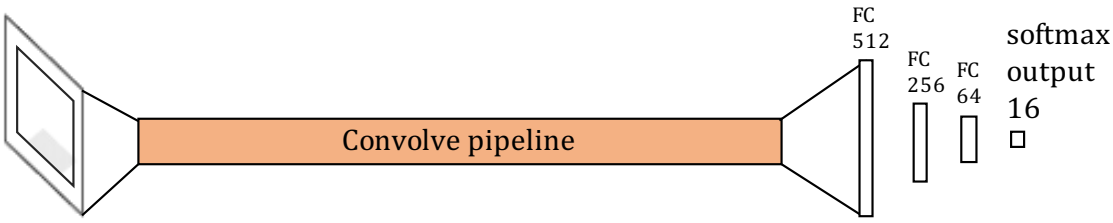
The classification models used in this section are by no means comprehensive, nor are they designed to be optimal. Instead, in the original publication [180], these models were presented as *baseline* models for future researchers to use as benchmarks. However, our results show that a simple neural network baseline works quite well even with a very small training set size, and we conclude by discussing next steps for improving upon our baseline classification accuracy.

2.3.1 Method

Given a training set of the N most popular images, we consider three models for classifying our images with milestones. For Models 2 and 3, we train only on **unique** images, and not the entire repeated image dataset, since we have a one-to-one mapping between image and milestone. We preprocess each bordered image by grayscaling and downsampling by 2 for an input of $346 \times 477 \times 1$.



(a) The general convolve pipeline.



(b) The convolve pipeline used for milestone classification.

Figure 2.10: Model 3, the neural network model for milestone classification.

Model 1: Unit test. Our first approach is based on the premise of unit testing: select one representative image for each milestone, and for each test image, predict a milestone only if the test image pixels exactly match that of the reference; otherwise it predicts nothing. Unlike the other two models, the unit test training set size is fixed at $N = 15$ unit test images; we do not save a representative image for the Off-track milestone (Milestone 15).

Model 2: K-Nearest Neighbors. The second approach, K-nearest neighbors (KNN), is a common baseline used for computer vision [86]; for each test image, define the K nearest neighbors as the K images from the reference training set (of size N) that have minimum sum-squared-difference (pixel-wise), and predict the most common milestone out of the nearest neighbors.

Model 3: Neural network. Our third approach is a deep learning approach: train a convolutional neural network to return the most likely milestone label, shown in Figure 2.10, which has a model size of 1.2 million parameters. We train on the top N most popular images and optimize for softmax cross-entropy loss.

Overall accuracy	Unit test ($N = 15$)	KNN ($K = 100$, $N = 11000$)	Neural network ($N = 11000$)
By milestone (16 milestones)	.275	.037	.562
By knowledge stage (5 stages)	.275	.552	.649

Table 2.3: Milestone classification results. Accuracy of each model (with training set size N) by milestone and by knowledge stage on the 11,000 most popular images.

2.3.2 Results

To evaluate our approaches to classifying milestones from graphical image input, we trained each classifier with a varying size of N on our training set of the most popular images, and we evaluated its overall accuracy on two datasets: the *validation set*, composed of the top 11,000 popular images (corresponding to 70% of our image dataset), and the *tail set*, composed of the remaining 1,077 labeled images. We use the validation set to decide which classifiers work for common snapshots; the tail set accuracy is an indicator for whether our classifiers work on rarely seen snapshots.

For all models, we use two metrics of accuracy: milestone accuracy—akin to an exact match metric—and knowledge stage accuracy—where if the predicted milestone is within the same knowledge stage as the actual milestone, the model classification is correct. While our models were trained with unique images, we incorporate image popularity when assessing model accuracy; both metrics are reported as an average, where each image is weighted by its frequency in the PyramidSnapshot dataset.

We first discuss accuracy results on the validation set. The results for each classifier with the best choice of training set size N is reported in Table 2.3, showing the overall accuracies in classifying all 16 milestones and all five knowledge stages. For both accuracy metrics, we immediately see that our neural network outperforms the other two unit-test and KNN ($K = 100$) models. In Figure 2.11 we report our neural network accuracy for predicting a milestone that is in the same knowledge stage as the correct milestone label; the model reports at least 70% accuracy for Stages 1, 2, and 3, which are the most formative in determining student progress towards the assignment goal, and therefore the low accuracy in identifying Stage 4 is acceptable.

To understand how our third model, the neural network, connects image to label, we use

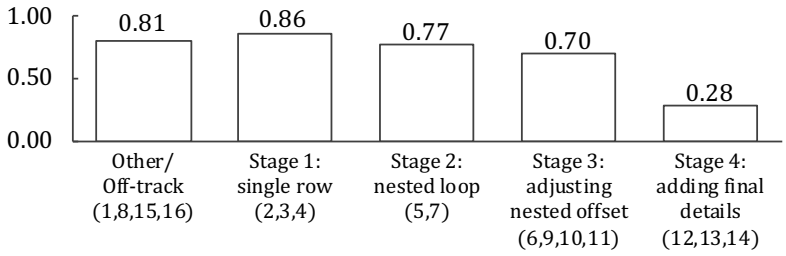


Figure 2.11: Accuracy breakdown of neural network model performance by knowledge state.

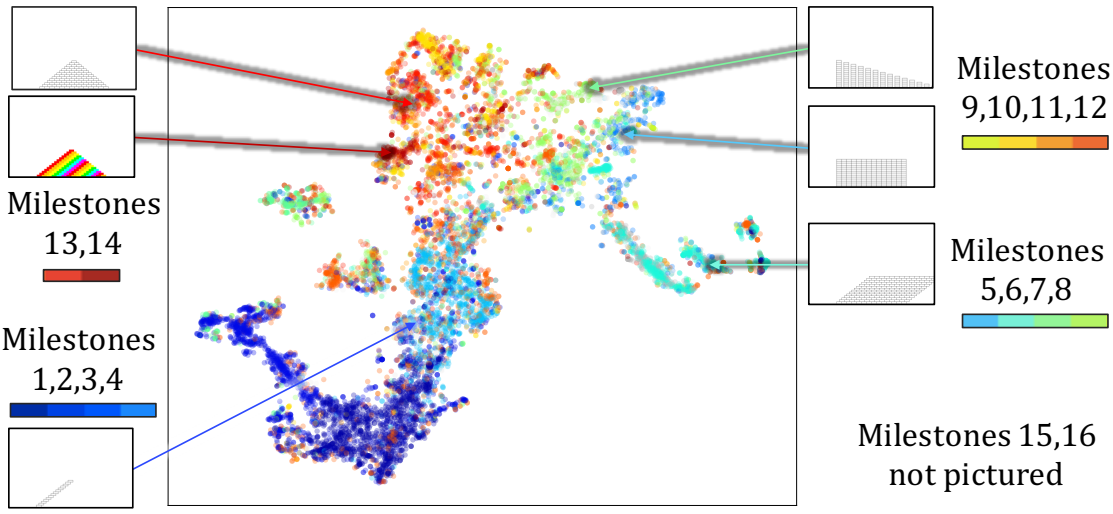


Figure 2.12: t-SNE plot of model embeddings, color-coded by milestone.

the t-SNE algorithm [165] to compress the model's internal image representation into a 2-D, clustered visualization. Figure 2.12 shows that our image embeddings roughly cluster by their milestone; however, there are some perfect pyramid images (Milestone 13, red) hidden among the Hello world milestones (Milestone 1, blue). Upon checking these incorrectly classified images, we found that some of them depicted a very small pyramid, which could easily be misclassified as a single block, but others were simply incorrectly placed in the embedding space. Our hypothesis is that since unique images of single bricks dominate our dataset, our model has a tendency to predict Milestone 1 in the absence of any other strong indicators.

Next, we discuss our results when evaluating on the tail set, corresponding to the 1,077 least popular (labeled) images in the tail set. All three models performed poorly: Unit test reported 0% accuracy (due to its pixel-based exact match strategy), and both KNN and our

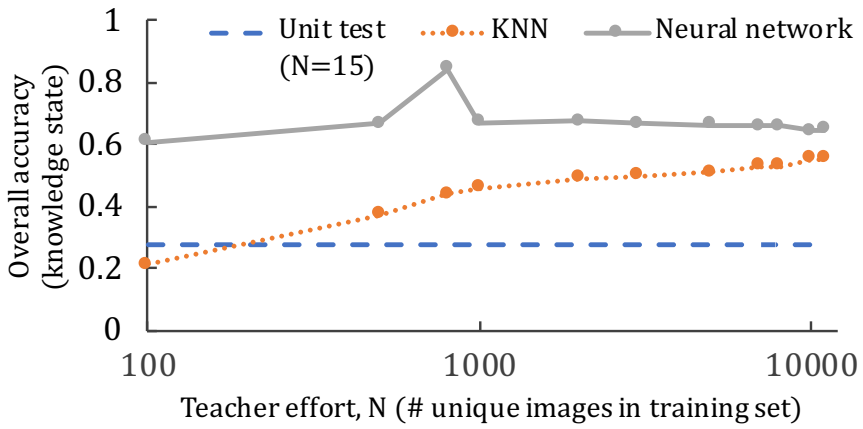


Figure 2.13: Efficiency analysis of model validation accuracy with varying training set sizes.

neural network model performed with less than 1% accuracy for almost all choices of our training set. This is a major area in which future models can improve; the more reliable we are in predicting the tail of our model, the more confident we can be in imputing milestone labels on unlabeled images in our dataset.

Efficiency analysis. Even if graphics-based assignments may generate similar image outputs, a model tuned with transfer learning to the new assignment still requires a newly labeled training set of the new assignment data. The most time-consuming component of characterizing functional progress is not the training time of a machine learning model; it is preparing a labeled training set (Section 2.2). We therefore define the *efficiency* of the method described in this chapter as a function of the labeling effort required by a researcher or expert instructor to prepare training data.

We discuss how the accuracy of each baseline model performs with more efficient labeling —that is, fewer training images labeled—on the full validation set. The training set still consists of the N most popular images, where we vary N from $N = 100$ (most efficient, least effort) to $N = 11,000$ (least efficient, most effort). We then evaluate the performance of the KNN and neural network models with different training set sizes N, and we evaluate their performance on the validation set of the top $11,000$ most popular images. The unit test efficiency is always the same—one image labeled for each milestone.

Our results are shown in Figure 2.13. Naturally, the KNN model performs best with a huge dataset, but what is surprising is that the neural network already outperforms the other two models even with only $N = 100$ items in the training set. There is a small peak at

$N = 800$ for the neural network model; this artifact is because the model correctly predicted the milestones of some high-rank images, therefore weighting the accuracy upwards; this trend disappears when we compare accuracy across unique images only. In general, the takeaway that we can get just over 60% accuracy with just $N = 100$ labeled data points is very promising.

2.3.3 Discussion and future work

The baseline models discussed in this section are first steps in designing classifiers for this task. We find that an $N = 100$ neural network model performs reasonably well on our validation set. We decided to train our models strictly on pixel-based image output by design; given that saving image output for graphics-based assignments is quite easy to implement across different classrooms. Future models could certainly incorporate additional input features to aid this particular task on the Pyramid assignment, such as the coordinates of each object on the canvas, features from a sequence of student snapshots, and preprocessors hand-coded to identify certain milestones.

Another feature of our baseline models is that all unique images were weighted equally during training. As a result, our neural network model was incredibly tuned to detect Milestone 1 (Hello world), which took up most of the unique images in our dataset (Figure 2.9e), but did not do as well when classifying column structures, for example. To improve classification accuracy, we could weight our training loss function by the popularity of each image. The near-dismal results on the tail set also raise concerns about skewed data. Importantly, we note that while many students share common popular images, the tail set is much more diverse: 16% images appear exactly once; 387 students are each responsible for 2.8 images, on average; and 40% of the milestones in this tail set are Off-track (Milestone 15). It is inevitable that we will encounter incredibly rare images in the tail of our set; improving performance on very unpopular images is left to future work.

Given that current and future machine classification models on this task could be impacted by the particular composition of students in our dataset, we propose a way forward to analyzing functional progress: Use a combination of human milestone labeling for the N most popular images—where N is small—and machine-classify the *remaining* images by popularity. We use this approach to visualizing and understanding students in the following section.

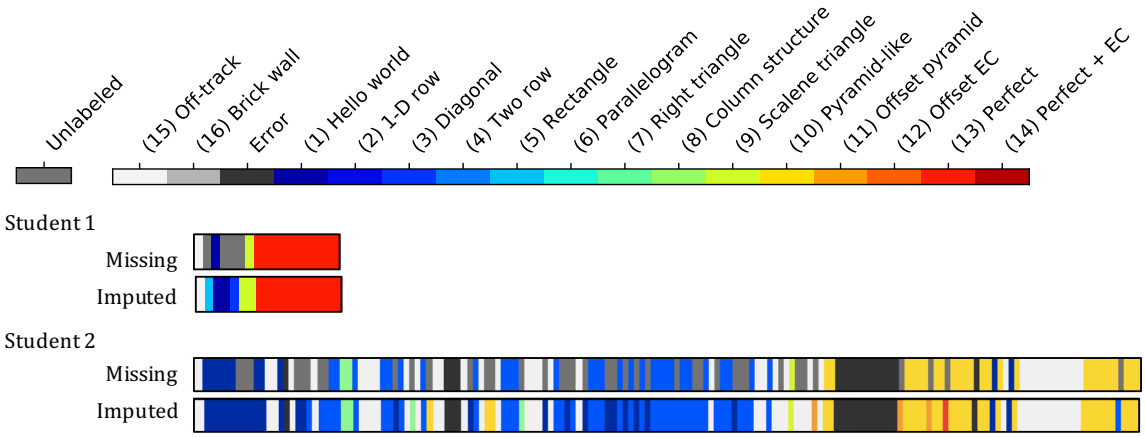


Figure 2.14: A colormap of milestones over two student work trajectories. The missing gray images are imputed with milestones that are similar to the neighboring milestones. Best viewed in color.

2.4 Understanding students

In the previous two sections, we described a method for defining and subsequently classifying functional progress through a labeling abstraction called *milestones*. Next, we describe how visualizing functional progress allows us to glean pedagogical insights about our students.

Our first task is to assign labels on the 17,509 unlabeled image snapshots in our dataset. Figure 2.14 shows this process; we first tag images that were human labeled, and then we follow with using machine labels to impute unlabeled image snapshots. Each colored bar represents a student's *work trajectory* of milestones in their process repository, where each vertical colored strip represents a single snapshot. The top and bottom students in Figure 2.14 represent students who finished quickly and slowly, respectively. The strip's color indicates its milestones, where a colored heatmap corresponds to knowledge stages: Stages 1 and 2 are blue colors, Stage 3 is yellow/green, Stage 4 is red, and all other off-track and error snapshots are in grayscale. We then use this milestone classification to gauge how students move through the Pyramid assignment, both on an individual student basis (in Section 2.4.1) and on an aggregate classroom level (in Section 2.4.2).

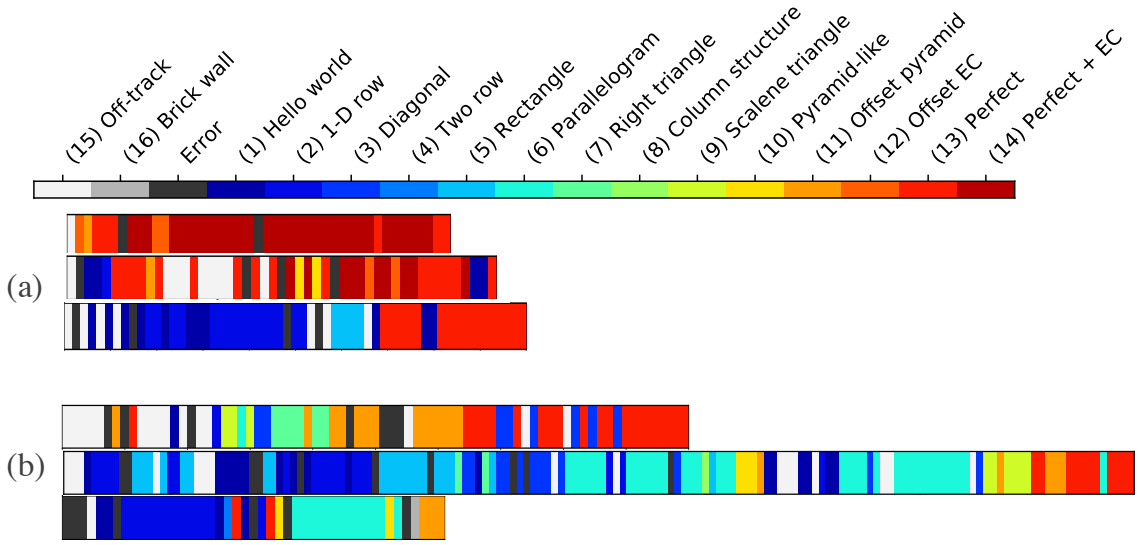


Figure 2.15: Student work trajectories during the Pyramid assignment. Two groups of three students, respectively scoring in the (a) 99th percentile, and (b) 3rd percentile or lower on the midterm exam. Best viewed in color.

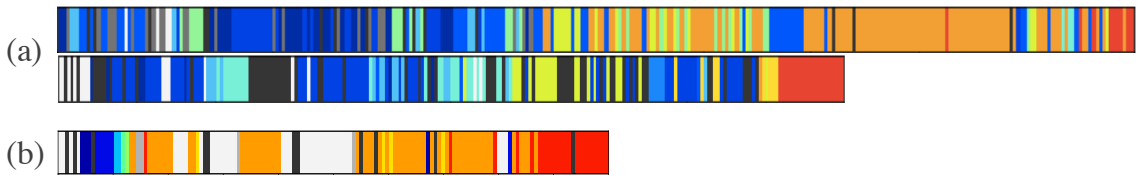


Figure 2.16: Three students with long work trajectories. (a) Struggling students; (b) Tinkering student. Best viewed in color.

2.4.1 Student work trajectories

After imputing the milestones on the remaining unlabeled snapshots, we found some very telling examples of how different students work on the assignment over time. When we compare high-performing students (Figure 2.15a) with low-performing students (Figure 2.15b), we observe that these groups of students tend to concentrate their work in different stages. All three high-performing students have few snapshots in Stages 2 and 3, instead spending a significant portion of their time working on the perfect pyramid (Milestone 13). In contrast, the low-performing students spend a large fraction of time in Stage 3's early milestones that emphasize brick offset adjustments. One student also fails to reach the correct solution, ending instead in the offset pyramid milestone (Milestone 11).

From our data, we can also visually discern between certain students who were struggling to get anything working and those who were tinkering [23, 162]—where a student spends a long time at a particular knowledge stage not because they are stuck, but because they are exploring the solution space. Figure 2.16 shows three students that have used over 100 snapshots for the Pyramid assignment. The first two students (Figure 2.16a) spend a large portion of their time working in Stage 1. In contrast, the last student (Figure 2.16b) spends most of their time in a late Stage 3 milestone, the offset pyramid (Milestone 11), suggesting some sort of tinkering and adjustment. When we connect these work trajectories with these students' performance on the midterm, the first two students score in the 19th and 21st percentile, respectively, while the third student scores in the 73rd percentile. While previous research has found that tinkering is valuable to student learning, we can now visualize tinkering in the context of functional progress.

Observing individual student performance is valuable during a course; instructors can identify and give constructive feedback to different students. However, there is also value to observing aggregate functional progress on an assignment in the classroom. Instructors who identify that most of their students spend time on a learning concept during the assignment can address or highlight those concepts as review. Alternatively, if instructors detect their students are struggling with portions of the assignment unrelated to core learning concepts, then they can revise and improve the assignment specifications for a future course offering.

2.4.2 Aggregate student work

We use our milestone information to gauge how much time a student spends on average in

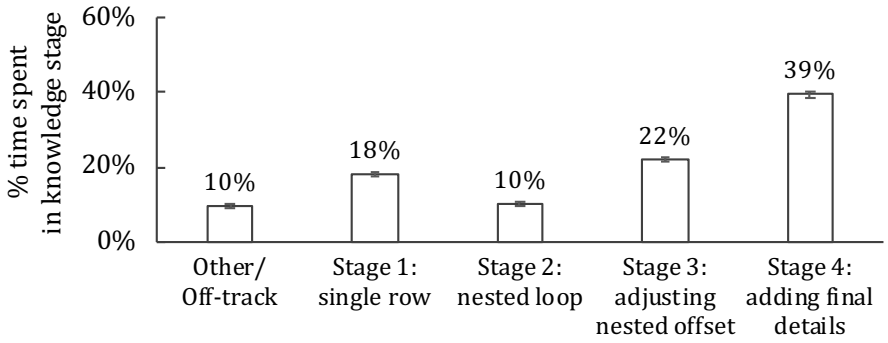


Figure 2.17: Average amount of assignment (by number of snapshots) that students spent in different knowledge stages.

each of our knowledge stages to decide which content to emphasize in classroom teaching. Figure 2.17 graphs the average percentage of snapshots spent in each of the knowledge stages, where we use our neural network predictor to impute the knowledge stages of the unlabeled fraction of the dataset. We consider Stages 1 through 4 as monotonically increasing phases of the assignment; i.e., as soon as a student shows work in Stage 2, they have left Stage 1 and cannot return. Students can freely move between the Other/Off-track knowledge stage and the other stages; we did not graph the 24% of snapshots that students spend on average in compile/runtime errors. We observe that on average, students spend 22% of their time in Stage 3, where they must manipulate the loop index to correctly offset blocks in different rows of the Pyramid. Understandably, students spend most of their time (39%) adding final details and finishing up (Stage 4). Knowing this time distribution can inform which concepts students struggle with; after this analysis, instructors devoted more time in class to discussing loop index manipulation skills needed for Stage 3.

2.5 Image classification for grading

Given that a large portion of our milestone classification method depends on human expert effort to create a labeled dataset, we also want to understand how well image classifiers would perform for a dataset whose labels already exist. As an aside, we thus explored how to design an autograder that strictly uses an image-based, deep-learning approach to grade student final submissions. From anecdotal experience, graphics assignments tend to be more difficult and time-consuming to grade than their command-line counterparts, mainly

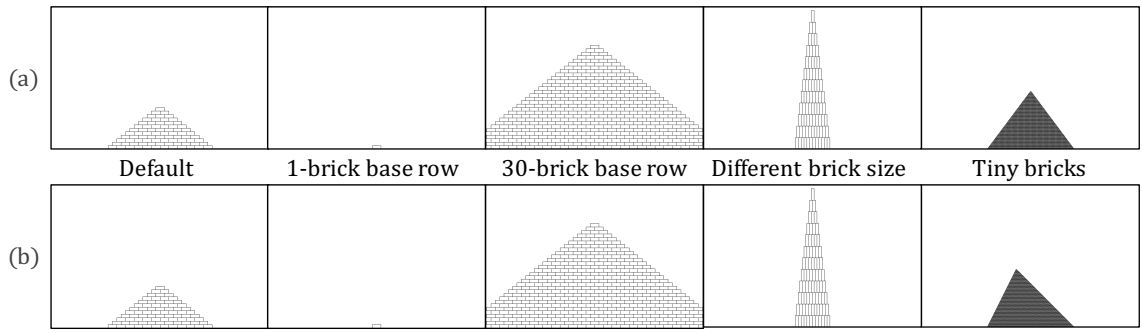


Figure 2.18: Running five different pyramid configurations on two final student submissions. (a) Student correctly draws all pyramids. (b) Student fails to convert to floating-point, resulting in round-off error in the fifth pyramid.

because the set of possible correct graphical outputs is often large. The CS1 course studied thus historically has used human graders to decide the functionality grading component of the Pyramid assignment. In this section, we discuss how these autograders fall short for two main reasons: human grading labels are unreliable, and rubrics for grading functionality in introductory CS1 encode more than just functionality.

