Title

Detecting Students of Concern in Introductory Programming Classes: Techniques to Indicate Potential Struggle or Cheating

Author
Alzahrani, Nabeel

Publication Date
2022

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Detecting Students of Concern in Introductory Programming Classes: Techniques to Indicate Potential Struggle or Cheating

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Nabeel Alzahrani

June 2022

Dissertation Committee:
Dr. Frank Vahid, Chairperson
Dr. Tony Givargis
Dr. Mariam Salloum
Dr. Paea LePendu

The Dissertation of Nabeel Alzahrani is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to sincerely thank my Ph.D. advisor Dr. Frank Vahid for his guidance during my Ph.D. journey. Also, I would like to thank Dr. Tony Givargis, Dr. Mariam Salloum, and Dr. Paea LePendu for serving on my dissertation committee.

Some of the work in this dissertation (Chapters 2, 3, 4, and 5) has been published in the American Society for Engineering Education (ASEE) Annual Conferences (2018, 2019, 2021, and 2022)

ABSTRACT OF THE DISSERTATION

Detecting Students of Concern in Introductory Programming Classes: Techniques to Indicate Potential Struggle or Cheating

by

Nabeel Alzahrani

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2022
Dr. Frank Vahid, Chairperson

This work is motivated by the observation that interest in computer science continues to grow, but failure rates in introductory programming courses ("CS1") have been concerning. Student frustration is a common source of dissatisfaction, attrition, and cheating in programming courses. A common cause of frustration is getting “stuck”, what we call struggling, on programming tasks, where students spend excessive time or make excessive attempts on a problem with little progress. We argue that early detection of struggle and cheating can help students, where instructors can provide early proper intervention. This dissertation addresses two problems: (1) Detecting and preventing struggle, and (2) Detecting and preventing cheating. The focus is mostly in CS1 but the work may apply to various programming courses. First, this work addresses the problem of how to identify struggle. We develop techniques and tools to measure and detect student struggle. A key novelty here is to identify programming errors that cause struggle rather than just errors that slightly annoy students, or common errors that may not cause struggle. This is important because some struggle is a normal part of learning to program.

Second, this work addresses the problem of how to identify cheating. Many CS instructors use code similarity detection tools (such as Moss) to detect copied solutions shared among students in class. However, the code similarity detection approach is not effective to detect cheating when students submit unique solutions. Due to the accessibility and affordability of online “tutoring” services, a student can get a unique solution in a few minutes for about $1 (with a monthly subscription), even while the student is in a lab session. A key novelty here is to detect cheating that may not be detected by the code similarity approach. We propose techniques and tools to detect cheating. We show that our techniques and tools are powerful and able to identify and detect struggle and cheating in CS1 classes with good accuracy in a short time. To maximize the impact of our work, we plan to make our tools available to the CS community as free web tools. The techniques and tools can help improve education in CS1 and other programming courses.

Table of Content

List of Figures

List of Tables

1. Introduction

Interest in computer science continues to grow, with college computer science majors tripling between 2006 and 2015 [1, 2]. However, failure rates in introductory programming courses ("CS1") have been at a rather high 25-30% for several decades [3]. Frustration is a common source of dissatisfaction or attrition in CS1 courses [4, 5]. A common cause of frustration is getting “stuck”, what we call struggling, on programming tasks, where students spend excessive time or make excessive attempts on a problem with little progress. Although some struggle is a normal part of learning to program, our focus is to reduce *excessive* struggle that goes beyond normal learning and instead may cause frustration and ultimately cheating or attrition.

Detecting struggle was harder in the past due to a lack of online logging of student activity, but is more possible today with newer tools being used in CS1 classes, such as auto-graders, integrated development environments (IDEs), software version control tools, etc.

Previous work examined common errors encountered by students in CS1 courses [6, 7, 8, 9, 10, 11]. Spohrer [6] analyzed programming errors using a cognitive science model. Spohrer used a Goal And Plan tree to trace the root causes of errors, which defined plans (steps/procedures) as the techniques to solve the problem, and the goals as the desired result to achieve or accomplish. Spohrer found that once there is a mismatch between a plan and a goal, an error occurs. Yarmish [7] used a similar approach but added more components to a plan. Other work suggests that errors occur due to inaccurate mental models of program state [6, 7, 8, 12, 13]. Horstmann [14] presented a list of common errors when introducing C++ programming concepts and constructs. Horstmann presented common errors in each chapter, which may help students avoid such errors. Oualline [15] devised debug examples that ask the reader to find and correct those errors. Ginat [16] suggested learning from student errors by building debugging examples based on student misconceptions.

In Chapter 2 we discuss our investigation into errors that cause struggle. In Section 2.1, we defined a struggle metric using a combination of excessive time spent and excessive attempts, relative to other students in a course and reasonable thresholds. We analyzed struggle on 78 short, auto-graded coding homework problems for an 80-student Spring 2017 introductory C++ programming course at a research university. We found the struggle rate to be 10-15%. Our main focus was to determine the errors that led to such struggle, and thus we manually examined the student submissions for the 10 homework problems having the highest struggle rates. We described the errors and potential underlying student misconceptions that seemed to lead to that struggle. We found that most common errors belong to the following: nested loops, else-if vs. multiple if, random range, input/output, for loop and vector, for loop and if, vector index, negated loop expression, and boolean expressions. Having a deeper understanding of these common errors may aid teachers and authors to help students avoid or correct such errors, thus reducing struggle, which may reduce frustration and potential cheating and attrition.

In Section 2.2, we conduct an extensive manual analysis for errors that cause struggle, such as comparing struggling students with non-struggling students, finding average time-to-fix, and finding the most time-consuming errors to fix. For 89 online auto-graded C++ coding homework problems in our CS1 class of 100 students (mostly engineering/science majors), we first automatically determined the 12 problems with the highest struggle rates. Then, we spent about 100 hours manually examining incorrect student submissions to determine what errors caused struggle and the time spent on each error. Like previous work, we found many common general errors, like using = rather than ==. However, we also found problem-specific errors, like misusing a particular library function, leading to a first conclusion that a help system should allow teachers/authors to add problem-specific hints. Furthermore, we analyzed errors that caused the longest struggle, and found some uncommon "one-off" errors, leading to a second conclusion that a help system will not be able to detect all errors and thus might need automated recommending or alerting for human assistance (or other techniques).

In Chapter 3, we summarize a survey we conducted of the literature to learn about errors that cause struggle. We focused on common logic errors made by students learning to program in CS1 classes, as reported in 47 publications in 3 decades (from 1985 to 2018). A logic error causes incorrect program execution, in contrast to a syntax error, which prevents execution. Logic errors tend to be harder to detect and fix and are more likely to