2.5.1 Data

We prepared a second dataset, the PyramidFinal dataset, which contained 4,383 students tagged with assignment grades across eight course offerings from 2009 to 2017. A student's Pyramid assignment grade is the number of correct rubric items awarded by a grader. The course assistant analyzes the student program code and its output on five different, predetermined configurations of the pyramid and marks the student on a set of 9 independent rubric items. The grader analyzes the student program code and its output on five predetermined sets of assignment parameters which vary the Pyramid's dimensions. The grader then evaluates the student based on a rubric of nine independent items, where each item is binary pass/fail; we represent this rubric marking as a *rubric vector*. Figure 2.18 shows two different student program outputs on each of the five different Pyramid configurations. Figure 2.18b depicts the most common error for this assignment, which is that lack of floating point arithmetic occasionally yields a skewed Pyramid. The rubric item descriptions were consistent across all 8 course offerings.

While the human course assistant has access to the code of each student's final submission, we evaluate an autograder who can only access image output. However, it is insufficient

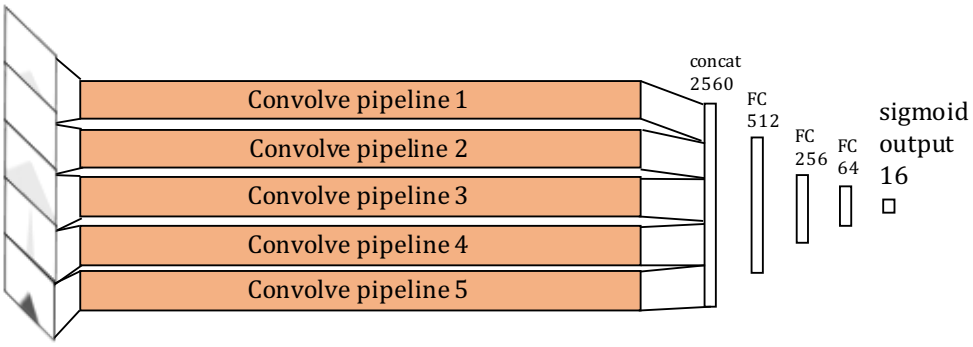


Figure 2.19: The neural network for rubric classification on final submissions.

to use just a single image per student. As an example, of the 2,799 student solutions that draw a perfect pyramid on all the default correct Pyramid configuration (Figure 2.1a), only 1,990 students receive full credit; the remaining 809 students are marked off in 82 different ways, with some solutions having up to five errors.

We therefore run each student submission five times on the five pre-determined Pyramid configurations used in assignment grading. The spread of students with the correct five-tuple of pyramid configurations is much smaller: of the 1,485 students that share the image five-tuple of a fully correct solution, only 85 students are marked off in 20 possible ways (up to three errors). However, since there is still variation in the rubric vector, for our training task we take the most common rubric vector for each unique five-tuple of images. Unlike the PyramidSnapshot dataset, we save only the canvas of the image—with no extra border—to mirror what a human grader would observe; each image thus has the default canvas size W754 $\times$ H492. Each of the samples in our PyramidFinal dataset therefore is a tuple of 5 image outputs and a boolean vector marked with the most common rubric vector. We split our dataset into training and test sets of size 3,945 students and 438 students, respectively.

2.5.2 Method

We compare two approaches to image-based autograding. The first is a unit-test-based model. For each rubric item, we identify a representative subset of pyramid configuration images that would lead to an *incorrect* rubric item. To mark this rubric item as incorrect for a student's five-tuple of images, if there is an exact match (pixel-wise) between the student's images and the representative subset, then we mark the student as having that rubric incorrect. There are 7 such subsets of representative images; two of the rubrics could

	Unit test image		Neural network	
	F1	Acc	F1	Acc
Train	0.15	0.95	0.46	0.97
Test	0.15	0.95	0.40	0.97

Table 2.4: Average accuracy and F1 scores when predicting rubrics on final submissions.

	Rubric item	Frequency	F1	Accuracy
1	Incorrectly indexes outer loop	.03(.02)	.00(.09)	.97(.98)
2	Incorrectly indexes inner loop	.02(.02)	.00(.09)	.98(.98)
3	Miscalculates the pyramid's x-position	.11(.11)	.79(.73)	.96(.94)
4	Miscalculates the pyramid's y-position	.12(.11)	.83(.85)	.96(.97)
5	1-brick base row unit-test fails	.03(.04)	.42(.47)	.97(.97)
6	30-brick base row unit-test fails	.01(.02)	.00(.24)	.98(.98)
7	Different brick size unit-test fails	.01(.01)	.00(.14)	.99(.99)
8	Tiny bricks unit-test fails	.14(.14)	.88(.88)	.97(.97)
9	Fails to convert to double	.08(.07)	.67(.69)	.95(.96)

Table 2.5: Neural network model performance for predicting final rubric items on training set (in parentheses) and test set.

not be paired with representative images, which we discuss in the following section.

Our second model implements a separate convolve pipeline for each image, combining the representations with an affine transformation prior to the fully connected layers (Figure 2.19). Our last layer uses a sigmoid activation instead of softmax, because each image can have multiple rubrics tagged (e.g., a pyramid can be both off-center horizontally and have arithmetic round-off error). The model optimizes the sigmoid cross-entropy loss function; after grayscaling and downsampling (again by 2) for each image in our five-tuple, we have an input with dimension $5 \times 246 \times 377 \times 1$, resulting in a model size of 5.1 million parameters. We keep all $3,945$ students as separate datapoints in our training set, instead of keeping only unique samples as in the neural network model for milestone classification.

2.5.3 Evaluation and discussion

We report in Table 2.4 our results of running our two final submission grading models. Both models have very high average accuracies; however, this is because very few students get marked off for any of the 9 items in the rubric vector (the average frequency of incorrect

rubric items was 10%). We therefore prioritize the F1 score metric, which is a harmonic mean of the precision and recall scores (measuring false positive and false negatives).

We notice that F1 scores for both models are quite low. For the unit test model, we were unable to create representative, incorrect subsets of images for rubric items 1 and 2. Both of these rubric items assess code-based student misconceptions; it is impossible for our models to distinguish whether a nested loop indexing error occurs in the inner loop or outer loop of student code by just looking at image output. The neural network model's F1 scores and accuracies for each rubric item are reported in Table 2.5. The model performs poorly for rubric items 1 and 2, possibly for similar reasons as the unit test model. For rubric items 6 and 7, we hypothesize that a dearth of examples for these rubric items in our training set led to poor performance.

The results from this section point to a broader question of how to design grading rubrics for introductory CS1 assignments. The Pyramid grading rubric discussed in this assignment is designed to assess a student's proficiency in the Pyramid assignment as well as their mastery of programming principles, and as a result some rubric items conflate incorrect functionality with incorrect programming. Yet it is essential to give students feedback *beyond* functionality at this stage in their CS career. Nevertheless, the variability of human grading on this task remains a concern. In the CS course studied, human graders follow a handout that carefully outlines how to interpret incorrect images of different pyramid configurations, but ultimately the students' marks are subject to grader discretion. We expect that these factors would be mitigated in more advanced CS courses—whose functionality grading rubrics would leave programming quirks to a separate style grading rubric—and designing image-based autograders for assignments in these courses is left for future work.

2.6 Summary

This chapter's analysis of the PyramidSnapshot dataset shows that characterizing functional progress is feasible *and* useful. Labeling intermediate snapshots for any assignment is a human, time-intensive task; we therefore design our milestone labeling method to minimize instructor labeling effort, with the hope that a low bar would encourage instructors to prepare new assignment data for functional progress analysis. In our study, our milestone labeling method hand-labels a small set of popular images, then uses image classifiers to extend milestone labeling to the rest of a dataset of snapshots of intermediate student work.

We found that a baseline neural network classifier trained on the 100 most popular images performs relatively well, suggesting that our original task is not as daunting as it may seem. Yet perhaps our biggest contribution is publishing the labeled PyramidSnapshot dataset as an annotated benchmark for classifying functional progress. We hope that future research will expand on our findings by improving machine prediction of functional labels of progress and by analyzing student work patterns in greater detail.

We explore initial directions for using milestones to visualize and compare student work patterns. The link between work pattern and performance is not new, but our work is the first to show the relationship with *functional progress*. In the future, functional labels for student data snapshots can be used in conjunction with existing indicators of progress, like error messages and code length to better predict student performance [22, 23, 79]. A reasonable next step for this work is to better understand the interplay between error resolution and functional progress [16, 79, 170]. We can imagine that such analysis can also be performed on both micro and macro scales. For the individual student, we can understand how different groups of students resolve errors, and when these errors are more likely to occur. On a classroom level, we can better understand at which stage in the problem students are encountering the most errors, and then use this information to better guide how we teach debugging in other facets of the course.

In the classrooms of tomorrow, instructors will have a deeper understanding of how students perform unsupervised work. This work is a first step in sparking research to better understand the learning process, from which future findings will undoubtedly translate quickly to classroom practice. We hope that instructors are inspired by our work to generate functional progress labels for their own students, so that as researchers we can ensure that our analysis reaches as many classrooms as possible.

Chapter 3

Pensieve

Assignment feedback is a critical component of student learning [54, 133]. Given that one of the primary learning goals of CS1 is to teach students *how* to solve programming challenges, it would be immensely useful to provide feedback on *how* a student worked through solving an assignment. Such *formative feedback*—suggestions for how students can improve their problem-solving skills in the future—would help students take control of their own learning [21, 32, 133, 117]. Yet for a variety of reasons, many contemporary classrooms provide only *summative feedback*—feedback evaluating the correctness of a student's final answer [84, 164]. Hours of unsupervised work, during which a student actively learns and interacts with the material, are manifested in a single deliverable—a single snapshot of the student's thinking—from which an instructor must glean enough information to understand learning process. The current classroom model of evaluating only a student's final submission misses the opportunity to give students feedback on their problem-solving process.

However, providing feedback on a final student submission is already challenging, and providing feedback on the hundreds of steps a student may take to get to their answer seems prohibitively difficult. While the previous chapter covered a methodology for characterizing functionality during the learning process, its seamless integration into a CS classroom is still many years into the future. This chapter introduces Pensieve, a simple-to-use tool that can be used immediately in classrooms today to facilitate human conversations about how students progress through a programming assignment.

Pensieve[1] is an interactive visualization of student work history on an assignment and

[1] In the well-known Harry Potter franchise, the Pensieve is a magical tool used to save and examine a user's past memories.

can be used to give feedback on a student's problem-solving approach. After a student completes their assignment, an instructor can use Pensieve to review the student's learning process and give feedback to the student in person. Pensieve gives both students and teachers a means to see progress on an assignment—from the time a student first looked at the assignment starter code to when they submitted the final product—thereby facilitating conversation around metacognition and student learning that is critical for introductory computer science learners. When teachers can observe a student's process, they are able to give timely feedback addressing student mistakes and to adjust and personalize their own teaching [27]. When students observe their own process—even in the absence of the instructor—they can internalize metacognitive observations [32]. Through our streamlined user interface, understanding progress becomes quick and feasible, even in a large classroom.

In this chapter, we begin with an overview of the pedagogical motivations for designing the Pensieve tool, as well as existing tools for giving feedback to CS students. After introducing the technical implementation details of the tool, we share our experiences using Pensieve in a 10-week CS1 course. We close with discussion on best practices to deploy our tool quickly and effectively in other CS classrooms. To speed up adoption of our tool, as part of this work we released Pensieve as an open source tool that both students and instructors can use.[2]

3.1 Pedagogy and motivation

Pensieve aims to "push back" against the trend of automated grading tools in classrooms by thoughtfully integrating a human grader into assignment feedback. We designed the tool with several objectives in mind:

1. Foster metacognitive skills
2. Identify methodology errors early
3. Counteract plagiarism effects in a large classroom
4. Gentle introduction of version control.

Our first objective is to improve metacognitive education in computer science. *Metacognition* is a learning theory for "thinking about thinking;" to most effectively learn, students

[2]The Pensieve tool is available at `https://github.com/chrispiech/pensieve`.

must not only understand the problem but also understand and reflect on where they are in the problem-solving process. A key finding of a landmark National Academy of Sciences study on learning science was the effectiveness of a metacognitive approach to instruction [28]. In educational theory, Bloom's Revised Taxonomy characterizes metacognition as one of the highest cognitive knowledge dimensions for any learning activity [13, 58, 161]. Research has also shown that students with stronger metacognitive awareness tend to perform better on programming tasks [18, 39, 52]. Importantly, metacognition directly supports the concept of a *growth mindset*, the theory that intelligence can be developed with experience [48]. Students who believe that ability is a fixed trait are at a significant disadvantage in STEM fields compared to their peers who believe in a growth mindset.

Our second objective is to encourage early identification of methodology errors. Most assignments in large-scale computer science classes are assessed summatively: the teaching assistant sees and grades only the students' final submission. However, research has conclusively shown that nongraded *formative assessments* are key to improved learning [28, 83]. They allow teachers to identify which thought processes work for students and to provide useful, directed feedback so that each student can improve. As such, this goal is intertwined with our goal of fostering metacognition in computer science—through reflecting critically on their programming process, students will be able to both identify areas for improvement and understand how to achieve that improvement earlier.

Thirdly, by emphasizing the importance of the learning process, we hope that Pensieve can act as a preemptive deterrent to potential plagiarism. Students who plagiarize all or parts of their assignments stunt their metacognitive development in programming and reap fewer benefits from formative feedback. They may also become stuck in a cycle of plagiarism in which they are increasingly unable to complete work independently [163]. Many current approaches to combating plagiarism in large CS classrooms focus on detecting similar code in the final submission; however, this further reinforces the importance of the final grade received above the intrinsic value of learning [29]. We hope that by monitoring the development process [163], we can better support students who may otherwise feel overwhelmed.

Lastly, we deploy a light version of Pensieve for students to facilitate project version control. In the absence of instructor feedback, Pensieve may still be useful to students; while a student works, they can use Pensieve to browse and restore previous snapshots of code. By presenting software version control as accessible and useful in introductory CS

courses, we hope that students can quickly adapt to more heavyweight version control tools like Git and Maven in more advanced courses.

3.2 Related work

While existing work research addresses the four objectives of Pensieve, to our knowledge our work is the first to combine all four objectives in a tool that can be adapted easily to existing CS1 classrooms.

Novice programmers benefit from metacognitive awareness. Lee et al. found that personifying the programming process increases online engagement with a coding task [96] and Marceau et al. studied how novice programmers interact with error messages [103]. In general education, Tanner et al. reported on generalizable teaching practices for promoting metacognition, such as instructor modeling of problem solving, tools to help students identify learning strategies, and guided reflection [160]. Pensieve incorporates all of these strategies by engaging students—with the support of teaching assistants—in their own metacognitive understanding of computer science. Loksa et al. found that students who are trained in problem solving procedures are more productive and have higher self-efficacy [100]. However, the intervention was studied in a camp setting, where instructors explicitly discussed problem-solving strategies with individual students as they worked on the assignment. It would be intractable to apply this tool to an unsupervised work context in a university CS classroom, where student-teacher interaction during the working period is more infrequent. Cao et al. proposed Idea Garden, a tool that automatically suggests problem-solving procedures within the student's IDE. While this tool is autonomous, it is still labor-intensive for the instructor, who must correctly anticipate and articulate suggestions that address potential obstacles. This suggestion procedure may fail to adequately support many open-ended assignments, in part because poorly constructed instructor hints could discourage students from exploring different solution paths.

Formative feedback in the classroom can have a variety of impacts [17]. Van der Kleij et al. found that detailed feedback contributes more to student learning than feedback on correctness does [164]. Students also benefit more from an assignment when they have interactive, dialogue-based critiques with peers or instructors [32, 117]. We design our tool with the awareness that feedback is not a one-sided conversation; teachers should hold discussions with the students to improve student learning for the rest of the course [27].

While many tools automate summative assignment feedback, Pensieve seeks to augment such tools with formative feedback generated by human instructors. In many large classrooms, automated assessment tools (AATs) use teacher- or student-written tests to generate assignment feedback [10, 49, 75]. Larger classrooms like massive open online courses push towards full-automation by using systems like intelligent tutors to personalize the learning experience for online students [42]. However, Prather et al. found that novice programmers struggle with interpreting results from AATs, which are not designed to explicitly support student metacognition. There have been studies on how to design automated computer agents for personalizing debugging hints explicitly for CS1 curricula [139]. Nevertheless, to our knowledge there is no automated tool for providing formative feedback. Pensieve leverages the in-person peer instructor system widely used in many large CS1 courses [56] to provide human feedback on higher-level cognitive tasks to teach students beyond the current assignment.

Some classrooms have used version control systems to give formative feedback, as version control simplifies management of large courses [37] and makes it easier to identify problems in work habits and progress [89, 92, 137]. We take a slightly different approach, placing less emphasis on learning *how* to manage a version control system than on encouraging student self-evaluation [99]. Our tool focuses on surfacing this information in a clear manner, providing a light introduction to the benefits of version control.

3.3 Pensieve details

Pensieve is primarily designed for educators to easily give feedback on the learning process. We minimize instructor workload by designing Pensieve as a drag-and-drop, out-of-the-box tool for viewing student assignment progress. It is ported as a JAR file; an instructor can simply download the JAR into a student assignment folder, run it, and begin viewing student intermediate snapshot data. In this section, we first explain the components of the Pensieve tool shown in Figure 3.1. We then discuss in detail a classroom pipeline for using Pensieve in a one-week assignment.

3.3.1 Tool implementation

The only required folder for Pensieve to work is a process repository of timestamped code snapshots of a single student working on an assignment—data which is increasingly available

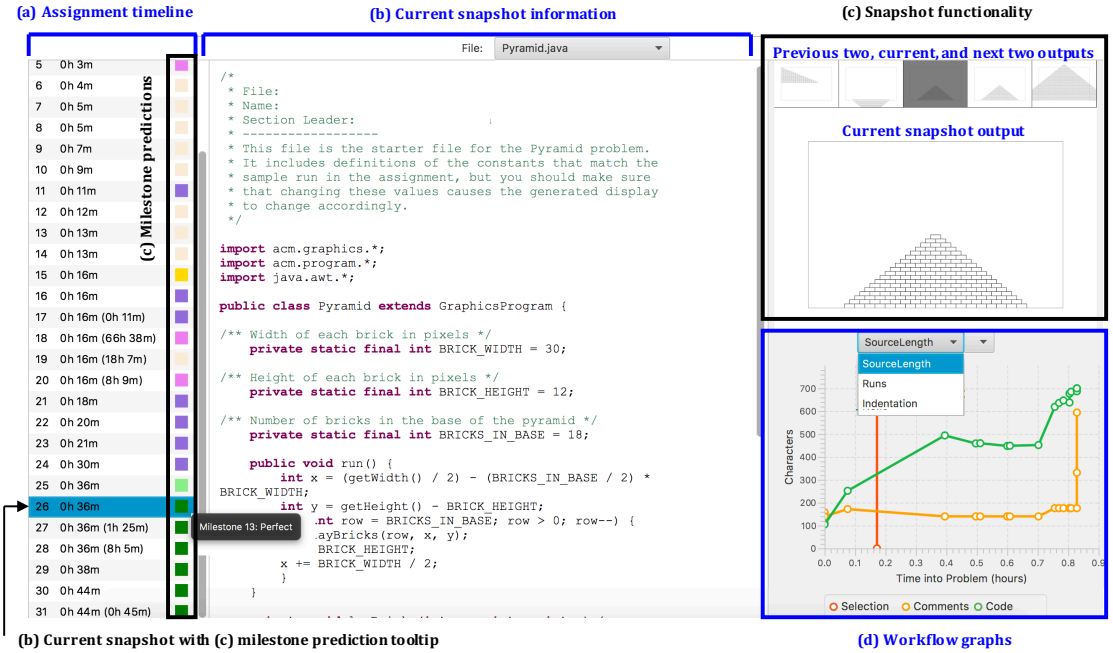


Figure 3.1: Diagram of Pensieve display, composed of four main components: (a) Assignment timeline, left; (b) Current snapshot information, center; (c) Snapshot functionality, top right; and (d) Workflow graphs, bottom right.

to educators [132]. In our case study, we use Git to timestamp and save snapshots of Java code in the Eclipse IDE, as described in Section 2.2. However, Pensieve depends neither on the use of Eclipse nor Java in the classroom; it merely requires that student work be cached over time in a timestamped code folder. Our tool can work at a coarser granularity, such as on assignments that require students to commit periodically to an online storage platform like GitHub. The tool can also operate on setups like those on Code.org, which save snapshots whenever students run their code. We utilize the timestamped code repository to analyze both student code progress and timing information.

Pensieve visualizes snapshots within the process repository, as shown in Figure 3.1. A user can (a) view a timeline of progress on the assignment, (b) select a current snapshot to analyze in more detail, (c) identify functional progress on the current snapshot, and (d) access metrics of code changes over time. We discuss each of these components in more detail below.

Assignment timeline

The leftmost panel of the tool (Figure 3.1a) displays all captured snapshots for a given student file. Each entry in the timeline contains the snapshot index (in temporal order), the amount of time spent so far on this file, break time, and an optional color key indicating functional progress (discussed more below). Time spent on the assignment is calculated as an aggregate of relative timing information between this snapshot and the previous one. We mark break time in parentheses, where a break occurs when two consecutive snapshots' timestamps differ by more than 10 minutes. By browsing the timeline, an instructor can infer where students took substantial breaks to get a cursory glance of which snapshots would be worth closer attention.

Current snapshot information

Selecting a snapshot in the assignment timeline list updates the displays in current snapshot information, snapshot functionality, and the red line in the workflow metrics graph (Figure 3.1b). The current snapshot's code in the center panel has appropriate syntax highlighting and is in plain-text, allowing an instructor to select and copy code to an editor if additional verification is needed. The top-right panel is used to display this code's output when compiled and run; if the code has a compile or runtime error, nothing is shown. The example assignment shown in Figure 3.1 draws a pyramid from the Pyramid assignment described in Chapter 2. It is important to note that Pensive is not running snapshot code live; the compiling and running of any student code is done in a preprocessing step. The details of designing such a preprocessor are discussed more below.

Snapshot functionality

The top-right panel (Figure 3.1c) visualizes the output of the currently selected code, if there were no compile or runtime errors. All outputs for all snapshots are run and saved prior to downloading and running Pensieve in a preprocessing step. In our implementation, this occurs on the course servers after students submit the assignment and before instructors begin grading. The preprocessor compiles and run each snapshot in each file; if there are no errors, then the output is saved into a separate folder. The Pensieve program then interfaces with the folder of outputs via a JSON metadata lookup, which associates the snapshot (represented by a (Unix Epoch timestamp, original filename) tuple) with the

following features: a flag for compile error, a flag for runtime error, the output file path, and a milestone metric, of which the latter two are only valid if both error flags are off. Any additional metadata per snapshot can also be saved in this JSON lookup file.

In our example assignment, we mainly use graphics exercises based on the Java ACM graphics library [142]. As a result, output file paths point to saved output PNG files. However, one could easily run and save console program output as plain-text log files, which can be displayed in the top-right panel of Pensieve with minor modifications. Milestone metrics are computed based on each snapshot's output; if there are unit tests for the assignment, this can simply be the number of passed unit tests. For our image files, we use an image classifier that determines a milestone label for each snapshot, as described in Chapter 2. The milestone metric per snapshot is displayed as a small square color indicator in the left timeline panel; runtime and compile errors are assigned separate color indicators.

Workflow graphs

The bottom-right panel (Figure 3.1d) contains time series of various file metrics to visualize student progress and style over time [22]. The SourceLength graph shown displays code length (in green) and comment length (in yellow) in number of characters, as well as a red, vertical indicator of the currently selected snapshot's metrics. Another graph displays indentation errors over time—which are allowed in Java but are indicative of messy coding style—and a third graph displays custom metadata from the timestamped code directory; for example, our process repositories also record the number of code runs per snapshot. Instructors can easily toggle between these time series, which are intended to highlight student work patterns, such as when they started thinking about good style and indentation, or whether the majority of their work and code changes were concentrated at the end of the timeline.

3.3.2 Classroom use

Figure 3.2 shows an assignment feedback pipeline that uses Pensieve. First, a student works in their own environment (a lab computer, personal laptop, or in-browser app), which saves timestamped snapshots of their code progress. The student then submits their work to a database, where optional preprocessing occurs; for example, to generate and save snapshot output for later review. Third, the teacher downloads and reviews the student code submissions using Pensieve. Finally, the teacher discusses the code with the student

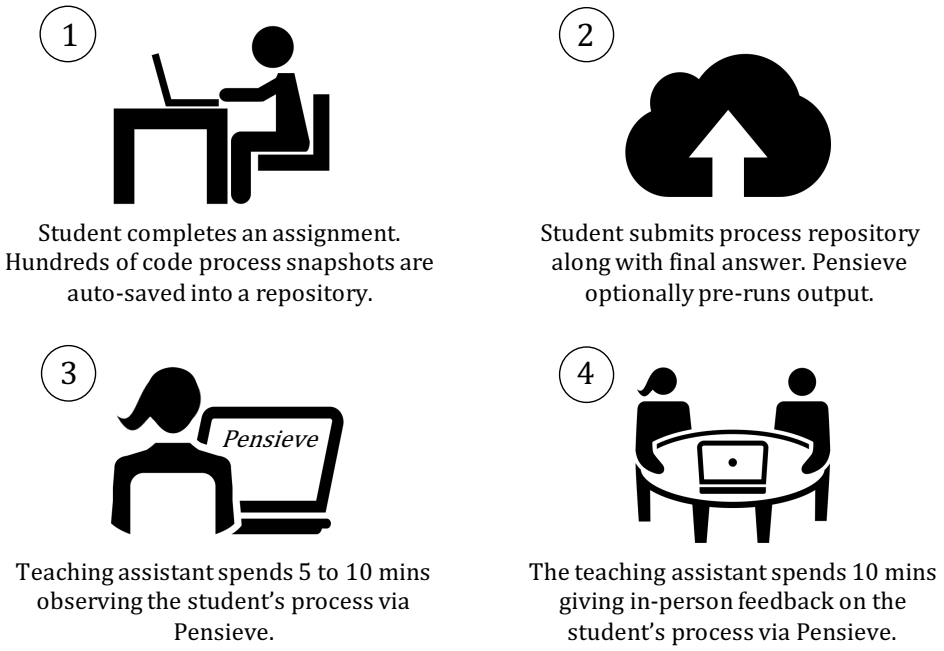


Figure 3.2: Enabling student-teacher conversations on learning process with Pensieve.

in a classroom setting. Because Pensieve is a tool designed for formative feedback, the in-person feedback on learning process is *ungraded*. To prevent outcomes of the student-teacher conversation from influencing any component of grading, autograders or human grading should happen prior to instructor use of Pensieve (i.e., before step 3).

Teachers in our classroom provide assignment feedback to their students in two ways: the functional and style grades produced from a fixed rubric, and an ungraded *interactive grading* (IG) session [143], where the teacher and student sit down for a one-on-one, 15-minute session to discuss the student's code, any misconceptions, and tips to improve for future assignments. The purpose of the IG session is to maintain conversation throughout the quarter so that students can actively reflect on their coding process. We therefore insert Pensieve into the IG session (step 4 of the pipeline in Figure 3.2).

We design Pensieve to quickly integrate into existing classrooms. In our classroom, each teaching assistant (TA) is assigned to eight students for the duration of the quarter. The TA's weekly responsibilities are to teach discussion sections, grade assignment work, and hold IG sessions. They download student solution code from the course database, typically a few days prior to the grading deadline, which is a week after the student submission deadline.

At the time of TA download, Pensieve is already included as part of each student folder. Any preprocessing to generate visual program output or console output logs must be included in the student folders prior to download; this is done to ensure that the grader can run Pensieve as a lightweight tool on their own computers. The preprocessing step to generate and save graphical output on three separate code files for a 400-student class took about half a day. In the absence of a course database or preprocessor, a teacher can simply drop the Pensieve JAR into the assignment directory on the students' computer. Once the JAR is within the correct directory, teachers and students alike can quickly navigate the process repository of timestamped code snapshots and discuss coding tips and misconceptions. If the only added time is the 5-10 minutes to review each of eight students' programming process, then the total anticipated extra time for each individual teacher incurred by Pensieve adoption is 40-80 minutes.

Teachers can use Pensieve during these IG sessions to gain a holistic view of how the student thought through the assignment, and to identify, for example, places where students struggled with concepts or showed good style habits. For the student in Figure 3.1, the teacher could identify from the SourceLength graph that prior to the last two snapshots, the student did not comment at all (the starter code has 170 characters of comments)—in other words, the student began adding comments right before submission. The code source length had a similar jump; upon clicking on later snapshots the teacher could identify the student decomposing their code into smaller helper functions in the last 10 minutes of work. The teacher could then gather that the student was opportunistically commenting and adding helper functions in order to increase the final submission's style score. As part of the IG session, the teacher could suggest that the student use comments and helper function design *during* the assignment to guide problem-solving.

In addition to the teacher-facing version of Pensieve, we also design a reduced version of the tool for students to use as a light version control software. Students are notified of this software during the course; the JAR file is available as part of assignment starter code.

3.4 Experience

In this section, we share our findings on using Pensieve in a CS1 course. At the time of performing this study, we used Pensieve in two terms: Winter 2018, which was used to refine the tool implementation and teacher training procedures in tool usage, and Spring

Agree (%)	Item
90	Insights from Pensieve lead to "more actionable and specific assistance"
70	Pensieve "is helpful for showing instructors which students might need extra help"
–	Data on assignment completion time are "especially valuable"

(a) Positive feedback.

Agree (%)	Item
80	Want an option "to show only major changes in code" to better prioritize information and feedback
75	Want more "student-facing features"
–	"All of the data Pensieve presents is helpful but can be overwhelming"

(b) Points to improve.

Table 3.1: Teacher survey results on using Pensieve.

2018, which we will refer to as the **Experience Term**.

We sought to evaluate the impact of our tool on the Experience Term in three ways: (1) a formal qualitative analysis of how useful instructors found the tool, (2) an official university survey on how useful students found the tool, and (3) quantitative measures of performance on exams, time to complete assignments, and honor code violations. All evaluations reveal a consistent story of educational benefit that we expect to see given the improved pedagogical benefits.

3.4.1 Student and teacher evaluations

At the end of the Experience Term, we asked students to provide a Likert rating for the statement, "It was useful to see the process of how I learned using Pensieve," as part of their formal course evaluation. Students on average strongly agreed ($\mu = 4.6/5, \sigma = 0.6$). Furthermore, students gave a higher rating for the quality of course instruction and feedback ($\mu = 4.8, \sigma = 0.4$, where 5 = Excellent, and 4 = Good) than in previous terms ($\mu = 4.6, \sigma = 0.4$) for the Experience Term instructor. Response rate was 70% for $N = 207$ students in the course.

To measure how useful teaching assistants found our tool, we requested an external facilitator to conduct a small group instructional diagnosis (SGID) [36]. External evaluators met with $M = 31$ TAs who used Pensieve in the Experience Term to discuss (1) whether using the tool improved the teachers' understanding of student learning process in CS, (2) if

	Baseline Term Winter 2017	Experience Term Spring 2018
Students	498	207
Women	51%	52%
$>$ 10 hrs exp.	37.4%	27.7%

Table 3.2: Student demographics in the CS1 course studied.

there were ways to improve the tool, and (3) if there were ways to improve how the tool was used in the classroom. The evaluators solicited feedback and consolidated major sentiments. To minimize bias, the course instructor and researchers were not present during the SGID. The results of the teaching evaluations are reported in Table 3.1. The TAs articulated that they found the tool to be useful, despite it requiring more work on their part. The feedback session suggests that more teacher training or better highlighting within the tool would improve the teacher experience.

3.4.2 Learning analysis

In addition to measuring the perception of students and teachers, we also place importance on quantifiable improvements in learning outcomes. Under our hypothesis that Pensieve had a substantial impact on student learning, we expect to see a notable change in student performance.

In the following analysis, we compare our findings in the Experience Term to those in a Baseline Term (Winter 2017), which did not use Pensieve. Table 3.2 shows the student demographics for each term. The Baseline Term had many more students, and they had significantly *more* CS background than those in the Experience Term; 37.4% of students in the Baseline Term reported $>$ 10 hours of programming experience, versus 27.7% of students in the Experience Term. Both terms had a comparable self-reported gender ratio.

Both the Baseline Term and the Experience Term had the same instructor, lectures, and assignments. We focus on the first half of the course, whose timeline is identical across both terms (Figure 3.3). The "Pensieve intervention" is defined as the Pensieve-assisted IG sessions for Assignments 1 and 2 during the Experience Term. The Baseline Term's corresponding IG sessions did not use the tool, and all conversations were based on the student's final submission.

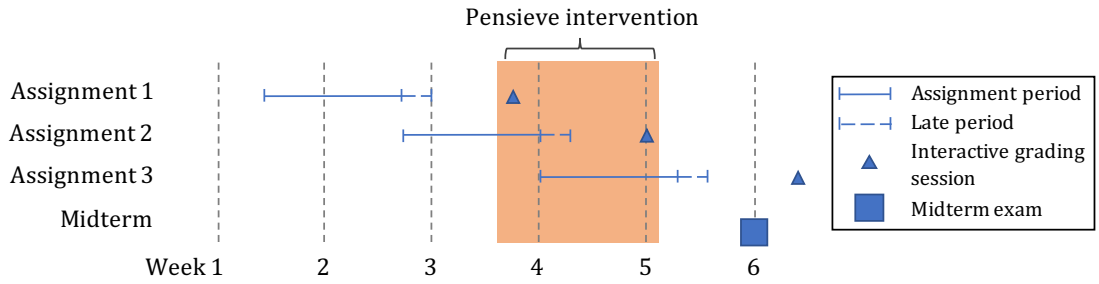
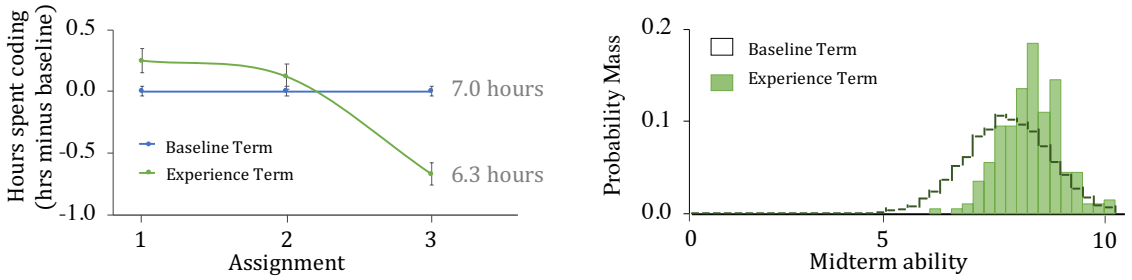


Figure 3.3: Assignment and exam timeline for both the Baseline and Experience terms.



(a) (b)

Figure 3.4: Comparing course performance between the Baseline Term and the Experience Term, where we deployed Pensieve, by (a) assignment completion time and (b) midterm ability.

In the Experience Term, we provided our tool to the entire class—as opposed to running a randomized control trial—and thus it is not possible to report on the causal impact of using Pensieve. While we could not observe causality, we can observe correlation. The co-occurrence of using the tool and notable learning improvement adds weight to our belief that the tool has a positive impact. To assess the co-occurrence of learning gains, we measured changes in (1) time spent on assignments after the Pensieve-assisted interactive grading assignment, (2) exam performance, and (3) plagiarism.

Assignment time

We used student assignment completion time as a pre-post measure of their ability to program. For each of Assignments 1, 2, and 3, we used the timestamps within the process repository to calculate assignment completion time. The Experience Term saw a significant decrease in the number of hours spent on Assignment 3, the assignment due after the

intervention, from an average completion time of 7.0 hours down to 6.3 hours (Figure 3.4a). A possible explanation for this phenomenon is that Assignment 1's IG session, although part of the Pensieve intervention, occurred just before the due date for Assignment 2, and thus Pensieve's impact could not manifest until students began Assignment 3.

This decrease in Assignment 3 completion time is particularly striking given that in the Experience Term, students were slower on average to complete Assignments 1 and 2. To account for the longer Assignment 1 time, for each student in the Experience Term, we compare their Assignment 3 completion time (X_3) with their predicted Assignment 3 completion time had they been in the Baseline Term, given their Assignment 1 completion time ($\hat{X_3}|X_1$). The average difference between these two times is as follows:

$$\begin{aligned}\text{Decrease in Assn 3 time} &= E[\hat{X_3}|X_1] - E[X_3] \\ &= 48 \text{ mins } (p < 0.0001)\end{aligned}$$

We observe an increase in Assignment 3 grades between the Baseline and Experience Term, but the change is not significant ($\delta = 3.2$pp, $p = 0.07$). There was no significant change in grades on Assignment 3, as most students score highly on this assignment ($\mu = 98\%$ in Baseline, $\mu = 101\%$ in Experience).

Exam ability

On its own, the ability to complete an assignment faster does not indicate that students have learned more. Another measure is how well students performed on the class midterm, which follows immediately after Assignment 3 is due. We use Item Response Theory [50] to calculate a midterm "ability" score for each student. Given that we gave different exams in each term, we first evaluated the difficulty of all exam questions on a consistent scale. We then define student i's score on question j as $S_{i,j} = n_j \cdot \sigma(a_i - d_j)$ where $S_{i,j}$ is the score of student i on question j, n_j is the number of points on the question, d_j is the difficulty of the question, a_i is the ability of the student, and $\sigma(\cdot)$ is the sigmoid function. We can then reverse calculate a_i, the ability of student i, given their observed score and problem difficulty. The difference in exam ability between terms is shown in Figure 3.4b. Student midterm abilities, measured on a scale from 0 to 10, increased from an average of 6.9 in the Baseline Term to an average of 7.6 in the Experience Term ($p < 0.0001$).

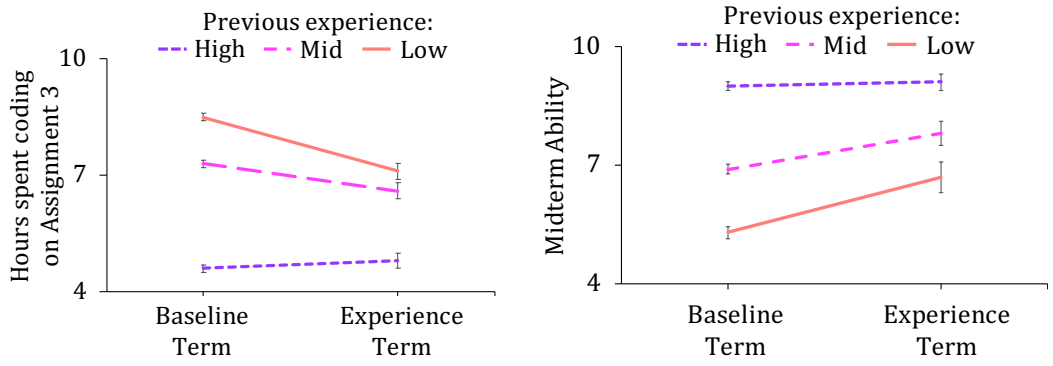


Figure 3.5: Comparison between the Baseline Term and the Experience Term, broken down by students with High, Mid and Low levels of initial CS background: (a) Hours taken to complete Assignment 3 and (b) Midterm ability.

Plagiarism

One of the theorized impacts of using Pensieve is that it would create a culture where plagiarism would be less prevalent: it is much harder to cheat if you are going to be presented with your process. For Assignment 3, trajectory-based plagiarism detection (TMOSS, discussed in Chapter 4) shows a small decrease (from 4.3% of the Baseline Term students down to 3.9% of the Experience Term students), but this statistic is not significant.

Impact on novice programmers

Given that background knowledge outweighs many other factors for predicting student performance in CS1, we disaggregate our assignment time results and exam ability results based on students' prior knowledge. We split students into three equal-sized terciles (Low, Mid, High) based on a background statistic, computed as a weighted sum of normalized: reported hours of experience (50%), Assignment 1 grade (30%) and Assignment 1 work time in hours (20%). Our results comparing Assignment 3 completion time and midterm ability between the Baseline Term and Experience Term are shown in Figure 3.5. We see that the improvements in both metrics are largely experienced by the Low tercile. The Mid tercile also had significant improvements, while the High tercile showed negligible changes

in either metric.

Despite these promising results with Pensieve, there are many uncontrollable, confounding factors. In particular, TAs are not required to use our tool, and it is highly likely that those who effectively use the tool are already more effective teachers to begin with. Instead of taking our analysis as conclusive proof of the efficacy of Pensieve, it is better to see these results as a positive indication that such a tool improves the student learning experience.

3.5 Best practices

Given that we would like Pensieve to work beyond the course studied in this chapter, this section is dedicated to best practices of Pensieve in future classrooms. These tips were gleaned from our own use of the tool and comments from TAs during the SGID sessions.

When to use Pensieve. With the goal of developing student metacognition at an early stage, Pensieve should be deployed for the first few programming projects in a course. This enables teachers to identify process errors early on and recommend good programming practices, as well as develop a student's metacognition.

Pensieve works best with timestamped process repositories; for introductory CS courses, it is best to design a system for collecting frequent snapshots of student work, on the order of minutes. For more advanced CS courses, it is more natural to focus on higher-level problem-solving procedures, and therefore coarser-grained snapshots from a student's self-managed online assignment code repository should be sufficient.

How to use Pensieve. Pensieve can be deployed in many ways. We believe that it is most valuable when teacher and student sit down together and use the tool to facilitate discussion. This type of conversation helps a teacher identify struggling students early on, so that they can be monitored and helped through the course. In our experience, keeping these short sessions ungraded helps to create an environment where students can ask questions about their own learning. Alternatively, the tool could be used for remote feedback, though we expect the missing human conversation will limit the impact on student learning.

Teacher training. If Pensieve will be used in student-teacher interactions, then it is important to train teachers to use this tool effectively. Teachers should be instructed on how to foster conversations with students about problem solving techniques. We specifically advise teachers to talk about top down "decomposition" and "iterative testing" [40]. We

recommend training teachers to use the graphs in Pensieve to efficiently find the most "important" snapshots of progress—i.e., snapshots that indicate a change in problem-solving approach, different milestone work (as in Chapter 2), or a shift in student objective (e.g., moving from functional progress to code style progress, such as commenting or decomposing functions).

In our experience, teachers found the image output of graphics-based programs to be very useful when skimming to understand how students implemented their code. However, Pensieve can also work well (and preprocessing will be faster, with low storage overheard) with console-type programming assignments. For assignments where students implement many small functions, all of which are autograded, autograder unit tests results are a valid measure of snapshot functionality. For these assignments, we suggest that the preprocessor run autograder unit tests and display the results in Pensieve as snapshot functionality.

Establishing culture. Pensieve works best with fine-grained code snapshots in process repositories; naturally, this brings up questions of student privacy. When presenting the tool to students, we clearly disclosed that our intentions were to help students learn as much as possible. Just like in other classes, the more work a student shows, the more helpful feedback a teacher can give. Similarly, when introducing Pensieve to students, it is an opportunity to explain that plagiarism is much more obvious when a teacher has access to what happens during unsupervised work.

3.6 Summary

In this chapter, we presented Pensieve, a tool that gives teachers a window into how students work over the course of a programming assignment. Pensieve is most effective with a human teacher in the loop—a trained teacher can glean learning process information from our tool to give students feedback on programming methodology and problem-solving strategies. Our tool is explicitly designed to enrich the discussion and flow of ideas between teacher and student. There are many potential extensions to Pensieve that can reduce the time that a teacher needs to understand student unsupervised work, but our initial study has already received overwhelmingly positive feedback by both students and teachers alike.

Perhaps the most important contribution of Pensieve is that the tool can provide more formative, more effective feedback to students. Despite the growing size of undergraduate classrooms, we believe that using tools like Pensieve in student-teacher discussions are

worthwhile investments for the time-constrained instructor. Given the amount of time needed to learn how to code, giving novice programmers formative feedback *early* can be a highly efficient use of scarce human resources. If students are encouraged to reflect on their process, they improve their metacognition and can become more effective, self-regulated learners in their future endeavors.

Chapter 4

TMOSS

In the past decade, an increasing number of educators have begun digitizing and transforming the learning experience in order to meet the ever-rising demand for education. This phenomenon is especially apparent in CS education at the undergraduate level, where the nature of the learning material enables efficient use of computing resources. Not only can the content be delivered via online lectures, but an entire homework assignment can also be delivered, submitted, and graded with the aid of autograders.

Due to the size of the CS1 class at many universities today, it is intractable to monitor every student as they progress through an assignment, even with the use of undergraduate or graduate TAs [56, 147]. Instead of each teacher quickly identifying who needs help, it is more common that struggling students must seek out instructor help through scheduled office hours. By asking students to take on the responsibility of seeking help, a pressing issue arises. Sometimes, the students who struggle the most will not access the provided learning environments and will instead find their own resources; seeking shortcuts to learning, they may occasionally resort to unpermitted outside resources. This is especially prevalent in large online courses [184].

Two questions arise from the current state of large CS1 classes: Given that there is a risk of *excessive collaboration*, where students overly rely on outside resources like peer code or online solutions, how can we identify students who exhibit such behavior during unsupervised work on assignments? Furthermore, how does excessive collaboration correlate with student assignment work patterns and overall course performance? If we can address these two questions, we can learn more about our students, detect students exhibiting excessive collaboration, and ensure a healthy learning environment.

In this chapter we tackle both of these problems by introducing Temporal Measure of Software Similarity (TMOSS), based on the well-known Measure of Software Similarity (MOSS) system [148]. TMOSS[1] is a tool that builds on traditional software *similarity score* measures like MOSS. Instead of reinventing the wheel of software similarity detection, TMOSS uses existing software similarity tools to detect excessive collaboration over the entire duration of unsupervised work, not just on a student's final submission.

After summarizing related work in Section 4.1, we describe our dataset of unsupervised work on the Breakout assignment in Section 4.2. Section 4.3 introduces the TMOSS tool, which uses student process repositories to compute similarity scores; these summaries are then verified by a human to hypothesize which students may have exhibited excessive collaboration. Section 4.4 gives a theoretical framework for interpreting similarity scores and show that the similarity scores of regular students can be modeled with a parametric Gumbel distribution. Finally, in Section 4.5 we report our results using TMOSS on our dataset, and we find that students who exhibit excessive collaboration spend significantly less time on their assignment, use fewer class tutoring resources, and perform worse on exams than their peers.

4.1 Related work

Many software similarity systems have been developed over the past two decades to detect for software plagiarism. Some MOOCs use biometrics measures like keystroke logging to authenticate students at the beginning of a work session [91, 101]. In university CS courses, however, students often have multiple sessions of unsupervised work, where they work from an offline IDE and submit online at the conclusion of an assignment. A larger portion of classroom software similarity detectors detect plagiarism on the final version of code that students submit [34, 59, 129, 131, 148, 172, 175]. These tools can be used in conjunction with one another, e.g. as features in a machine learning detector [51]. However, these tools may report many false negatives, as they cannot detect excessive collaboration that happens *during* the assignment.

Online learning and teaching CS courses at scale have prompted new research on modeling and understanding student learning from programming assignment solutions. Spohrer et al. [156] observe and collect student progress based on observation of student programming

[1]The TMOSS tool is available at `https://github.com/yanlisa/tmoss`.

bugs, whereas Piech et al. [127] use assignment progress repositories to analyze intermediate student work. Social science research in the past has attempted to explain motivations for student plagiarism, citing that plagiarism is most common when there are small penalties and high rewards [31, 104].

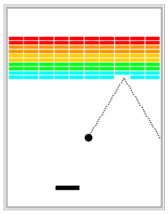
Recent work has combined these two areas of research and analyzed how excessive collaboration affects student performance. Pierce et al. [129] developed a tool that correlated plagiarism with negative performance over other assignments in the course, and there has also been anecdotal evidence that over-reliance on outside help can incite negative student experiences in a CS1 course [121]. Schneider et al. [149] implemented a plagiarism detection tool that analyzes logs of student interaction with course software. To the best of our knowledge, our work is both the first to use information *during* unsupervised work to identify excessive collaboration and the first to model the probability of false positives in similarity software detection.

4.2 Data

In this study we focus on three offerings of CS1 at Stanford University: Fall 2012 (416 students), Fall 2013 (476 students), and Fall 2014 (528 students). In the course studied, Breakout is the first large, intensive creative project assignment, and enthusiastic students often extend their work beyond the minimum requirements (Appendix B). At the same time, the assignment unfortunately also creates the first opportunity for excessive collaboration. In our course, we know of eight *online* solutions (from online repositories or coding blogs) that students regularly use for excessive collaboration; other students use solutions from peers who are concurrently taking or have previously taken the course.

The Breakout assignment is held on week four of a 10-week course. All three course offerings studied were taught by the same instructor, and students had nine days to work on Breakout (with an extra two late days to submit with potential grade penalty). While students work on the assignment individually until the deadline, they can also attend walk-in office hours held Sunday to Thursday evenings to clarify concepts or discuss debugging tips with undergraduate TAs. In our analysis, we leverage attendance logs of these TA hours—students record their check-in time and their assignment or course issue, and TAs record the students' check-out time and issue resolution.

Similar to Chapters 2 and 3, we use Git and the Eclipse IDE to save process repositories



	μ	SE
Start day	-5.22	2.73
# Snapshots	253	199
Hours on task	9.77	4.93
TA hours	4.18	9.09

Table 4.1: Statistics per Breakout repository (1,420 students).

for the Breakout assignment (Section 2.2). Table 4.1 provides a summary of the 1,420 process repositories used in our dataset. On average, students started the assignment 5.22 (bootstrapped SE=2.73) days before the deadline and spent a mean of 9.77 (bootstrapped SE=4.93) hours on task. TA office hours attended were calculated over the entire course. Hours on task were determined by grouping snapshot times that were within half an hour of each other. Due to the graphics-based, open-ended nature of the Breakout assignment, it is not fully autograded; for this work, we did not have unit tests or metrics for functionality per snapshot.

4.3 Method

In this section, we give an overview of the typical process of flagging student final submissions for excessive collaboration with other students or online solutions. We then present TMOSS, a tractable method for identifying excessive collaboration over intermediate versions of student code.

4.3.1 Traditional software similarity detection

Traditional software similarity detectors compute similarity scores over a student's submitted code; these tools compare the student code to peer submissions and online solutions on various metrics, and flag student submissions that deviate from standard statistics [34, 59, 129, 131, 148, 172, 175]. Among these, Measure of Software Similarity (MOSS) is highly regarded as the standard for detecting software plagiarism in the classroom [8, 148]. The software acts as a filtering tool prior to human decision; submissions with high similarity scores are checked manually by course staff, who identify excessive collaboration as plagiarism on a case-by-case basis. These pairwise detectors all scale quadratically with the

ALGORITHM 1: Computing top matches in TMOSS.

Input: N students (students)
n online solutions (online)
Output: N top matches (top_matches). Each student has a tuple of (highest similarity score, match code).

```
top_matches = [];
for s in 1...N do
    compare_set = getFinalSubmissions(students - students[s]) + online;
    snapshots = getRepository(students[s]);
    M = snapshots.length(); student_matches = [];
    for i in 1...M do
        results = compute_similarity(snapshots[i], compare_set);
        student_matches.append( argmax_{j=1...N+n-1} results[j].getScore() );
    end
    top_matches.append( argmax_{i=1...M} student_matches[i].getScore() );
end
```

size of the class, as comparing N student submissions to N peers and $n < N$ known online solutions requires a runtime of $O(N^2)$.

Running these tools on student final submissions can only identify a subset of the students exhibiting excessive collaboration. For example, suppose a student pastes in an online solution momentarily to check the desired output, then removes the online solution and submits his or her own work. While this is a plagiarism case at many academic institutions, existing software similarity detectors cannot flag this student from his or her final submission work.

4.3.2 TMOSS: Temporal Measure of Software Similarity

TMOSS is a tool designed to extend traditional software similarity detectors by identifying excessive collaboration on any intermediate snapshot of a student's process repository, not just final code submission. While TMOSS (as the name implies) currently implements MOSS-based similarity scores [148], the tool can be easily adapted to different similarity detector backends.

An outline of TMOSS's operation is shown in Algorithm 1. TMOSS produces N top matches—one per student—corresponding to the final peer or online code that returned the highest similarity score over all M snapshots per process repository. The

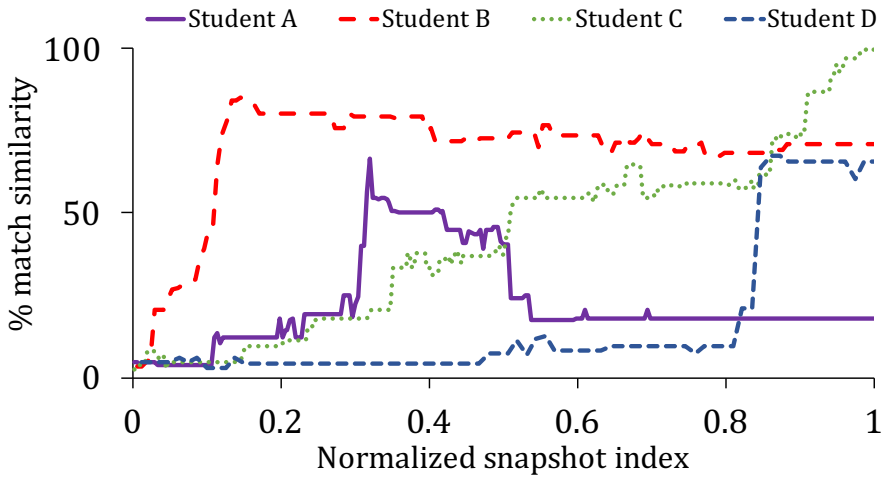


Figure 4.1: Examples of students in the HEC group.

`compute_similarity()` function is our traditional similarity detector backend, which computes similarity scores of a student snapshot to each of the final and online code files. Attempting to run existing pairwise similarity detection algorithms on all M snapshots for each of N students would require $O(N^2M^2)$ runtime, which is impractical. Instead, we avoid pairwise comparison by comparing a snapshot to N final peer submissions and $n < N$ online solutions, thus reducing the runtime per snapshot to $O(N)$ and the overall runtime to $O(N^2M)$. We note that comparing snapshots to final code is often preferable; similarity score algorithms that we surveyed scale much better with larger code files, and comparing two intermediate code files often produces a very low similarity score. Finally, we use human detection as the final step to determine which of these N top matches exhibits excessive collaboration. All students who fall into this category are put into the *hypothesized excessive collaboration* (HEC) group.

Figure 4.1 shows the excessive collaboration patterns over time of four students from our HEC group. Using MOSS as the detector backend for TMOSS, we plot the percent (%) similarity of a matched final submission to snapshot index i, defined as the i-th snapshot in a process repository. Time is computed as the normalized snapshot index (from their first compiled snapshot at 0.0 to their final submission at 1.0). While all of the students graphed exhibit excessive collaboration, Student A would go undetected by a typical run of MOSS on final submissions. We define Student A as a TMOSS-only student.

MOSS similarity scores. While any pairwise software similarity detector that fits the

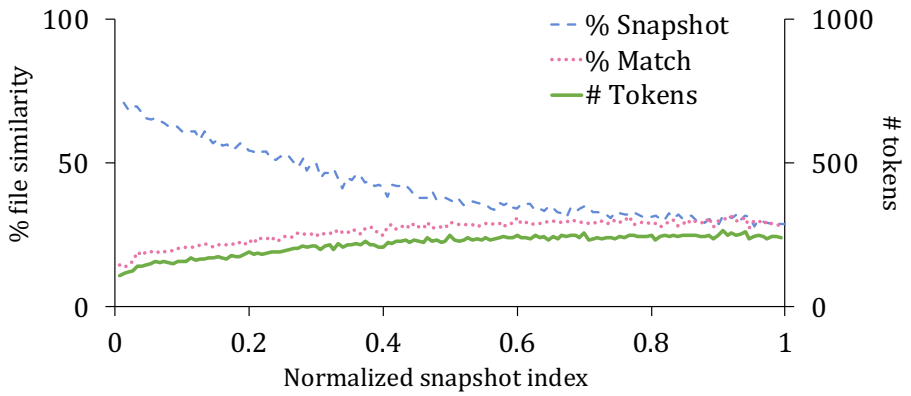


Figure 4.2: 95th percentiles of different MOSS similarity scores over time.

function signature of `compute_similarity()` can be used in TMOSS, we describe MOSS, which is used in our implementation [8, 148]. MOSS returns a triplet of similarity scores for each pair of (snapshot, match), where match refers to a final or online solution file: (1) number of tokens—MOSS's internal, tokenized representation of code—shared between the two programs, (2) % snapshot similarity, and (3) % match similarity. The latter two scores are computed as the percentage of shared tokens in the tokenized MOSS representation of the student snapshot file (% snapshot similarity) and the matched peer or online solution file (% match similarity).

In our implementation of TMOSS, we only consider the first of these metrics—the number of shared tokens—to pick the top match for each student. We ignore the % snapshot similarity score since it varies inversely with the length of the student snapshot, and therefore it will be high when the student starts and will decrease as the student progresses (Figure 4.2). By contrast, the other two metrics do not exhibit this behavior. We further compare the use of number of tokens and % match similarity as similarity scores in Section 4.5.

4.4 Theory

What is the chance that a student, who did not cheat, is reported as having plagiarized? In this section we provide a theoretical framework for calculating the probability of a false-positive maximum similarity score. The theory presented applies to all similarity measures for plagiarism detection, including MOSS and TMOSS.

Consider a student S who worked independently. The score Y that is analyzed for

student S is the highest similarity score between the student and all other submissions: $Y = \max_j X_j$, where X_j is the similarity score between student S and another student j. When we compute X_j, there is an non-zero probability that the score will be accidentally large. While we assume that this probability is exceedingly small, the likelihood of a false-positive report for student S increases when we compute a similarity score between them and *every other* student in the history of the course. The potential for a large max similarity score arising by chance—in the absence of collaboration—is concerning and worth exploring in detail.

We assume that the probability distribution of X_j is unknown but exponentially-tailed.[2] We also assume that, if student S did not collaborate with any of their peers, the values X_j should be mutually independent. Finally, it is reasonable to suppose that the X_j scores have the same (though unknown) distribution. The Fisher-Tippett-Gnedenko Theorem, a more obscure cousin of the Central Limit Theorem, tells us that the **max** of exponentially-tailed independent identically distributed (IID) variables can only converge to a Gumbel distribution [64]. If our assumption holds that Y is the max of exponentially-tailed IID random variables, Y should have a Gumbel distribution, where the following relation holds for all values of k:

$$\Pr(Y \geq k) = 1 - e^{-e^{-(k-\mu)/\beta}} \tag{4.1}$$

The parameters of mode (μ) and scale (β) can be estimated using minimal datapoints via the method of probability weighted moments [102].

With the above assumptions, the MOSS and TMOSS scores of a non-HEC student S (who was hypothesized to *not* have excessively collaborated) should come from a Gumbel distribution. In the case of MOSS, X_j is precisely the pairwise similarity score between the final submissions of students S and j. For TMOSS, $Y = \max_k X_k$, where X_k is the maximum similarity score for the i-th snapshot of student S Therefore if $X_k = \max_j M_{ij}$, where M_{ij} is the pairwise similarity score between the i-th code snapshot of student S and student j's final submission, then $Y = \max_i \max_j M_{ij} = \max_j \max_i M_{ij}$. Then $\max_i M_{ij}$ is the result of `compute_similarity()` of student j's final submission across all snapshot in

[2] An exponential tail is a reasonable assumption for the token count similarity score X_j. Notably, this assumption has a light impact on the results: If X_j has a sub-exponential tail (for example % similarity has a fixed upper limit and thus has no tail), Y will have a Reverse Weibull Distribution, which along with the Gumbel is a special case of the Generalized Extreme Value Distribution.

	MOSS	TMOSS
HEC students	35 (2.5%)	61 (4.3%)
Runtime	0.03 hr	9.77 hr

Table 4.2: Results of TMOSS (per snapshot) and MOSS (per final submission) on set of 1,420 students. HEC students are determined with human verification on the tool's results.

student S's process repository; in the case of MOSS, this is exponentially-tailed. If student S did not collaborate, then $\max_i M_{ij}$ should be mutually independent over different students j, and therefore the distribution of the TMOSS score (with MOSS as a backend) follows equation 4.1.

Once the parameters of Y are known, we can answer questions using Equation 4.1: What is the probability that we mark student S as a false positive, where S, who did not work with any other student, has a similarity score Y that is greater than some threshold k used to report plagiarism? The Fisher-Tippett-Gnedenko Theorem should apply for any plagiarism measure—not just MOSS and TMOSS, and not just for programming assignments.

4.5 Results

We ran both TMOSS and MOSS on our dataset of $N = 1420$ students (along with $n = 8$ known online solutions) and used human verification to determine students in the HEC group. With TMOSS, we found 61 HEC students (4.1% of the dataset); in contrast, we found only 35 students (2.5% of the dataset) using the traditional MOSS tool (Table 4.2). We ran both experiments per course offering by restricting peer submissions to those in the same course; for this particular CS1 course, we did not find significant excessive collaboration across different course offerings. MOSS takes at most a few minutes to run, while TMOSS needs to be run overnight.

We compare the distribution of final submission similarity scores in regular MOSS (Figure 4.3a) with that of the maximum similarity score over all snapshots in TMOSS (Figure 4.3b). In both cases, the distribution of similarity scores for the non-HEC student group (1,359 students, as identified via TMOSS) fits a Gumbel distribution (TMOSS: $\mu = 153.4, \beta = 37.8$; MOSS: $\mu = 118.8, \beta = 39.2$). However, the distribution of TMOSS scores for the HEC group is easily differentiable from the non-HEC group, whereas the MOSS scores for the HEC group are less differentiable, further suggesting that TMOSS is

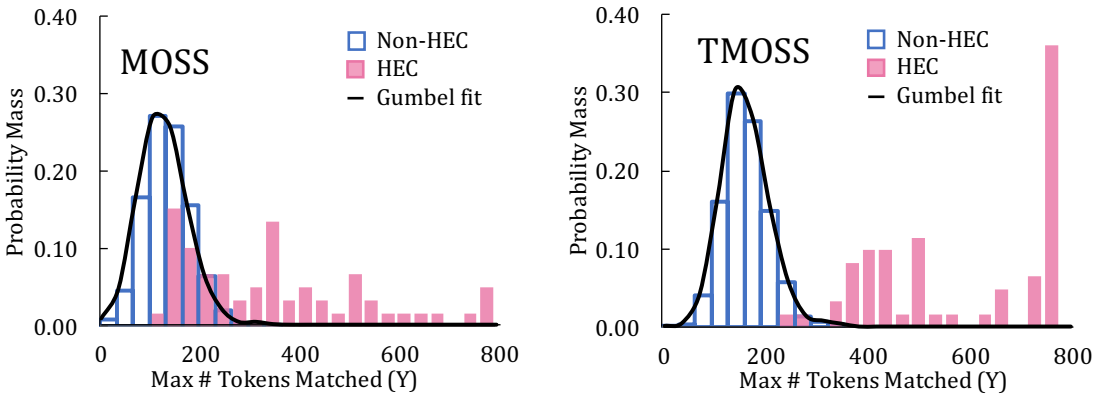


Figure 4.3: Distribution of HEC vs Non-HEC scores by (a) MOSS and (b) TMOSS.

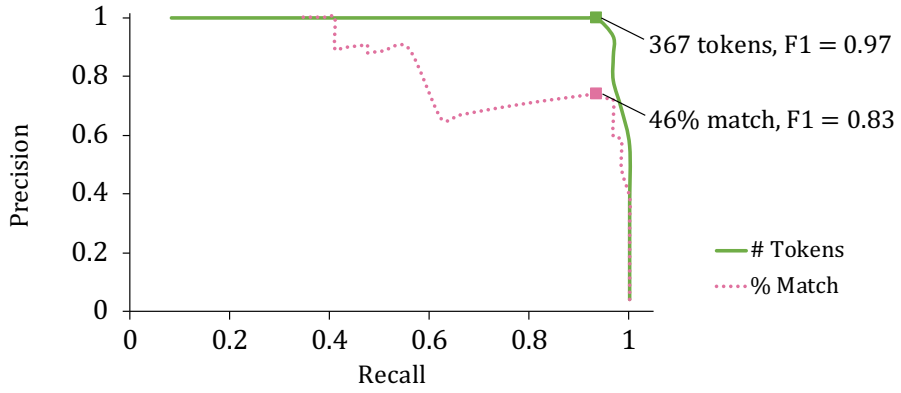


Figure 4.4: Precision-recall curve for TMOSS with two different backend MOSS scores.

a better metric for detecting students in the HEC group.

We next consider whether different MOSS similarity scores would have produced better results. From Section 4.2, the number of shared MOSS tokens and % match similarity can both be considered valid scoring backends as they are less dependent on snapshot length than the third similarity score, % snapshot similarity. We would like to use the score that gives us higher precision and recall (i.e., low false negatives and low false positives) of the HEC group. For each similarity score, we filter the TMOSS top matches through various score thresholds k and compute the precision and recall of the resultant candidate HEC group, shown in Figure 4.4. We find that the optimal threshold for number of tokens ($k = 367$ tokens) produces an F1 score (a harmonic mean of precision and recall) of F1 = 0.97,

# Students	Non-HEC 1359	HEC 61	
	μ (SE)	μ (SE)	p
(a) Midterm	.519 (.282)	.191 (.207)	**<.0001**
(b) Final	.518 (.282)	.167 (.189)	**<.0001**
(c) Start day	-5.31 (2.69)	-3.31 (2.92)	**<.0001**
(d) Hours on task	9.88 (4.92)	7.23 (4.34)	**<.0001**
(e) TA hours	4.27 (9.24)	2.10 (4.04)	.02

Table 4.3: Work patterns of Non-HEC and HEC students.

compared to the optimal threshold for % match similarity ($k = 46\%$) at F1 = 0.83. From these results, we conclude that the number of MOSS tokens leads to lower false negatives and false positives in detecting excessive collaboration.

4.5.1 Performance analysis

We next consider how excessive collaboration is correlated with assignment work patterns and overall course performance. Table 4.3 compares students who excessively collaborated (HEC) and those who did not (non-HEC). Exam scores (Midterm and Final, respectively in Table 4.3a and 4.3b) are calculated as exam ranking in the respective course offering; students who dropped the class partway were removed when calculating exam statistics. Start days (Table 4.3c) are numbers of days *prior* to the scheduled deadline that a student started the assignment. All standard errors are bootstrapped. All p-values are computed by bootstrapping for 100,000 iterations on a one-tailed hypothesis test.

We observe that students in the HEC group compared to those in the non-HEC group tend to start significantly later (an mean of 3.31 days before the deadline, $\delta = 2.0$ days later), have significantly lower midterm and final scores ($\delta = 0.328$ and $\delta = 0.351$, respectively), spend 27% less time on task, and often attend fewer TA office hours (51% less over the entire course).

We considered whether factors other than HEC group membership correlated strongly with student exam performance. A natural one to consider is start date on the Breakout assignment; one could imagine that procrastination predicts course performance, but we found that this signal was not as strong as HEC group membership. Figure 4.5 shows that for non-HEC students, a later start on the assignment is correlated with lower performance on the midterm. However, the HEC group performs consistently lower than the non-HEC

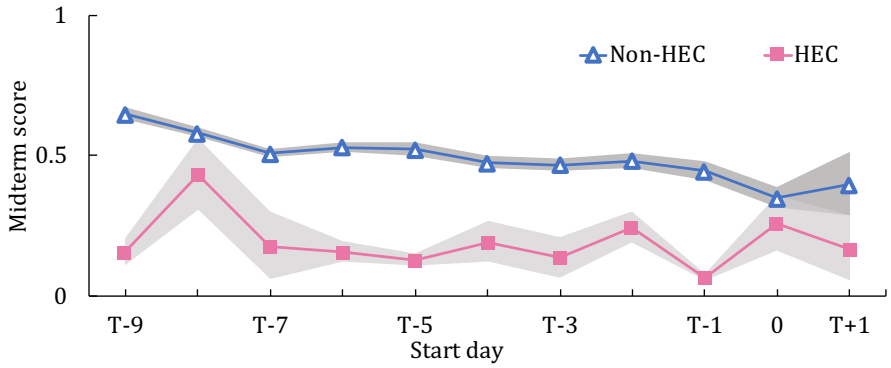


Figure 4.5: Average exam rank of students grouped by Breakout assignment start date, with bootstrapped ±1SE.

# Students	Online-Match 55	Peer-Match 6		MOSS 35	TMOSS-only 26	
	μ (SE)	μ (SE)	p	μ (SE)	μ (SE)	p
(a) Midterm	.192 (.208)	.183 (.202)	.50	.158 (.161)	.235 (.250)	.08
(b) Final	.169 (.193)	.156 (.148)	.49	.114 (.106)	.246 (.247)	**<.01**
(c) Start day	-3.13 (2.95)	-5.00 (1.83)	.07	-2.97 (2.85)	-3.77 (2.94)	.14
(d) Hours on task	6.98 (4.18)	9.54 (5.04)	.09	5.98 (3.65)	8.91 (4.61)	**<.01**
(e) TA hours	2.03 (4.18)	2.67 (2.38)	.25	2.38 (4.61)	1.71 (3.08)	.25

Table 4.4: Work patterns of different groups of HEC students.

group regardless of how early or late they start the assignment. The distribution for the final exam scores were consistent with this finding; for clarity they are not shown.

We next compare students *within* the HEC group and separate students based on their collaboration source (online solution or peer solution) and their detectability on TMOSS and MOSS. In order to detect which students used online solutions, we constructed an undirected connectivity graph of all N students and n online solutions, where edges connect student nodes to their top match nodes. We create this graph because for some students, their top match in TMOSS connected them to a different student, when in reality both students collaborated with an online solution.

In Figure 4.6, we show all components of our graph that contain HEC students; other components that contain only non-HEC students are not shown. Thicker edge weights represent higher TMOSS scores. There are 68 nodes total: 61 HEC students, 3 online solutions, and 4 non-HEC students. We define an *online match* HEC student as belonging

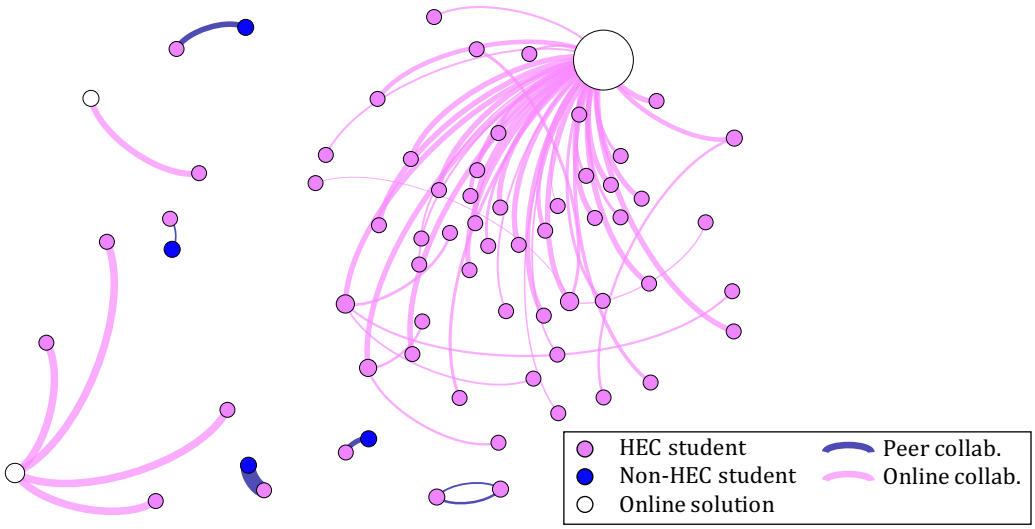


Figure 4.6: Network of HEC students.

to a graph component containing an online solution, a *peer match* HEC student as connected only to other students, and a *conspirator* student as a non-HEC student connected to HEC students. These conspirator students were not flagged for excessive collaboration because their process repository did not significantly match any other student's *final* solution; in other words, HEC students used their non-HEC peer's final solution, but not vice versa.

We observe that out of the 61 students in the HEC group, only 6 students are in our peer-match group. The largest cluster of excessive collaboration contains 50 students connected to an online solution that is the top result when searching the course assignment in a common internet search engine. In Table 4.4, the students in the online-match HEC group tend to start slightly later than those in the peer-match HEC group; however, the p-values are limited by the small sample size and we are unable to draw any real conclusions.

We also compare the 35 students who were detectable via MOSS and the 26 students who were only detectable via TMOSS; these TMOSS-only students had high similarity scores over the course of their assignment, but did not exhibit excessive collaboration in their final submission. Looking again at Table 4.4, we find that the students who were detectable only through TMOSS tended to start the assignment slightly earlier and perform better on the midterm than the MOSS-detectable students, but these statistics were not significant. However, the TMOSS-only students spent significantly more time on task ($\delta = 2.93$ hours) and perform significantly better on the final ($\delta = .132$) than the MOSS-detectable

students.

4.6 Discussion

Our results show that TMOSS, a temporal analysis of student progress to detect excessive collaboration, is a tool with feasible runtime for our classrooms. TMOSS is more precise than the final-submission-based detection algorithms used today. Temporal tools are not only effective for detecting unusual patterns of students, but they can also be used to further understand work patterns as correlated with class performance in an effort to provide improved feedback.

We also found that the Gumbel distribution provides us a probabilistic interpretation for scores; this interpretation can be used in a more formal model for separating HEC from non-HEC for a future tool. It also helps absolve students' concerns of a false positive; the likelihood of getting more than 376 tokens (the similarity score threshold determined in our F1 score thresholding analysis) is $\Pr(Y \geq 376) < 0.003$. We therefore believe that the likelihood of having a false positive in our results is incredibly low.

While the likelihood of false positives with TMOSS is low, our tool does not absolve all false negatives. Figure 4.3 shows that the HEC and non-HEC student populations are not entirely separable; there are some HEC students whose TMOSS scores fall within the Gumbel distribution curve. Furthermore, our connectivity graph in Figure 4.6 shows that there is a small population of non-HEC students that most likely finished their work first, then gave their final submission to their HEC student peer. While these conspirator students do not fall under our definition of excessive collaboration, at many institutions their behavior would still beget instructor intervention. We hope to explore this new student group in future work.

We have shown that the students who have a high match with online solutions comprise a large portion of the HEC group, and students who excessively collaborate with peers within their quarter are very low. Work remains to gauge how the HEC group would change when we expand our analysis to compare against submissions from previous course offerings. Moreover, we have yet to understand how the use of TMOSS changes student behavioral patterns. We hope that the use of similarity scores on intermediate work acts as a deterrent and helps engender an academically honest ecosystem. It would even be possible to modify our Eclipse IDE to upload intermediate work periodically, prior to final

submission; this would allow us to run TMOSS over the course of the assignment and provide timely interventions for students that help them get back on track.

4.7 Summary

We have shown that TMOSS can be used to identify students who exhibit excessive collaboration with online or peer solutions and to understand student work patterns over the course of an assignment. The use of MOSS is not critical; any software similarity detector with numerical similarity scores can be used as a backend to TMOSS. TMOSS can thus also be extended to non-programming scenarios; for example, TurnItIn similarity scores [15] can be used to check for collaboration in essay grading.

We found that students exhibiting excessive collaboration perform worse on exams and make more limited use of TA office hours. In addition, we have found that a Gumbel distribution fits the distribution of similarity scores in the absence of any excessive collaboration.

We must emphasize that TMOSS is not a substitute for human verification; it only helps to provide a more accurate picture of the student landscape. Nor is it intended to impose a negative, pressuring environment on students; instead, we hope that TMOSS is an example of how intermediate student work can be incorporated into the overall feedback system. We hope that improvements to tools like TMOSS will simultaneously deter plagiarism efforts and facilitate the learning process in future classrooms.

TMOSS is just one step in the direction of designing software to better understand students in large classrooms. Such a tool separates typical students from atypical students, and can even be used to indicate at what time a student was experiencing issues. While the use of TMOSS in this work is to detect excessive collaboration, its analysis of intermediate student work can be extended to provide more timely, more accurate feedback. We can therefore identify struggling students in a large classroom, instead of waiting for them to come to us. By reaching out to these students early on, we can give them the right resources to achieve success.

Chapter 5

Reproducing Research Results in Education

When we talk about tools to understand how students learn, we should consider not just how to understand the learning process, but also how to design assignments to meet student learning goals. For advanced students, instructors should consider what projects can teach both engineering rigor and critical thinking, qualities that are necessary for careers in research and industry. As the state of computing evolves worldwide, instructors must give more thought as to how course assignments should reflect the problems that students will encounter in the future.

In many research communities, while many acknowledge the importance of replicating research work, there is also pushback against reporting replicated or reproduced results, because publishing on new phenomena is associated with more impact and possibly more prestige [6, 150]. However, when we remove the pressure of publishing new work, the process of replicating existing research results can reap valuable educational benefits. Since 2012, as part of the graduate networking course at Stanford, over 200 students have participated in a course project to reproduce research results from over 40 networking papers. We have found that through reproducing research, advanced networking students can both hone technical skills with real systems as well as participate and contribute to the networking research community.

5.1 Networking education in universities

Stanford University offers two main networking courses for CS students: an introductory undergraduate class where students learn how the Internet works, including basic principles such as packet-switching, layering, routing, congestion control (CS144: "An Introduction to Computer Networks", `https://cs144.stanford.edu`), and a graduate class where students interested in careers in networking as engineers or researchers read and discuss 20-30 notable research papers (CS244: "Advanced Topics in Networking", `https://cs244.stanford.edu`). Networking classes covering similar topics are prevalent at many universities around the world; importantly, networking courses seem to differ the most in what instructors assign for unsupervised work. For example, in most undergraduate classes it is common for students to write programs that start with the sockets layer and build *upwards* to create applications and libraries on top. At Stanford—and some other universities—students start at the sockets layer and work their way *down*: Our students build transport layers, routers, then Network Address Translation (NAT) devices, and finally download web pages from a public website to their own computer through their NAT in their router, using their transport protocol. In our experience, students who experience "building their own Internet" gain a thorough knowledge of how the Internet works, how to read and implement RFCs, and how to build network systems.

For a graduate class in networking, it is more difficult to determine what to prioritize in programming assignments. Should students build more advanced pieces of the Internet—such as firewalls, load-balancers, and new transport layers? This gives them more experience in building network systems, but it may not challenge their ingenuity and research skills by allowing them to devise and test their own ideas. And so, it is more common in graduate studies for students to undertake a more creative, open-ended project of their own design, perhaps using a simulator, testbed, or analytical tool. In our earlier experiences with CS244, we opted for the latter approach and asked students to create open-ended projects of their own design. But we frequently found the projects lacking—primarily because it is hard to build a meaningful networking system or a persuasive prototype in such a limited time. Often, students were overambitious in their initial project scope and were unable to collect meaningful experimental results on incomplete prototypes. As a result, the projects were rather incremental in nature, and the students' educational experience seemed to be too susceptible to their choice of project. After all, it is hard enough to build a realistic,

interesting, and functioning networking system in a matter of weeks; it is even harder to devise a novel one from scratch and then execute it successfully.

5.2 Why we chose reproducibility

The primary, overarching motivation for asking graduate students to reproduce published research results is the belief that reproducing work brings educational value. Our approach is very similar to how high school and college students study science worldwide: in conjunction with attending lectures and reading textbooks, students reinforce their learning by repeating well-known experiments. Although the students know and anticipate the experimental outcomes prior to entering the classroom lab, it is widely agreed that students are more engaged in their learning when they go through the process of reproducing experiments [78, 114, 130]. Thus, our main goal for adapting this scientific approach to our networking class is for students to attain a detailed, in-depth understanding of a significant paper's key ideas and key results.

The second biggest benefit is the experience our students gain by building—or *recreating*—the experiment for themselves. In the science community, reproducing research generally means repeating the experiment and reproducing results identical to the original. In our class, however, students spend much more time building and recreating the original experiment than collecting and verifying the results. We found that our students found recreating the experiments the most time-intensive and most fulfilling aspect of the project; achieving identical results is something they may (or may not) undertake at the end, after their experiment is working. We therefore distinguish the initial step of *recreating* the experimental infrastructure from the second step of collecting and possibly *reproducing* the same results as the original authors. We rate students highly if they successfully recreate the experiment, regardless of whether they can reproduce the same results. In fact, we find that students learn even more when their experiments yield different results from the original research: they must figure out where the discrepancies lie and discern if there are unstated assumptions or inaccuracies in either their own results or the published results. This is a fascinating, educational experience and often a good lesson in diplomacy.

There are many additional benefits to repeating experiments: if students spend significant time studying and repeating a published experiment, it leads them to ask *meta*-questions about the paper: Why did the researchers pose this problem? Why did they use

or build a particular prototype or simulator, and why did they collect this specific set of results? These questions allow students to understand what the researchers were thinking about when they initially began the research, much more than they would understand by simply reading the paper. By working through the exercise of reproducing results, students gain a deeper understanding of the research process.

The project also gives students the essential experience of building a novel prototype, system, emulator, or simulator, without necessarily having to pioneer an original solution. The students already know the idea in the published work is good: it is practical and has some value—at least enough to warrant publication at a top conference. Furthermore, there is low risk of devising an overambitious project, as the project scope is clear; as a result, we can expect more students to achieve satisfactory results. With a high degree of confidence, they already know interesting results are possible, which encourages them (or perhaps goads them via peer pressure) to complete the work.

This assignment also instills an important principle in our future researchers—that their research results should be reproducible by others, whenever possible. If results can be reproduced, then it is more likely to be adopted by industry, or to be extended upon by other researchers—perhaps by directly reusing the experiment's software. Many fields of research have had various levels of success with promoting and supporting reproducible, published results: in medicine and psychology, these discussions are motivated by observations that published results do not hold up when reproduced [76, 109]; meanwhile, the field of communications and signals has well-established tools for reproducing results [30, 166]. There is a growing movement in systems research to make published results more easily reproduced by others [26, 66]. In the field of computer science, there has been an upward trend of including tools and methods for replication [25, 119, 145]. Our students add to the corpus of reproduced results by providing a simple, packaged reproduction experiment; in this manner, they can encourage the whole networking community to make results more reproducible by others.

All of the above reasons suggest that this type of project is valuable to graduate students, especially those preparing for a career in networking systems research or in industry.

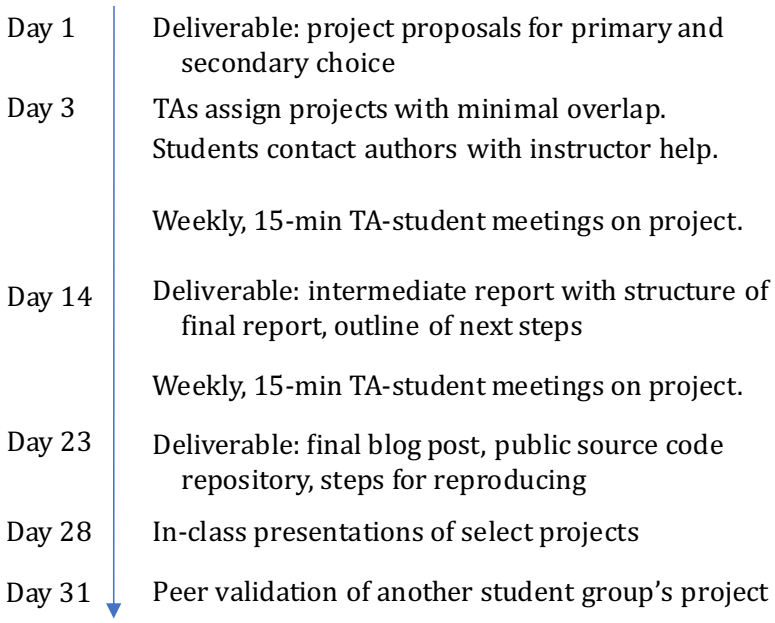


Figure 5.1: Reproducing Research Results project timeline in a 10-week course.

5.3 The Reproducing Research Results project

Our students work in pairs and have three weeks (out of a 10-week course) to complete the assignment. They then have an additional week to verify each other's projects and give in-class presentations. Figure 5.1 shows the project timeline; we describe the main steps of the project below.

Select a project. Each student pair starts by choosing a figure or table from a research paper of interest that is integral to the paper's motivation or claims. Results may include comparing the performance of an algorithm against existing algorithms, demonstrating a metric's usefulness, or recording important traffic and workload data. We provide the students with a list of suggested conferences and research publications that we think make good examples, and we encourage students to choose more recent works, or ones that have not yet been attempted by students in previous course offerings. At Stanford we have had students successfully reproduce results ranging from widely cited papers such as Hedera [9] and DCTCP [11] to traditional papers like RED [55], to unpublished but timely, relevant industry work like SPDY [120].

Choose a method of reproduction. We encourage students to use either the Mininet [90] or Mahimahi [115] emulation systems for their experiment platform, largely because they are most familiar to the instructors. Mininet is best suited for multi-node topologies, whereas Mahimahi is good when modifying and testing congestion control protocols running over a single link. While we generally prefer students to use emulators—as emulators exhibit more realistic network characteristics, such as real-time, live traffic handling for a given node topology [66]—we also encourage the use of simulators, such as ns-3 [68], if the scale or performance is beyond the reach of an emulator. All students are provided with computing resources on Amazon Web Service (AWS) Elastic Compute Cloud (EC2) to run their experiments. By requiring students to start with a standard virtual machine setup on a widely used cloud service platform, any project artifacts become easier for others to use for replicating results.

Contact original authors. After deciding which experiments to run, we help the students contact the authors. Opening up this communication channel between students and researchers has two main benefits: the first is for the students, who now have a primary source to contact regarding the tools, setup, workload and use-cases of the given experiment or research tool. The second is for the researcher, who is now aware that their work is being analyzed critically. Upon completion of the students' project, the researcher will have additional feedback on the benefits, caveats, and persistence of their findings. We discuss anecdotal evidence on the importance of this communication later in Section 5.5.

Work with instructors and peers. Recreating other researchers' work is non-trivial; it is essential that course staff support the students throughout their task. In our course of 40 students we have two teaching assistants, who meet every group every week to check in on the project status and provide guidance if needed. In some instances, we pair up students with graduate student mentors whose expertise overlaps with the target research project. We also require a short intermediate report in the middle of the assignment where students describe what they have done so far, and what they plan to do for the unfinished time. This report gives instructors an opportunity to give feedback and advise on the feasibility of any unfinished steps.

The course ends with students giving short talks about their projects. Students present the main highlights of their reproduced research to the whole class for ten minutes, followed by a short Q&A session.

Write a public blog. Each group is required to document their project—successful

or unsuccessful—and any additional findings or conclusions in a public blog post on the course's *Reproducing Network Research* blog. The blog entry must contain all the code and workload in order for someone else to easily repeat the experiments too. And many do: over the years, our website has been visited by the authors of the original research paper, reviewing our students' work, and by new researchers looking for ideas or ways to get started in their own research (see Section 5.5 for anecdotes).

We verify the results in every blog post using peer validation: every student group is required to replicate the results of another student group. The reproduction effort is required to be an easy, two-step process: (1) download and install any code, and (2) click "run." All code must be available in public code repositories. The students therefore provide all their software source code, experimental data, the means to generate the results, and a detailed interpretation of their results to other researchers. They also upload a public snapshot of their Amazon EC2 machine for easy installation and setup. The public code repositories have proven beneficial for other researchers, who contact the students through the blog in order to use these selected research projects as a base of inspiration or comparison for their own work. These requirements ensure others can build on our students' results, furthering our goal to make more network systems research reproducible.

5.4 Overview of reproduction results

Since 2012, we have seen over a hundred student projects in reproducing networking research. Most have been successful—and a few have not—but overall we have observed that students walk away with the confidence that they can overcome difficult, technical challenges in networking research. In this section, we summarize our experiences in more detail.

Figure 5.2 reports how many student projects successfully recreated research experiments each year, where success is defined as being able to recreate the experiment and generate a result comparable to the original research. The graph shows that a small number of projects each year consistently fall into the "unsuccessful" category, primarily because students were over-ambitious: they attempted reproductions in emulators unsuitable for their project, they could not find the right tools in time, or they overestimated their abilities to build a system from scratch. There are a few other reasons discussed later in Section 5.4.1.

The most popular research papers selected by students are shown in Table 5.1. These

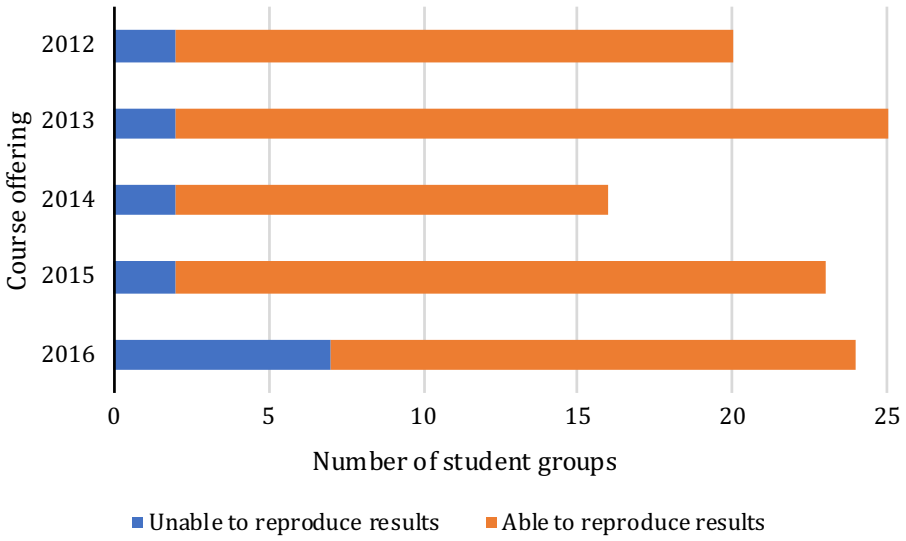


Figure 5.2: The number of successful student projects, listed by course year. Success is defined as being able to recreate the experiment and generate comparable results.

Publication	Reproducibility	
	Able	Unable
TCP Opt-ack Attack [152]	6	2
Jellyfish [154]	8	-
Init CWND [47]	7	-
TCP Fast Open [134]	7	-
Low-rate TCP DoS [88]	4	-
MPTCP [135]	6	-
RCP [46]	4	1
DCTCP [11]	5	-
HTTP-based Video Streaming [73]	5	-
DCell [65]	3	1
Hedera [9]	4	-
Mosh [173]	4	-
PCC [45]	4	-
pFabric [12]	2	1
Sprout [174]	3	-

Table 5.1: The 15 most popular research papers selected for student projects.

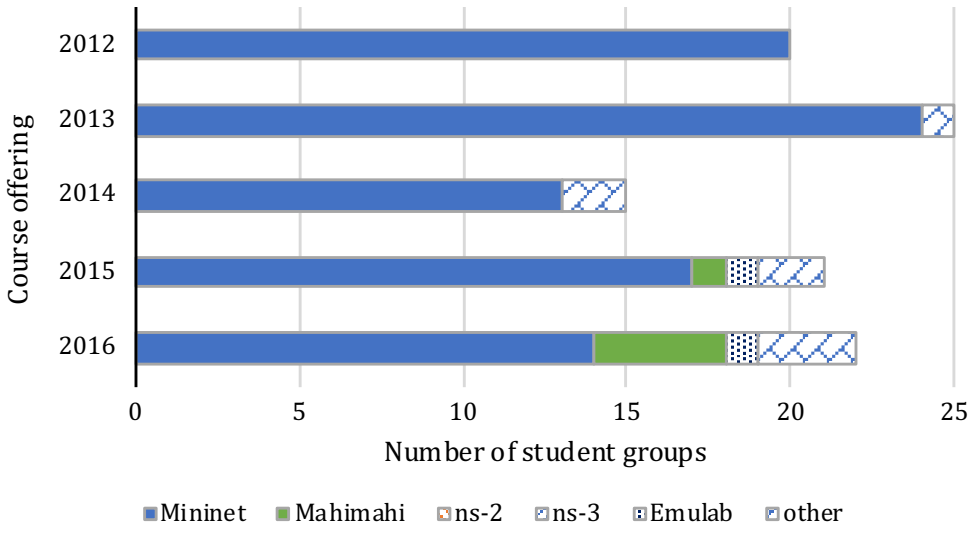


Figure 5.3: Emulator and simulator platforms used by students for reproducing research.

papers were most likely selected because of their ease of setup in the emulators we chose; most of them are variations of TCP, and some of them are application-based or topology-based. Some experiments are more difficult to recreate than others, even if they are from the same research paper; this accounts for some of the unsuccessful projects in Table 5.1. Other students are more ambitious in their projects, e.g., opting to port an existing experiment to a different emulator, which often leads to more difficulties.

Figure 5.3 summarizes the variety of emulators and simulators that students have used. As instructors, while we encouraged the use of Mininet [90] and Mahimahi [115], some groups used ns-2 [77], ns-3 [68], and Emulab [69] instead, often because the original research used these platforms too, making it easier for the students to re-use existing open source code. In some cases, students who started out using simulators ported their experiments to an emulator in order to get real-time, realistic results, all within the three-week time span of the project.

Availability of research code. Running an experiment typically requires two components: the system and the experiment workload. Students can sometimes obtain both from the authors or from online, open-source repositories; other times, they must implement it themselves using a detailed description from a paper or technical report. Overall, we have found that the availability of the original experimental code and workload plays a large part

System source code		Workload generation	
Open-source	12	Open-source	9
Open-source but out-of-date	6	Sufficient details in paper	17
Open-source but inconsistent w/results	1	Student-created	14
Contacted author	2		
Binary available	1		
Student-created	9		
Not needed	9		

Table 5.2: Availability of source code and workload generation code for each paper.

in determining the likely success of reproducing results.

A summary of the availability of code and data for each of the forty unique research papers studied by students in our course is shown in Table 5.2. Occasionally, the research paper lacked key numbers or details about the experiment environment, so students had to reason about additional features and generate their own network workloads. Sometimes, the system source code was open-sourced, but upon further inspection the open-source code produced results inconsistent with those published in the paper, and students had to resort to developing the system from scratch. Despite these setbacks, we have found that students who designed their own experiments gained expert intuition in how their system operated and were thus often very successful in recreating experiments.

If they are running experiments in an emulator, students typically need to scale the experiment (by number of nodes or by datarate) so the emulate can keep up. For example, some research results are gathered in large datacenters with hundreds of nodes and link speed of 10–100 Gb/s. A typical emulator can handle up to tens or hundreds of nodes, with links running at 1–10 Gb/s at most.

5.4.1 Project successes

Our students had varying levels of success with recreating research results. Due to the complexity of the project, it is an accomplishment in itself for students to simply get the system up and running. We therefore have defined success in this project based on three criteria:

1. Are the students able to recreate the experiment?

2. Are the student-generated results and the original results similar in shape?

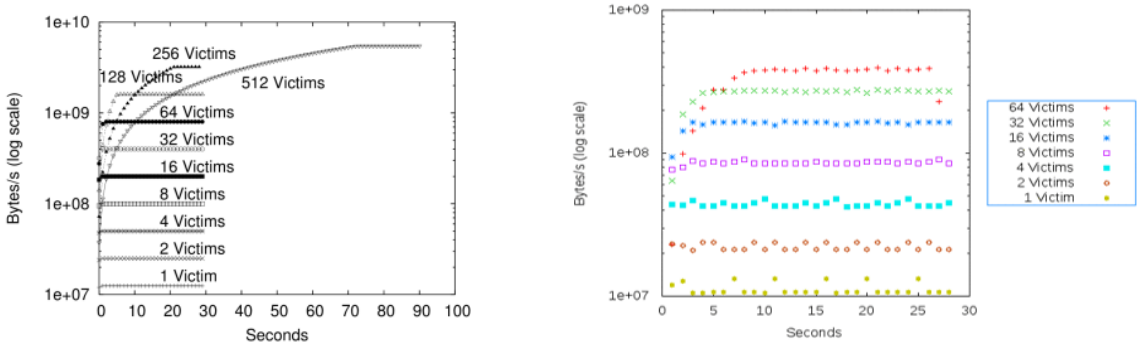
3. Can the students justify any discrepancies in results?

Sometimes, students are able to recreate the original work almost perfectly, subject to scaling or computation resource limits. One student group replicated a TCP opt-ack attack[1], where the task was to create a TCP attacker sending optimistic acknowledgements (opt-acks) to multiple victims over a bottleneck link, generating enough traffic to cause congestion collapse (Figure 5.4a). Even though the original experiment was simulated in ns-2, the students decided to emulate the experiment in Mininet by first designing a Mininet topology and then programming their own opt-ack attacker in Python. They also had to adjust IP table and ARP cache settings on Linux in order to send raw socket traffic on an Amazon EC2 instance. Finally, they were able to produce Figure 5.4b, which shows very similar traffic patterns to the original, simulated experiment. They explained discrepancies in their results; in particular, they were unable to recreate the attack for more than 64 victims due to performance limitations on an emulator for even the largest Amazon EC2 instance with the highest compute power. They also noted that their emulated results had a more jagged shape than the original results, perhaps due to artifacts in measurement. Overall, because the students were able to generate emulated results very similar to the original paper's simulations—and gave sufficient justification for any differences—we consider the student project a success.

Occasionally, students identify discrepancies with the original results for other reasons, despite high confidence in their own recreation of the experiment. For example, one student group compared the performance of ECMP and Hedera on both a hardware testbed and on Mininet.[2] After contacting the original authors, the students reran the benchmark tests and were able to exactly recreate the performance characteristics of Hedera in both the hardware and emulated environments. However, the students consistently found their own hardware ECMP performed significantly better than the original paper's ECMP results. The students reran the ECMP results with spanning tree enabled (something you would not expect in a datacenter) and discovered that the resulting, worse performance was identical to the results in the paper. They subsequently contacted the authors to see if they could verify

[1]Original work by Sherwood et al. [152]. Student blog post at `https://reproducingnetworkresearch.wordpress.com/2016/05/30/cs-244-16-misbehaving-tcp-receivers-can-cause-internet-wide-congestion-collapse/`.

[2]Original work by Alfares et al. [9]. Student blog post at `https://reproducingnetworkresearch.wordpress.com/2012/06/06/hedera/` and further elaborated upon in Handigol et al. [66].



(a) . (b)

Figure 5.4: A successfully recreated experiment for maximum traffic induced by a TCP opt-ack attacker over time for multiple connected victims.[1] (a) Author results (Figure 7 in the original paper); (b) student-recreated results.

their findings, but the original testbed had been torn down years ago, and there was no way to rerun the experiment for additional verification [66]. As one of the students reflected, "when you create a new testbed and create a new environment, there is no way to ascertain the truths or possibilities of the results. On the other hand, if the original authors had used an emulator, such as Mininet, maybe they could've packaged it…so that other people [could use that setup for] experiments."

Reproducing research (un)successfully. There are also cases where students are not able to achieve all three criteria of success. Sometimes, there are limitations in the emulation environment: while setting up an experiment for QJump [63], a student group had to engineer multiple queueing disciplines in Mininet, a feature that did not come out-of-box with the emulator. Another group reported issues configuring POX [105] and Mininet in tandem when trying to recreate the switch controller topology in DCell [65]. Other times, the age of a paper can affect modern reproductions. A group attempted to reproduce the observation that RED maintains significantly higher throughput than Drop Tail Queueing at low queue sizes [55]. However, they found that in most cases, Drop Tail and RED performed equally well. After discussion with a blog commenter, the likely cause is that the underlying TCP mechanism in modern times has evolved considerably, perhaps reducing the relative performance of these two queueing mechanisms.

Students are also occasionally too ambitious: A pair of students tried to implement the rate-based adaptive video streaming of FastMPC [183] in a popular open-source media player. They began the project by successfully finding the same video and wireless traces

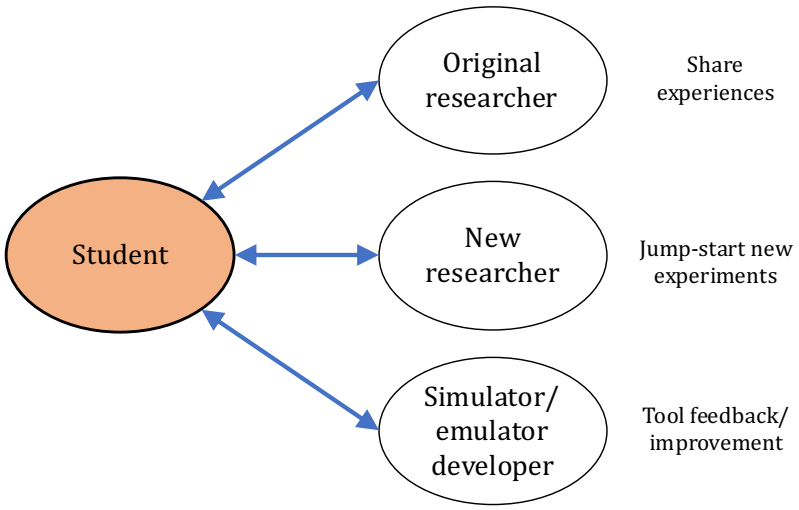


Figure 5.5: Influences of student project on other parts of networking community.

used in the original experiment. However, they ran out of time trying to find an appropriate optimizer that could solve the mixed linear programming model for FastMPC. In retrospect, situations like these could have be avoided with timely interventions by teaching staff, who can help students find appropriate tools, or scale down the scope of their project.

5.4.2 Participating in the community

A bonus outcome of this project is that students who work through reproducing research results play a larger role in the networking research community. While designing and running the experiments, students may interact with the original authors, new researchers who visit our course blog, and even developers of the emulators or simulators. We summarize the interactions in Figure 5.5. We believe the benefits of these interactions go both ways; the networking community at large can also benefit from these student research reproduction projects. Original researchers can share their experiences with students to aid in the reproduction effort, and students can give feedback on how well the system works in different environments. New researchers can use the blog posts and public repositories published by students to jump-start new experiments in new research.

Interacting with platform developers. In particular, simulator and emulator developers can also treat student projects as use cases for evaluating their platform utility. If the platform is still in development, these student projects give developers more opportunities to improve their tool. In our course, we started students with Mininet [90] and

Mahimahi [115], emulators that were initially developed at Stanford and MIT, respectively. When we began this project in 2012, we also received a large volume of feedback from students regarding the usability of Mininet-Hifi, an extension of Mininet designed to give more accurate timing information and detect when it fails to faithfully meet timing. Through this project, students discover the advantages and disadvantages of the platform they are using to recreate results. They can then critique differences in the results and analyze whether the platform setup influenced the replication. When administering a research reproduction project at other academic institutions, we encourage trying out home-grown tools, as student projects are a valuable way of getting feedback on the robustness and accuracy of these new tools.

5.5 Student experiences

With all of the effort involved in recreating research experiments, what's in it for the students? After reviewing course evaluations and blog posts, we invited some students to share how their experiences with the project shaped their perspectives on networking and research. Overall, students said that the project allowed them to undertake and understand new networking topics, gain confidence in their own research abilities, and participate in the networking community. We share some anecdotes highlighting each of these experiences below.

Encountering an unknown facet of networking. Networking is a broad area at the intersection of many fields, and often it is difficult for students with domain expertise in computer science to interact with the lower layers of the networking stack without first understanding the principles of electrical engineering and communications. This project is a good way for students to get a quick, in-depth view of unfamiliar networking areas. A third-year undergraduate and his partner were curious about wireless research; having come from computer science backgrounds, neither of them knew what areas of wireless research were often studied, but they selected a recent paper tackling Wifi handovers with MPTCP [38]. As the student reflected, he and his partner chose their particular project because it was the best way to learn about a handover problem they had both experienced as end users. When asked how they felt about the experience, he said, "I liked it. I specifically liked the level of familiarity I got [with] the paper. There's a level you can only get by reproducing it or implementing it." After communicating with the authors, the students managed to run

an experiment simulation in ns-2 to confirm results illustrating the throughput of different transport protocols during handover between two Wifi access points. After this initial confidence, they then showed results for three Wifi access points in a three-dimensional graph, extending the project beyond the original paper's scope.

Understanding cutting-edge research. Senior students are also interested in learning cutting-edge research, so that they can generate ideas for their own future projects. Reproducing research on a short timeline is a great way to interact with other researchers and understand how to use common tools without needing to expend the rigorous engineering efforts required to achieve research-level system mastery. A pair of second-year graduate students were inspired to reproduce the results from QJump [63] due to their mutual research interests in networked systems. One of the students had attended NSDI 2015 and had heard the authors' presentation in person; at the time, her own research was focused on reducing the latency of networked memory in datacenters, and she felt that QJump was an innovative method for scheduling datacenter traffic. As she recounted, "You could tell from their paper that they really tried to make everything reproducible." The researchers had published methods for recreating experiment workloads for all figures in their NSDI publication.

However, she noted that they ultimately did not use the authors' work directly: "Their assumption was that [people] would reproduce the results in an actual datacenter, whereas we did the emulation in Mininet. In the end, we did not use their scripts directly, but it was nice to see that the authors were enthusiastic to have their work reproduced." This pair of students contacted the authors throughout the project to reconcile scaling and timing differences that arose from using an emulated environment in place of a datacenter and were ultimately successful in recreating the experiments in Mininet. The other student commented that the original authors even "tweeted about [our final blog post], actually." The overall reproduction effort helped the students understand on a deeper level what types of traffic control schemes work in datacenters. The first student mentioned that after the course, she implemented a scheduler for her research similar to one from the project, "which is something that I wouldn't have done if [I had just read] the paper."

Digging deep into workshop papers. Reproducing research also boosts student confidence. One first-year graduate student commented on the project selection process, saying, "you don't want to reproduce something that requires a lot of previous knowledge

or that is hard to reproduce, and you want something that's interesting to you. That process itself takes some learning." Initially, her group selected a very complex and ambitious project that would require significant time to engineer; eventually, they selected a workshop paper on a self-clocked rate adaption for multimedia (SCReAM) [82]. Using workshop papers for a student project are sometimes more challenging than using full-length works, perhaps because the former has a shorter, briefer publication format. However, the students were able to use the authors' public repository code as reference and transformed the authors' simulation into a real-time UDP-based solution on Mahimahi. While recreating the experiment, they realized that there were parameters that functioned well for the original simulation but not for their emulation. After adjusting these parameters, the students were finally able to observe the same results in Mahimahi. One of the students reflected that it was surprising to see that "papers written at this level could also be understood by students who have taken only two courses in networking, and results can be reproduced in part."

With this realization, she and her partner were happy with their published course blog post, which they considered an important facet of their contribution to the overall research work: "[From an educational standpoint,] blog posts are easier to read than papers. If there is one cool idea from a paper that you can reproduce and put into a blog post, I think that could be very valuable. Because in a sense you already did the set up for them, and you wrote it in a [clearer] way." As if confirming her newfound perspective, the authors—whom the students did not contact during their project—incidentally came across our course website and contacted the student and her partner to address critiques and questions that were raised in the students' project blog post. The students also received emails from another graduate student to ask for additional details on running the experiment for his own research.

Boosting experience for future careers. The process of recreating experiments from research can be useful for fostering career skills for both academia and industry. A now-graduated student who recreated experiments for the congestion control protocol DCTCP [11] mentioned that interacting with Mininet was invaluable in her current industry job in network emulators. She and her partner set about recreating an ns-2 simulation of DCTCP's performance in Mininet and came across setbacks in "the kernel version, the amount of memory, the software version …things that [we] didn't really anticipate having trouble with." However, overcoming these struggles were valuable for her current engineering career; she said that "reading up about Mininet and being familiar with how to use it

helped me ramp up [faster in my job] because it happens that my team builds something similar to Mininet." Furthermore, thinking critically about networking papers was a skill that aided her technical conversations with coworkers: the course was "very different from any other course I've taken…[where] you're taught principles, learn how to apply them, and write an exam. This [course, on the other hand] would actually prepare me for the real world."

5.6 Summary

In this chapter, we have highlighted some of our experiences offering a graduate-level networking project in recreating experiments from network research. We have provided a step-by-step guide for an example project in an advanced networking class. After conducting a series of interviews, we verified that the experience is rewarding and interesting for students, and it gives them an opportunity to interact with researchers. In addition, we have learned that documenting the results of these reproduction studies is an essential resource for both future students and the research community as a whole. We hope that the materials presented in this chapter inspire more conversations on offering similar projects in graduate networking curricula, as well as a broader discussion on well-scoped assignments for advanced courses in other fields of computer science.

Chapter 6

Conclusion

In this dissertation, we have presented four contributions to understanding how students learn during unsupervised work. Three of these contributions discuss the learning process—what activities a student participates in while they work on assignments. The first of these, milestones, is a method for quantifying functional progress on a CS1 coding assignment. This quantitative metric enriches the tools that researchers have to understand the learning process, allowing further study on relevant assignments, like open-ended ones used in CS1 classrooms today. The second and third contributions, Pensieve and TMOSS, are tools that instructors can use immediately in their current classrooms to support their students.

These contributions are a first step to understanding how students learn on all assignments in today's classrooms. We have made a breakthrough on simple graphics-based assignments like Pyramid, but tougher tasks like Breakout, which involve human mouse interaction and game design, remain unsolved. We have just scratched the surface on how contemporary techniques in computer vision can help education; as machine learning improves, we can imagine techniques from reinforcement learning, object detection, and natural language processing expanding to improve autograder systems on advanced, complex assignments.

The tools described in this thesis are designed to understand learning process *after* students submit their final work. Such tools benefit both the student, who receives more detailed, in-depth feedback on their problem-solving process, and the instructor, who can aggregate and understand how students interact with an assignment. We have purposefully designed all tools to include human instructor input. A large component of higher education today relies and benefits from human input, and it is only natural for tools designed for

improving today's classroom to enhance a teacher's ability to directly help their students.

As learning environments shift towards an online experience, intelligent tutors [85, 176] will be able to give hints to students *while* they work. Research today has started to explore hint-based feedback [139], conversational tutor systems [61, 146], and adaptive learning with question generation [67, 94]. We can easily envision classrooms where students can receive formative feedback automatically from their learning environments, allowing students to reflect immediately on their problem solving during the assignment. The human instructor can then discuss the learning process in more detail after assignment submission.

The tools on learning process discussed above paint an early picture of how instructors can transform unsupervised work in classrooms, from how to design learning environments that collect data on learning process—here described as process repositories—to how to deliver feedback not just on the final submission, but on the entire problem-solving process. Our fourth contribution in this dissertation explores the design of the assignment itself, which should continue to evolve to support the goals of higher education.

As we move into the future, the unsupervised work component of our students' classroom learning experience will undoubtedly evolve for the better. In order to enable this transformation, students should be motivated to use tools that allow instructors to better understand the learning process. For example, when using Pensieve, students receive intrinsic, positive reward for submitting data, in the form of useful, formative feedback for future assignment work. However, students may view TMOSS as a tool that punishes bad actors—where providing more information improves the tool's ability to detect excessive collaboration. It remains future work to design positive reinforcement systems to support any tool that collects data on learning process—such that students are not only discouraged from bad behavior, but also *encouraged* to follow good practices.

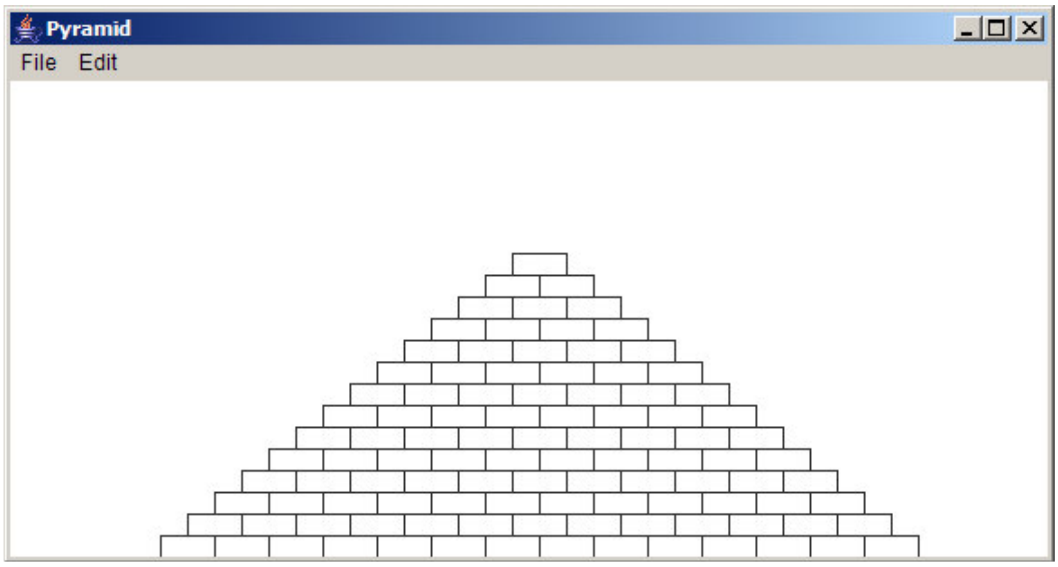
As educators, when we adopt tools to our classrooms we must consider how to motivate students to participate and find data collection the norm. A larger question looms about how the impact of these tools, which will collect data on and give feedback on learning process, will stretch beyond the classroom. While instructors have huge incentives to gather more data about their classrooms, students may refrain from sharing data about their learning process, citing concerns of privacy and security. Both instructors and students together must hold ongoing conversations about the connection between classroom data and student learning. We are confident that the tools discussed in this dissertation—and tools yet undiscovered—will improve our classrooms and help our students learn.

Appendix A

The Pyramid Assignment

This appendix contains the Pyramid assignment from Stanford University's CS1 course (CS106A: Programming Methodologies) from the Winter 2019 school term [124]. The Pyramid assignment is assigned as part of the second homework assignment (consisting of ten short exercises) in this 10-week course.

Write a **`GraphicsProgram`** that draws a pyramid consisting of bricks arranged in horizontal rows, so that the number of bricks in each row decreases by one as you move up the pyramid, as shown in the following sample run:



The pyramid should be **centered** at the bottom of the window and should use constants for the following parameters:

`BRICK_WIDTH`	The width of each brick (30 pixels)
`BRICK_HEIGHT`	The height of each brick (12 pixels)
`BRICKS_IN_BASE`	The number of bricks in the base (14)

The numbers in parentheses show the values for this diagram, but you must be able to change those values in your program.

Figure A.1: The Pyramid assignment handout for Stanford University's CS1 course (CS106A: Programming Methodologies).

```
/*
 * File: Pyramid.java
 * This file is the starter file for the Pyramid problem.
 * It includes definitions of the constants that match the
 * sample run in the assignment, but you should make sure
 * that changing these values causes the generated display
 * to change accordingly.
 */

import acm.graphics.*;
import acm.program.*;
import java.awt.*;

public class Pyramid extends GraphicsProgram {

/** Width of each brick in pixels */
  private static final int BRICK_WIDTH = 30;

/** Height of each brick in pixels */
  private static final int BRICK_HEIGHT = 12;

/** Number of bricks in the base of the pyramid */
  private static final int BRICKS_IN_BASE = 14;

  public void run() {
    /* You fill this in. */
  }
}
```

Figure A.2: Starter code for the Pyramid assignment.

Appendix B

The Breakout Assignment

This appendix contains the Breakout assignment from Stanford University's CS1 course (CS106A: Programming Methodologies) from the Winter 2019 school term [124]. The Breakout assignment is the third homework assignment in this 10-week course.

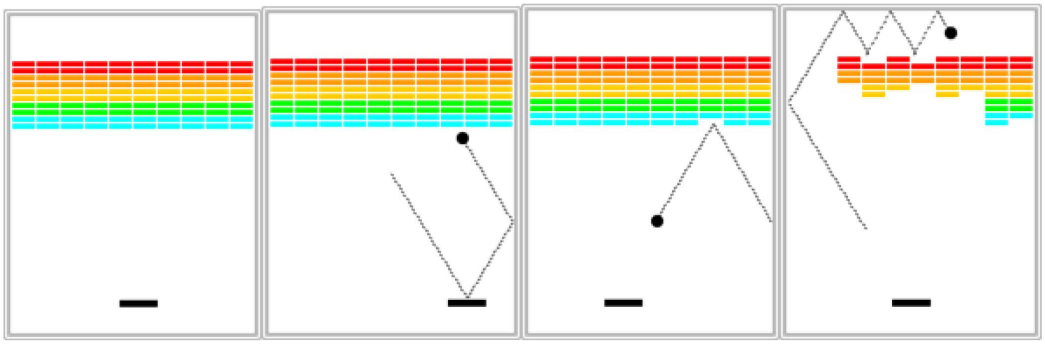
Chris Piech
CS 106A

Assn #3
January 28, 2019

Assignment #3—Breakout!
Due: 12pm on Wednesday, February 6th
This assignment should be done <u>individually</u> (not in pairs)

Based on a handout by Eric Roberts with some modifications by Brahm Capoor

Your job in this assignment is to write the classic arcade game of Breakout, which was invented by Steve Wozniak before he founded Apple with Steve Jobs. It is a large assignment, but entirely manageable as long as you break the problem up into pieces. The decomposition is discussed in this handout, and there are several suggestions for staying on top of things in the "Strategy and tactics" section later in this handout.



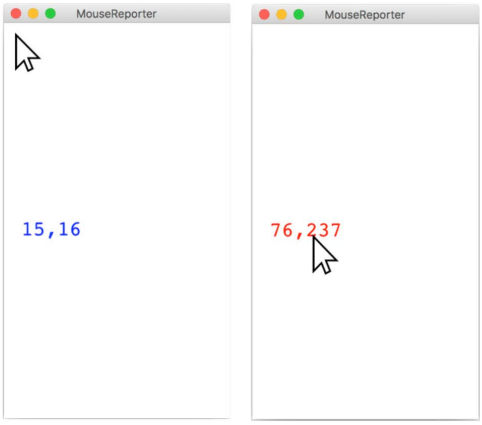
Sandcastle: Prime Checker

A key challenge when writing large programs like Breakout is how best to decompose our solutions into manageable and effectively-implemented methods. In order to practice this skill, you'll begin this assignment by writing a short method that takes in a positive integer greater than 1 as an input and returns a boolean indicating whether or not that integer is prime.

As a reminder, a prime number that is defined such that its only factors are 1 and itself. In `PrimeChecker.java`, we provide a `run` method that tests whether or not a series of numbers are prime. Your job is to implement the `isPrime` method to check whether or not the number is prime.

Sandcastle: Mouse Reporter

To get you warmed up, first write a minimal program that leverages the essential concepts needed for Breakout. Write a `MouseReporter` that creates a GLabel on the left side of the screen. When the mouse is moved the label is updated to display the current x, y location of the mouse. If the mouse is touching the label it should turn **red**, otherwise it should be **blue**.



You should take advantage of the **`setLabel`** method that can be called on a **`GLabel`**. If you look in Chapter 9 (page 299) at the methods that are defined at the **`GraphicsProgram`** level, you will discover that there is a method

```
public GObject getElementAt(double x, double y)
```

that takes a position in the window and returns the graphical object at that location, if any. If there are no graphical objects that cover that position, **`getElementAt`** returns the special constant **`null`**. If there is more than one, **`getElementAt`** always chooses the one closest to the top of the stack, which is the one that appears to be in front on the display. The starter code for **`MouseReporter`** stores the label as an instance variable and adds it to the screen. Make sure to submit your MouseReporter with Breakout.

The Breakout game

In Breakout, the initial configuration of the world appears as shown in the first pane of the image on the previous page. The colored rectangles in the top part of the screen are bricks, and the slightly larger rectangle at the bottom is the paddle. The paddle is in a fixed position in the vertical dimension, but moves back and forth across the screen along with the mouse until it reaches the edge of its space.

A complete game consists of three turns. On each turn, a ball is launched from the center of the window toward the bottom of the screen at a random angle. That ball bounces off the paddle and the walls of the world, in accordance with the physical principle generally expressed as "the angle of incidence equals the angle of reflection" (which turns out to be very easy to implement as discussed later in this handout). Thus, after two bounces—one off the paddle and one off the right wall—the ball might have the trajectory shown in the second diagram. (Note that the dotted line is there to show the ball's path and won't appear on the screen.)

As you can see from the second diagram, the ball is about to collide with one of the bricks on the bottom row. When that happens, the ball bounces just as it does on any other collision, but the brick disappears. The third diagram shows what the game looks like after that collision and after the player has moved the paddle to put it in line with the oncoming ball.

The play on a turn continues in this way until one of two conditions occurs:

1. The ball hits the lower wall, which means that the player must have missed it with the paddle. In this case, the turn ends and the next ball is served if the player has any turns left. If not, the game ends in a loss for the player.
2. The last brick is eliminated. In this case, the player wins, and the game ends immediately.

After all the bricks in a particular column have been cleared, a path will open to the top wall. When this situation occurs, the ball will often bounce back and forth several times between the top wall and

the upper line of bricks without the user ever having to worry about hitting the ball with the paddle. This condition is called "breaking out" and gives meaning to the name of the game.

It is important to note that, even though breaking out is a very exciting part of the player's experience, you don't have to do anything special in your program to make it happen. The game is simply operating by the same rules it always applies: bouncing off walls, clearing bricks, and otherwise obeying the laws of physics.

The starter file

The starter project for this assignment has a little more in it than it has in the past, but none of the important parts of the program. The starting contents of the `Breakout.java` file appear in Figure 1 (on the next page). This file takes care of the following details:

- It includes the imports you will need for writing the game.
- It defines the named constants that control the game parameters, such as the dimensions of the various objects. Your code should use these constants internally so that changing them in your file changes the behavior of your program accordingly.

Success in this assignment will depend on breaking up the problem into manageable pieces and getting each one working before you move on to the next. The next few sections describe a reasonable staged approach to the problem.

– 4 –

Figure 1. The `Breakout.java` starter file has many constants

```
public class Breakout extends GraphicsProgram {

      // Dimensions of the canvas, in pixels
      // These should be used when setting up the initial size of the game,
      // but in later calculations you should use getWidth() and getHeight()
      // rather than these constants for accurate size information.
      public static final double CANVAS_WIDTH = 420;
      public static final double CANVAS_HEIGHT = 600;

      // Number of bricks in each row
      public static final int NBRICK_COLUMNS = 10;

      // Number of rows of bricks
      public static final int NBRICK_ROWS = 10;

      // Separation between neighboring bricks, in pixels
      public static final double BRICK_SEP = 4;

      // Width of each brick, in pixels
      public static final double BRICK_WIDTH = Math.floor(
                  (CANVAS_WIDTH - (NBRICK_COLUMNS + 1.0) * BRICK_SEP) /
                  NBRICK_COLUMNS);

      // Height of each brick, in pixels
      public static final double BRICK_HEIGHT = 8;

      // Offset of the top brick row from the top, in pixels
      public static final double BRICK_Y_OFFSET = 70;

      // Dimensions of the paddle
      public static final double PADDLE_WIDTH = 60;
      public static final double PADDLE_HEIGHT = 10;

      // Offset of the paddle up from the bottom
      public static final double PADDLE_Y_OFFSET = 30;

      // Radius of the ball in pixels
      public static final double BALL_RADIUS = 10;

      // The ball's vertical velocity.
      public static final double VELOCITY_Y = 3.0;

      // The ball's minimum and maximum horizontal velocity; the bounds of the
      // initial random velocity that you should choose (randomly +/-).
      public static final double VELOCITY_X_MIN = 1.0;
      public static final double VELOCITY_X_MAX = 3.0;

      // Animation delay or pause time between ball moves (ms)
      public static final double DELAY = 1000.0 / 60.0;

      // Number of turns
      public static final int NTURNS = 3;

      ...

}
```

Set up the bricks

Before you start playing the game, you have to set up the various pieces. Thus, it probably makes sense to implement the `run` method as two method calls: one that sets up the game and one that plays it. An important part of the setup consists of creating the rows of bricks at the top of the game, which look like this:

The number, dimensions, and spacing of the bricks are specified using named constants in the starter file, as is the distance from the top of the window to the first line of bricks. The only value you need to compute is the *x* coordinate of the first column, which should be chosen so that the bricks are centered in the window, with the leftover space divided equally on the left and right sides. The color of the bricks remain constant for two rows and run in the following rainbow-like sequence: `RED`, `ORANGE`, `YELLOW`, `GREEN`, `CYAN`.

This part of the assignment is almost exactly like the pyramid problem from Assignment #2. The parts that are only a touch more difficult are that you need to fill and color the bricks. This extra complexity is more than compensated by the fact that there are the same number of bricks in each row, and you don't have to change the coordinate calculation from row to row.

Create the paddle

The next step is to create the paddle. At one level, this is considerably easier than the bricks. There is only one paddle, which is a filled `GRect`. You even know its position relative to the bottom of the window.

The challenge in creating the paddle is to make it track the mouse. Mouse tracking makes use of the event-driven model discussed in Chapter 9 of the textbook. Here, however, you only have to pay attention to the *x* coordinate of the mouse because the *y* position of the paddle is fixed. The only additional wrinkle is that you should not let the paddle move off the edge of the window. Thus, you'll have to check to see whether the *x* coordinate of the mouse would make the paddle extend beyond the boundary and change it if necessary to ensure that the entire paddle is visible in the window.

This entire part of the program takes fewer than 10 code lines, so it shouldn't take so long. The hard part lies in reading the Graphics chapter and understanding what you need to do.

Create a ball and get it to bounce off the walls

At one level, creating the ball is easy, given that it's just a filled `GOval`. The interesting part lies in getting it to move and bounce appropriately. You are now past the "setup" phase and into the "play" phase of the game. To start, create a ball and put it in the center of the window. As you do so, keep in mind that the coordinates of the `GOval` do not specify the location of the center of the ball but rather its upper left corner. The mathematics is not any more difficult, but may be a bit less intuitive.

The program needs to keep track of the velocity of the ball, which consists of two separate components, which you will presumably declare as instance variables like this:

```
private double vx, vy;
```

The velocity components represent the change in position that occurs on each time step. Initially, the ball should be heading downward, and you might try a starting velocity of +3.0 for `vy` (remember that

y values in Java increase as you move down the screen). The game would be boring if every ball took the same course, so you should choose the **vx** component randomly.
In line with our discussion of generating random numbers this week, you should simply do the following:

1. Declare an instance variable **rgen**, which will serve as a random-number generator:

   ```
   private RandomGenerator rgen = RandomGenerator.getInstance();
   ```

   Remember that instance variables are declared outside of any method but inside the class.

2. Initialize the **vx** variable as follows:

   ```
   vx = rgen.nextDouble(1.0, 3.0);
   if (rgen.nextBoolean(0.5)) vx = -vx;
   ```

   This code sets **vx** to be a random **double** in the range 1.0 to 3.0 and then makes it negative half the time. This strategy works much better for Breakout than calling

   ```
   nextDouble(-3.0, +3.0)
   ```

   which might generate a ball going more or less straight down. That would make life far too easy for the player.

Once you've done that, your next challenge is to get the ball to bounce around the world, ignoring entirely the paddle and the bricks. To do so, you need to check to see if the coordinates of the ball have gone beyond the boundary, taking into account that the ball has a nonzero size. Thus, to see if the ball has bounced off the right wall, you need to see whether the coordinate of the right edge of the ball has become greater than the width of the window; the other three directions are treated similarly. For now, have the ball bounce off the bottom wall so that you can watch it make its path around the world. You can change that test later so that hitting the bottom wall signifies the end of a turn.

Computing what happens after a bounce is extremely simple. If a ball bounces off the top or bottom wall, all you need to do is reverse the sign of **vy**. Symmetrically, bounces off the side walls simply reverse the sign of **vx**.

Checking for collisions

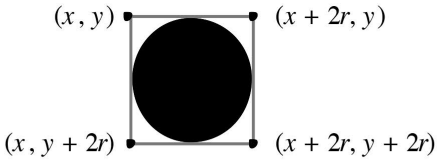
Now comes the interesting part. In order to make Breakout into a real game, you have to be able to tell whether the ball is colliding with another object in the window. As scientists often do, it helps to begin by making a simplifying assumption and then relaxing that assumption later. Suppose the ball were a single point rather than a circle. In that case, how could you tell whether it had collided with another object? What happens if you call

```
getElementAt(x, y)
```

where **x** and **y** are the coordinates of the ball? If the point **(x, y)** is underneath an object, this call returns the graphical object with which the ball has collided. If there are no objects at the point **(x, y)**, you'll get the value **null**.

So far, so good. But, unfortunately, the ball is not a single point. It occupies physical area and therefore may collide with something on the screen even though its center does not. The easiest thing to do—which is in fact typical of the simplifying assumptions made in real computer games—is to check a few carefully chosen points on the outside of the ball and see whether any of those points has collided with anything. As soon as you find something at one of those points, you can declare that the ball has collided with that object.

In your implementation, the easiest thing to do is to check the four corner points on the square in which the ball is inscribed. Remember that a **GOval** is defined in terms of its bounding rectangle, so that if the upper left corner of the ball is at the point (*x*, *y*), the other corners will be at the locations shown in this diagram:



These points have the advantage of being outside the ball—which means that **getElementAt** can't return the ball itself—but nonetheless close enough to make it appear that collisions have occurred. Thus, for each of these four points, you need to:

1. Call **getElementAt** on that location to see whether anything is there.
2. If the value you get back is not **null**, then you need look no farther and can take that value as the **GObject** with which the collision occurred.
3. If **getElementAt** returns **null** for a particular corner, go on and try the next corner.
4. If you get through all four corners without finding a collision, then no collision exists.

It would be very useful to write this section of code as a separate method

```
private GObject getCollidingObject()
```

that returns the object involved in the collision, if any, and **null** otherwise. You could then use it in a declaration like

```
GObject collider = getCollidingObject();
```

which assigns that value to a variable called **collider**.

From here, the only remaining thing you need to do is decide what to do when a collision occurs. There are only two possibilities. First, the object you get back might be the paddle, which you can test by checking

```
if (collider == paddle) . . .
```

If it is the paddle, you need to bounce the ball so that it starts traveling up. If it isn't the paddle, the only other thing it might be is a brick, since those are the only other objects in the world. Once again, you need to cause a bounce in the vertical direction, but you also need to take the brick away. To do so, all you need to do is remove it from the screen by calling the **remove** method.

Finishing up

If you've gotten to here, you've done all the hard parts. There are, however, a few more details you need to take into account:

- You've got to take care of the case when the ball hits the bottom wall. In the prototype you've been building, the ball just bounces off this wall like all the others, but that makes the game pretty hard to lose. You've got to modify your loop structure so that it tests for hitting the bottom wall as one of its terminating conditions.
- You've got to check for the other terminating condition, which is hitting the last brick. How do you know when you've done so? Although there are other ways to do it, one of the easiest is to have your program keep track of the number of bricks remaining. Every time you hit one, subtract one from that counter. When the count reaches zero, you must be done. In terms of the requirements of the assignment, you can simply stop at that point, but it would be nice to give the player a little feedback that at least indicates whether the game was won or lost.

- You've got to experiment with the settings that control the speed of your program. How long should you pause in the loop that updates the ball? Do you need to change the velocity values to get better play action?
- You've got to test your program to see that it works. Play for a while and make sure that as many parts of it as you can check are working. If you think everything is working, here is something to try: Just before the ball is going to pass the paddle level, move the paddle quickly so that the paddle collides with the ball rather than vice-versa. Does everything still work, or does your ball seem to get "glued" to the paddle? If you get this error, try to understand why it occurs and how you might fix it.

Strategy and tactics

Here are some survival hints for this assignment:

- *Start as soon as possible*. This assignment is due in just over a week, which will be here before you know it. If you wait until the day before this assignment is due, you will have a very hard time getting it all together.
- *Implement the program in stages, as described in this handout*. Don't try to get everything working all at once. Implement the various pieces of the project one at a time and make sure that each one is working before you move on to the next phase.
- *Don't try to extend the program until you get the basic functionality working*. The following section describes several ways in which you could extend the implementation. Several of those are lots of fun. Don't start them, however, until the basic assignment is working. If you add extensions too early, you'll find that the debugging process gets really difficult.

Possible extensions

This assignment is perfect for those of you who are looking for + or (dare I say it) ++ scores, because there are so many possible extensions. Remember that if you are going to create a version of your program with extensions, you should submit two versions of the assignment: the basic version that meets all the assignment requirements and the extended version. Here are a few ideas of for possible extensions (of course, we encourage you to use your imagination to come up with other ideas as well):

- *Add sounds*. You might want to play a short bounce sound every time the ball collides with a brick or the paddle. This extension turns out to be very easy. The starter project contains an audio clip file called `bounce.au` that contains that sound. You can load the sound by writing

  ```
  AudioClip bounceClip = MediaTools.loadAudioClip("bounce.au");
  ```

  and later play it by calling

  ```
  bounceClip.play();
  ```

  The Java libraries do make some things easy.
- *Add messages*. The game is more playable if at the start it waits for the user to click the mouse before serving each ball and announces whether the player has won or lost at the end of the game. These are just `GLabel` objects that you can add and remove at the appropriate time.
- *Improve the user control over bounces*. The program gets rather boring if the only thing the player has to do is hit the ball. It is far more interesting, if the player can control the ball by hitting it at different parts of the paddle. The way the old arcade game worked was that the ball would bounce in both the *x* and *y* directions if you hit it on the edge of the paddle from which the ball was coming.
- *Add in the "kicker."* The arcade version of Breakout lured you in by starting off slowly. But, as soon as you thought you were getting the hang of things, the program sped up, making life just a bit more exciting. As one example of this, you might consider adding this feature by doubling the horizontal velocity of the ball the seventh time it hits the paddle, figuring that's the time the player is growing complacent.

- *Keep score*. You could easily keep score, generating points for each brick. In the arcade game, bricks were more valuable higher up in the array, so that you got more points for red bricks than cyan bricks. You could display the score underneath the paddle, since it won't get in the way there.
- *Use your imagination*. What else have you always wanted a game like this to do?

Bibliography

[1] Software Developers : Occupational Outlook Handbook: : U.S. Bureau of Labor Statistics. Online. Retrieved April 2, 2019 from https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm.

[2] CodeHS Project: Breakout. Online, 2016. Retrieved October 25, 2016 from https://codehs.com/library/course/1/module/469.

[3] Generation CS: Computer Science Undergraduate Enrollments Surge Since 2006. Technical report, Computer Research Association, 2017. Retrieved April 2, 2019 from https://cra.org/data/generation-cs/.

[4] The World Bank. Education Statistics. Online, 2017. Retrieved April 25, 2019 from https://datacatalog.worldbank.org/dataset/education-statistics.

[5] Deeplearning.ai. Online, 2018. Retrieved April 7, 2019 from https://www.deeplearning.ai/.

[6] Alireza Ahadi, Arto Hellas, Petri Ihantola, Ari Korhonen, and Andrew Petersen. Replication in Computing Education Research: Researcher Attitudes and Experiences. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, Koli Calling '16, pages 2–11. ACM, 2016.

[7] Alireza Ahadi, Raymond Lister, Heikki Haapala, and Arto Vihavainen. Exploring Machine Learning Methods to Automatically Identify Students in Need of Assistance. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 121–130. ACM, 2015.

[8] Alex Aiken. A System for Detecting Software Plagiarism. 2014. Retrieved October 25, 2016 from https://theory.stanford.edu/~aiken/moss/.

[9] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI '10, pages 19–19. USENIX Association, 2010.

[10] Kirsti M. Ala-Mutka. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education*, 15(2):83–102, 2005.

[11] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM, pages 63–74. ACM, 2010.

[12] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM, pages 435–446. ACM, 2013.

[13] Lorin W. Anderson and David R. Krathwohl. *A Taxonomy for Learning, Teaching, and Assessing. A Revision of Bloom's Taxonomy of Educational Objectives*. Allyn & Bacon, New York, 2nd edition, 2001.

[14] Regina Barzilay, Tommi Jaakkola, Wojciech Matusik, and Pablo Parrilo. 6.036 Introduction to Machine Learning. Online, 2017. Retrieved April 10, 2019 from http://courses.csail.mit.edu/6.036/.

[15] Tshepo Batane. Turning to Turnitin to Fight Plagiarism among University Students. *Educational Technology & Society*, 13(2):1–12, 2010.

[16] Brett A. Becker. A New Metric to Quantify Repeated Compiler Errors for Novice Programmers. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 296–301, New York, NY, USA, 2016. ACM.

[17] Randy Elliot Bennett. Formative assessment: A critical review. *Assessment in Education: Principles, Policy & Practice*, 18(1):5–25, 2011.

[18] Susan Bergin, Ronan Reilly, and Desmond Traynor. Examining the Role of Self-Regulated Learning on Introductory Programming Performance. In *Proceedings of the First International Workshop on Computing Education Research*, ICER '05, pages 81–86. ACM, 2005.

[19] Matthew Berland and Taylor Martin. Clusters and Patterns of Novice Programmers. Annual meeting of the American Educational Research Association, 2011.

[20] Matthew Berland, Taylor Martin, Tom Benton, Carmen Petrick Smith, and Don Davis. Using Learning Analytics to Understand the Learning Pathways of Novice Programmers. *Journal of the Learning Sciences*, 22(4):564–599, 2013.

[21] Paul Black and Dylan Wiliam. Developing the theory of formative assessment. *Educational Assessment, Evaluation and Accountability (formerly: Journal of Personnel Evaluation in Education)*, 21(1):5, 2009.

[22] Paulo Blikstein. Using Learning Analytics to Assess Students' Behavior in Open-ended Programming Tasks. In *Proceedings of the 1st International Conference on Learning Analytics and Knowledge*, LAK '11, pages 110–116. ACM, 2011.

[23] Paulo Blikstein, Marcelo Worsley, Chris Piech, Mehran Sahami, Steven Cooper, and Daphne Koller. Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences*, 23(4):561–599, 2014.

[24] Benjamin S. Bloom. The 2 Sigma Problem: The Search for Methods of Group Instruction as Effective as One-to-One Tutoring. *Educational Researcher*, 13(6):4–16, 1984.

[25] Carl Boettiger. An introduction to Docker for reproducible research, with examples from the R environment. *CoRR*, abs/1410.0846, 2014.

[26] Olivier Bonaventure. The January 2017 Issue. *SIGCOMM Computer Communication Review*, 47(1):1–3, 2017.

[27] David Boud and Elizabeth Molloy. Rethinking models of feedback for learning: The challenge of design. *Assessment & Evaluation in Higher Education*, 38(6):698–712, 2013.

[28] John D. Bransford, Ann L. Brown, and Rodney R. Cocking. *How People Learn*. National Academy Press, 2000.

[29] Tracey Bretag. Challenges in Addressing Plagiarism in Education. *PLoS Medicine*, 10(12), 2013.

[30] Jon B. Buckheit and David L. Donoho. Wavelab and reproducible research. *TIME*, 474:55–81, 1995.

[31] Richard Caldarola and Tanya MacNeil. Dishonesty deterrence and detection: How technology can ensure distance learning test security and validity. In *Proceedings of the 8th European Conference on E-Learning*, pages 108–115, 2009.

[32] David Carless, Diane Salter, Min Yang, and Joy Lam. Developing sustainable feedback practices. *Studies in Higher Education*, 36(4):395–407, 2011.

[33] Adam S. Carter, Christopher D. Hundhausen, and Olusola Adesope. The Normalized Programming State Model: Predicting Student Performance in Computing Courses Based on Programming Behavior. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 141–150. ACM, 2015.

[34] Xin Chen, Brent Francis, Brian Li, McKinnon, and Amit Seker. Shared information and program plagiarism detection. *IEEE Transactions on Information Theory*, 50(7):1545–1551, 2004.

[35] Parmit K. Chilana, Celena Alcock, Shruti Dembla, Anson Ho, Ada Hurst, Brett Armstrong, and Philip J. Guo. Perceptions of non-CS majors in intro programming: The rise of the conversational programmer. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 251–259, 2015.

[36] D. Joseph Clark and Jean Bekey. Use of small groups in instructional evaluation. *POD Quarterly*, 1(2), 1979.

[37] Curtis Clifton, Lisa C. Kaczmarczyk, and Michael Mrozek. Subverting the Fundamentals Sequence: Using Version Control to Enhance Course Management. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, pages 86–90. ACM, 2007.

[38] Andrei Croitoru, Dragos Niculescu, and Costin Raiciu. Towards Wifi Mobility without Fast Handover. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pages 219–234. USENIX Association, 2015.

[39] Quintin Cutts, Emily Cutts, Stephen Draper, Patrick O'Donnell, and Peter Saffrey. Manipulating Mindset to Positively Influence Introductory Programming Performance. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 431–435. ACM, 2010.

[40] Nell B. Dale. Most difficult topics in CS1: Results of an online survey of educators. *ACM SIGCSE Bulletin*, 38(2):49–53, 2006.

[41] Wanda P. Dann, Stephen Cooper, and Randy Pausch. *Learning To Program with Alice*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2nd edition, 2008.

[42] Thanasis Daradoumis, Roxana Bassi, Fatos Xhafa, and Santi Caballé. A Review on Massive E-Learning (MOOC) Design, Delivery and Assessment. *2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 208–213, 2013.

[43] Andrew Davis, Shikha Singh, and Franklyn Turbak. CS111 - Spring 2019. Online, 2019. Retrieved April 7, 2019 from http://cs111.wellesley.edu/.

[44] Ben Dicken. Motion Parallax. Online, 2019. Retrieved April, 30 2019 from http://nifty.stanford.edu/2019/dicken-motion-parallax/.

[45] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pages 395–408. USENIX Association, 2015.

[46] Nandita Dukkipati and Nick McKeown. Why Flow-completion Time is the Right Metric for Congestion Control. *SIGCOMM Compute Communication Review*, 36(1):59–62, 2006.

[47] Nandita Dukkipati, Tiziana Refice, Yuchung Cheng, Jerry Chu, Tom Herbert, Amit Agarwal, Arvind Jain, and Natalia Sutin. An Argument for Increasing TCP's Initial

Congestion Window. *SIGCOMM Computer Communication Review*, 40(3):26–33, 2010.

[48] Carol Dweck. *Mindsets and Math/Science Achievement*. New York: Carnegie Corporation of New York, Institute for Advanced Study, Commission on Mathematics and Science Education, New York, NY, USA, 2008.

[49] Stephen H. Edwards and Manuel A. Perez-Quinones. Web-CAT: Automatically Grading Programming Assignments. *SIGCSE Bull.*, 40(3):328–328, 2008.

[50] Susan E. Embretson and Steven P. Reise. *Item Response Theory*. Psychology Press, 2013.

[51] Steve Engels, Vivek Lakshmanan, and Michelle Craig. Plagiarism Detection Using Feature-based Neural Networks. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '07, pages 34–38. ACM, 2007.

[52] Anneli Eteläpelto. Metacognition and the Expertise of Computer Program Comprehension. *Scandinavian Journal of Educational Research*, 37(3):243–254, 1993.

[53] Mark Everingham, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. The Pascal Visual Object Classes (VOC) Challenge. *Int. J. Comput. Vision*, 88(2):303–338, 2010.

[54] Peter Ferguson. Student perceptions of quality feedback in teacher education. *Assessment & Evaluation in Higher Education*, 36(1):51–62, 2011.

[55] Sally Floyd and Van Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, 1993.

[56] Jeffrey Forbes, David J. Malan, Heather Pon-Barry, Stuart Reges, and Mehran Sahami. Scaling Introductory Courses Using Undergraduate Teaching Assistants. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 657–658. ACM, 2017.

[57] Peter H. Fröhlich. Department of Computer Science at The Johns Hopkins University: 600.111: Python Scripting. Online, 2016. Retrieved October 25, 2016 from http://gaming.jhu.edu/~phf/2011/fall/cs111/assignment-breakout.shtml.

[58] Ursula Fuller, Colin G. Johnson, Tuukka Ahoniemi, Diana Cukierman, Isidoro Hernán-Losada, Jana Jackova, Essi Lahtinen, Tracy L. Lewis, Donna McGee Thompson, Charles Riedesel, and Errol Thompson. Developing a Computer Science-specific Learning Taxonomy. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '07, pages 152–170. ACM, 2007.

[59] David Gitchell and Nicholas Tran. Sim: A Utility for Detecting Similarity in Computer Programs. In *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '99, pages 266–270. ACM, 1999.

[60] Joshua Goodman, Julia Melkers, and Amanda Pallais. Can Online Delivery Increase Access to Education? *Journal of Labor Economics*, 37(1):1–34, 2018.

[61] Arthur C. Graesser, Patrick Chipman, Brian C. Haynes, and Andrew Olney. AutoTutor: An intelligent tutoring system with mixed-initiative dialogue. *IEEE Transactions on Education*, 48(4):612–618, 2005.

[62] David Gries, Lillian Lee, Steve Marschner, and Walker White. CS 1110: Introduction to Computing Using Python Fall 2016. Online, 2016. Retrieved October 25, 2016 from http://www.cs.cornell.edu/courses/cs1110/2016fa/.

[63] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Dont Matter When You Can JUMP Them! In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pages 1–14. USENIX Association, 2015.

[64] Emil J. Gumbel. Les valeurs extrêmes des distributions statistiques. *Ann. Inst. Henri Poincaré*, 5(2):115–158, 1935.

[65] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: A Scalable and Fault-tolerant Network Structure for Data Centers. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM, pages 75–86. ACM, 2008.

[66] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible Network Experiments Using Container-based Emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT, pages 253–264. ACM, 2012.

[67] Michael Heilman. *Automatic Factual Question Generation from Text*. PhD Thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2011.

[68] Thomas R. Henderson, Mathieu Lacage, and George F. Riley. Network simulations with the ns-3 simulator. In *SIGCOMM Demonstration*, 2008.

[69] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale Virtualization in the Emulab Network Testbed. In *USENIX 2008 Annual Technical Conference*, ATC, pages 113–128. USENIX Association, 2008.

[70] Paul Hilfinger. Project 1: The Game of Hog | CS 61A Spring 2017. Online. Retrieved April 30, 2019 from https://inst.eecs.berkeley.edu/~cs61a/sp17/proj/hog/.

[71] David Hovemeyer, Arto Hellas, Andrew Petersen, and Jaime Spacco. Control-flow-only abstract syntax trees for analyzing students' programming progress. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, pages 63–72. ACM, 2016.

[72] Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas. Syntactic and functional variability of a million code submissions in a machine learning MOOC. In *AIED 2013 Workshops Proceedings Volume*, page 25, 2013.

[73] Te-Yuan Huang, Nikhil Handigol, Brandon Heller, Nick McKeown, and Ramesh Johari. Confused, Timid, and Unstable: Picking a Video Streaming Rate is Hard. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC, pages 225–238. ACM, 2012.

[74] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *arXiv:1811.06965 [cs]*, 2018.

[75] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93. ACM, 2010.

[76] John P. A. Ioannidis. Why Most Published Research Findings Are False. *PLOS Medicine*, 2(8):e124, 2005.

[77] Teerawat Issariyakul and Ekram Hossain. *Introduction to Network Simulator NS2*. Springer Publishing Company, Incorporated, 2nd edition, 2011.

[78] T. Jaakkola and S. Nurmi. Fostering elementary school students' understanding of simple electricity by combining simulation and laboratory activities. *Journal of Computer Assisted Learning*, 24(4):271–283, 2008.

[79] Matthew C. Jadud. Methods and Tools for Exploring Novice Compilation Behaviour. In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 73–84. ACM, 2006.

[80] Shanna Smith Jaggars. Choosing Between Online and Face-to-Face Courses: Community College Student Voices. *American Journal of Distance Education*, 28(1):27–38, January 2014.

[81] David S. Janzen and Hossein Saiedian. Test-driven learning: Intrinsic integration of testing into the CS/SE curriculum. In *ACM SIGCSE Bulletin*, volume 38, pages 254–258. ACM, 2006.

[82] Ingemar Johansson. Self-clocked Rate Adaptation for Conversational Video in LTE. In *Proceedings of the 2014 ACM SIGCOMM Workshop on Capacity Sharing Workshop*, CSWS, pages 51–56. ACM, 2014.

[83] Charles Juwah, Debra Macfarlane-Dick, Bob Matthew, David Nicol, David Ross, and Brenda Smith. Enhancing student learning through effective formative feedback. *The Higher Education Academy*, 140, 2004.

[84] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 41–46. ACM, 2016.

[85] Kenneth R. Koedinger, John R. Anderson, William H. Hadley, and Mary A. Mark. Intelligent Tutoring Goes To School in the Big City. 1997.

[86] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[87] Deepak Kumar. Digital Playgrounds for Early Computing Education. *ACM Inroads*, 5(1):20–21, 2014.

[88] Aleksandar Kuzmanovic and Edward W. Knightly. Low-rate TCP-targeted Denial of Service Attacks: The Shrew vs. The Mice and Elephants. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM, pages 75–86. ACM, 2003.

[89] Oren Laadan, Jason Nieh, and Nicolas Viennot. Teaching Operating Systems Using Virtual Appliances and Distributed Version Control. In *Proceedings of the 41st SIGCSE Technical Symposium on Computer Science Education*, pages 480–484. ACM, 2010.

[90] Bob Lantz, Brandon Heller, and Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets, pages 19:1–19:6. ACM, 2010.

[91] Edmond Lau, Xia Liu, Chen Xiao, and Xiao Yu. Enhanced user authentication through keystroke biometrics. *Computer and Network Security*, 6, 2004.

[92] Joseph Lawrance, Seikyung Jung, and Charles Wiseman. Git on the Cloud in the Classroom. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 639–644. ACM, 2013.

[93] Lucas Layman, Laurie Williams, and Kelli Slaten. Note to Self: Make Assignments Meaningful. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '07, pages 459–463. ACM, 2007.

[94] Nguyen-Thinh Le, Tomoko Kojiri, and Niels Pinkwart. Automatic Question Generation for Educational Applications – The State of Art. In Tien van Do, Hoai An Le Thi, and Ngoc Thanh Nguyen, editors, *Advanced Computational Methods for Knowledge Engineering*, Advances in Intelligent Systems and Computing, pages 325–338. Springer International Publishing, 2014.

[95] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[96] Michael J. Lee and Andrew J. Ko. Personifying Programming Tool Feedback Improves Novice Programmers' Learning. In *Proceedings of the Seventh International Workshop on Computing Education Research*, ICER '11, pages 109–116. ACM, 2011.

[97] Sergey Levine and Stuart Russell. CS 188: Introduction to Artificial Intelligence, Spring 2019. Online, 2019. Retrieved April 10, 2019 from http://inst.eecs.berkeley.edu/~cs188/sp19/.

[98] Colleen M. Lewis. The Importance of Students' Attention to Program State: A Case Study of Debugging Behavior. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ICER '12, pages 127–134. ACM, 2012.

[99] Ying Liu, Eleni Stroulia, Kenny Wong, and Daniel German. CVS historical information to understand how students develop software. In *26th International Conference on Software Engineering*, pages 32–36. International Workshop on Mining Software Repositories (MSR), 2004.

[100] Dastyni Loksa, Andrew J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 1449–1461. ACM, 2016.

[101] Andrew Maas, Chris Heather, Chuong Tom Do, Relly Brandman, Daphne Koller, and Andrew Ng. Offering Verified Credentials in Massive Open Online Courses: MOOCs and technology to advance learning and learning research (Ubiquity symposium). *Ubiquity*, 2014(May):2, 2014.

[102] Smail Mahdi and Myrtene Cenac. Estimating Parameters of Gumbel Distribution using the Methods of Moments, probability weighted Moments and maximum likelihood. *Revista de Matemática: Teoría y Aplicaciones*, 12(1-2), 2005.

[103] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Mind Your Language: On Novices' Interactions with Error Messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, pages 3–18. ACM, 2011.

[104] Donald L. McCabe. Cheating among college and university students: A North American perspective. *International Journal for Educational Integrity*, 1(1):10–11, 2005.

[105] Murphy McCauley. Noxrepo/pox: The POX Controller. Online. Retrieved February 22, 2017 from https://github.com/noxrepo/pox.

[106] Nick McKeown and Keith Winstein. CS144: Introduction to Computer Networking. Online, 2019. Retrieved April 10, 2019 from https://web.stanford.edu/class/archive/cs/cs144/cs144.1194/.

[107] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602*, 2013.

[108] Bassam Mokbel, Sebastian Gross, Benjamin Paassen, Niels Pinkwart, and Barbara Hammer. Domain-independent proximity measures in intelligent tutoring systems. In *Educational Data Mining 2013*, 2013.

[109] Ramal Moonesinghe, Muin J. Khoury, and A. Cecile J. W. Janssens. Most Published Research Findings Are False—But a Little Replication Goes a Long Way. *PLOS Medicine*, 4(2):e28, 2007.

[110] Briana B. Morrison, Lauren E. Margulieux, Barbara Ericson, and Mark Guzdial. Subgoals Help Students Solve Parsons Problems. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, pages 42–47. ACM, 2016.

[111] Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 21–29. ACM, 2015.

[112] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 1287–1293. AAAI Press, 2016.

[113] Nachiappan Nagappan, Laurie Williams, Laurie Williams, Miriam Ferzli, Eric Wiebe, Kai Yang, Carol Miller, and Suzanne Balik. Improving the CS1 Experience with Pair Programming. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, pages 359–362. ACM, 2003.

[114] National Research Council. *A Framework for K-12 Science Education: Practices, Crosscutting Concepts, and Core Ideas*. The National Academies Press, Washington, DC, 2012.

[115] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 417–429. USENIX Association, 2015.

[116] Andrew Ng and Kian Katanforoosh. CS230 Deep Learning. Online. Retrieved April 7, 2019 from http://cs230.stanford.edu/.

[117] David J. Nicol and Debra Macfarlan-Dick. Formative assessment and self-regulated learning: A model and seven principles of good feedback practice. *Studies in Higher Education*, 31(2):199–218, 2006.

[118] Nick Parlante, Steven A. Wolfman, Lester I. McCann, Eric Roberts, Chris Nevison, John Motil, Jerry Cain, and Stuart Reges. Nifty Assignments. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '06, pages 562–563. ACM, 2006.

[119] Roger D. Peng. Reproducible Research in Computational Science. *Science*, 334(6060):1226–1227, 2011.

[120] Roberto Peon and Will Chan. Making the web speedier and safer with SPDY, 2012.

[121] Andrew Petersen, Michelle Craig, Jennifer Campbell, and Anya Tafliovich. Revisiting Why Students Drop CS1. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, Koli Calling '16, pages 71–80. ACM, 2016.

[122] Andrew Petersen, Jaime Spacco, and Arto Vihavainen. An Exploration of Error Quotient in Multiple Contexts. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, Koli Calling '15, pages 77–86. ACM, 2015.

[123] Steven T. Piantadosi. Zipf's word frequency law in natural language: A critical review and future directions. *Psychonomic Bulletin & Review*, 21(5):1112–1130, 2014.

[124] Chris Piech. CS106A - Assignment 2: Simple Java. Online. Retrieved June 1, 2019 from http://web.stanford.edu/class/archive/cs/cs106a/cs106a.1194/assn/simpleJava.html.

[125] Chris Piech. CS106A Graphics Key Handout. Online, 2018. Retrieved April 30, 2019 from https://web.stanford.edu/class/archive/cs/cs106a/cs106a.1184/handouts/19%20-%20Graphics%20Contest.pdf.

[126] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. Learning Program Embeddings to Propagate Feedback on Student Code. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1093–1102. PMLR, 2015.

[127] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. Modeling How Students Learn to Program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 153–160. ACM, 2012.

[128] Chris J. Piech. *Uncovering Patterns in Student Work: Machine Learning to Understand Human Learning*. PhD thesis, Stanford University, 2016.

[129] Jonathan Pierce and Craig Zilles. Investigating Student Plagiarism Patterns and Correlations to Grades. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 471–476. ACM, 2017.

[130] Patrice Potvin and Abdelkrim Hasni. Interest, motivation and attitude towards science and technology at K-12 levels: A systematic review of 12 years of educational research. *Studies in Science Education*, 50(1):85–129, January 2014.

[131] Lutz Prechelt, Guido Malpohl, and Michael Phlippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, Universitat Karlsruhe, 2000.

[132] Thomas W. Price, Neil C.C. Brown, Chris Piech, and Kelly Rivers. Sharing and Using Programming Log Data (Abstract Only). In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 729–729. ACM, 2017.

[133] Sarah Quinton and Teresa Smallbone. Feeding forward: Using feedback to promote student reflection and learning—a teaching model. *Innovations in Education and Teaching International*, 47(1):125–135, 2010.

[134] Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain, and Barath Raghavan. TCP Fast Open. In *Proceedings of the Seventh Conference on Emerging Networking EXperiments and Technologies*, CoNEXT, pages 21:1–21:12. ACM, 2011.

[135] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *Presented as Part of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, pages 399–412. USENIX, 2012.

[136] Brian P. Railing and Randal E. Bryant. Implementing Malloc: Students and Systems Programming. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 104–109. ACM, 2018.

[137] Karen L. Reid and Gregory V. Wilson. Learning by Doing: Introducing Version Control as a Way to Manage Student Assignments. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, pages 272–276. ACM, 2005.

[138] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for All. *Commun. ACM*, 52(11):60–67, 2009.

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

[140] Eric Roberts. Strategies for Encouraging Individual Achievement in Introductory Computer Science Courses. In *Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '00, pages 295–299. ACM, 2000.

[141] Eric Roberts. Breakout - Nifty Assignment. Online, 2006. Retrieved April 8, 2019 from http://nifty.stanford.edu/2006/roberts-breakout/.

[142] Eric Roberts, Kim Bruce, James H. Cross, II, Robb Cutler, Scott Grissom, Karl Klee, Susan Rodger, Fran Trees, Ian Utting, and Frank Yellin. The ACM Java Task Force: Final Report. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '06, pages 131–132. ACM, 2006.

[143] Eric Roberts, John Lilly, and Bryan Rollins. Using Undergraduates as Teaching Assistants in Introductory Programming Courses: An Update on the Stanford Experience. In *Proceedings of the 26th SIGCSE Technical Symposium on Computer Science Education*, pages 48–52. ACM, 1995.

[144] Stephanie Rogers, Dan Garcia, John F. Canny, Steven Tang, and Daniel Kang. *ACES: Automatic Evaluation of Coding Style*. Master's thesis, University of California, Berkeley, 2014.

[145] Bernard Rous. The ACM Task Force on Data, Software, and Reproducibility in Publication. Online, 2017. Retrieved May 28, 2019 from https://www.acm.org/publications/task-force-on-data-software-and-reproducibility.

[146] Sherry Ruan, Liwei Jiang, Justin Xu, Bryce Joe-Kun Tham, Zhengneng Qiu, Yeshuang Zhu, Elizabeth L. Murnane, Emma Brunskill, and James A. Landay. QuizBot: A Dialogue-based Adaptive Learning System for Factual Knowledge. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, pages 357:1–357:13. ACM, 2019.

[147] Mehran Sahami and Eric Roberts. Stanford University Computer Science Department: CS 106A - Programming Methodology. Online, 2016. Retrieved October 25, 2016 from http://cs106a.stanford.edu/.

[148] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85. ACM, 2003.

[149] Johannes Schneider, Avi Bernstein, Jan vom Brocke, Kostadin Damevski, and David C. Shepherd. Detecting Plagiarism based on the Creation Process. *CoRR*, abs/1612.09183, 2016.

[150] Jonathan W. Schooler. Turning the Lens of Science on Itself: Verbal Overshadowing, Replication, and Metascience. *Perspectives on Psychological Science*, 9(5):579–584, 2014.

[151] Gregory Schraw, Kent J. Crippen, and Kendall Hartley. Promoting Self-Regulation in Science Education: Metacognition as Part of a Broader Perspective on Learning. *Research in Science Education*, 36(1):111–139, 2006.

[152] Robert Sherwood, Bobby Bhattacharjee, and Ryan Braud. Misbehaving TCP Receivers Can Cause Internet-Wide Congestion Collapse. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05. ACM, 2005.

[153] Arjun Singh, Sergey Karayev, Kevin Gutowski, and Pieter Abbeel. Gradescope: A Fast, Flexible, and Fair System for Scalable Assessment of Handwritten Work. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S '17, pages 81–88. ACM, 2017.

[154] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking Data Centers Randomly. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, pages 17–17. USENIX Association, 2012.

[155] Robert E. Slavin. *Educational Psychology: Theory and Practice*. Pearson Education, 10th edition, 2012.

[156] James G. Spohrer and Elliot Soloway. Analyzing the high frequency bugs in novice programs. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, pages 230–251. Ablex Publishing Corp., 1986.

[157] Ben Stephenson. Coding Demonstration Videos for CS1. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 105–111. ACM, 2019.

[158] Elizabeth Sweedyk, Marianne deLaet, Michael C. Slattery, and James Kuffner. Computer Games and CS Education: Why and How. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '05, pages 256–257. ACM, 2005.

[159] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[160] Kimberly Tanner. Promoting Student Metacognition. *CBE Life Sciences Education*, 11(2):113–120, 2012.

[161] Errol Thompson, Andrew Luxton-Reilly, Jacqueline L. Whalley, Minjie Hu, and Phil Robbins. Bloom's Taxonomy for CS Assessment. In *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78*, ACE '08, pages 155–161. Australian Computer Society, Inc., 2008.

[162] Sherry Turkle and Seymour Papert. Epistemological Pluralism: Styles and Voices within the Computer Culture. *Signs: Journal of Women in Culture and Society*, 16(1):128–157, 1990.

[163] Peter Vamplew and Julian Dermoudy. An Anti-Plagiarism Editor for Software Development Courses. In *Proceedings of the 7th Australasian Conference on Computing Education*, pages 83–90, 2005.

[164] Fabienne M. Van der Kleij, Remco C. W. Feskens, and Theo J. H. M. Eggen. Effects of Feedback in a Computer-Based Learning Environment on Students' Learning Outcomes: A Meta-Analysis. *Review of Educational Research*, 85(4):475–511, 2015.

[165] Laurens van der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[166] Patrick Vandewalle, Jelena Kovăceviç, and Martin Vetterli. Reproducible research. *Computing in Science Engineering*, pages 5–7, 2008.

[167] Scott A. Wallace and Jason Margolis. Exploring the Use of Competetive Programming: Observations from the Classroom. *J. Comput. Sci. Coll.*, 23(2):33–39, 2007.

[168] Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic Neural Program Embedding for Program Repair. *International Conference on Learning Representations*, 2018.

[169] Lisa Wang, Angela Sy, Larry Liu, and Chris Piech. Deep Knowledge Tracing On Programming Exercises. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S '17, pages 201–204. ACM, 2017.

[170] Christopher Watson, Frederick W.B. Li, and Jamie L. Godwin. Predicting Performance in an Introductory Programming Course by Logging and Analyzing Student Programming Behavior. In *2013 IEEE 13th International Conference on Advanced Learning Technologies*, pages 319–323, 2013.

[171] Christopher Watson, Frederick W.B. Li, and Jamie L. Godwin. No Tests Required: Comparing Traditional and Dynamic Predictors of Programming Success. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 469–474. ACM, 2014.

[172] Geoffrey Whale. Plague: Plagiarism Detection Using Program Structure. In *Tech. Rep. 8805*, 1988.

[173] Keith Winstein and Hari Balakrishnan. Mosh: An Interactive Remote Shell for Mobile Clients. In *Presented as Part of the 2012 USENIX Annual Technical Conference*, USENIX ATC '12, pages 177–182. USENIX, 2012.

[174] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *Presented as Part of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, pages 459–471. USENIX, 2013.

[175] Michael J. Wise. YAP3: Improved Detection Of Similarities In Computer Program And Other Texts. In *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, pages 130–134. ACM Press, 1996.

[176] Beverly Park Woolf. *Building Intelligent Interactive Tutors: Student-Centered Strategies for Revolutionizing E-Learning*. Morgan Kaufmann, 2010.

[177] Mike Wu, Milan Mosse, Noah Goodman, and Chris Piech. Zero Shot Learning for Code Education: Rubric Sampling with Deep Learning Inference. *CoRR*, abs/1809.01357, 2018.

[178] Lisa Yan, Annie Hu, and Chris Piech. Pensieve: Feedback on Coding Process for Novices. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19. ACM, 2019.

[179] Lisa Yan and Nick McKeown. Learning Networking by Reproducing Research Results. *SIGCOMM Comput. Commun. Rev.*, 47(2):19–26, 2017.

[180] Lisa Yan, Nick McKeown, and Chris Piech. The PyramidSnapshot Challenge: Understanding student process from visual output of programs. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19. ACM, 2019.

[181] Lisa Yan, Nick McKeown, Mehran Sahami, and Chris Piech. TMOSS: Using Intermediate Assignment Work to Understand Excessive Collaboration in Large Classes. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 110–115. ACM, 2018.

[182] Ayça Yildirim. COMP130/131. Online, 2019. Retrieved April 7, 2019 from https://sites.google.com/a/ku.edu.tr/comp130/spring-2019.

[183] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 325–338. ACM, 2015.

[184] Jeffrey R. Young. Dozens of plagiarism incidents are reported in Coursera's free online courses. *The Chronicle of Higher Education*, 2012.

[185] Daniel Zingaro, Michelle Craig, Leo Porter, Brett A. Becker, Yingjun Cao, Phill Conrad, Diana Cukierman, Arto Hellas, Dastyni Loksa, and Neena Thota. Achievement Goals in CS1: Replication and Extension. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 687–692. ACM, 2018.

[186] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2017.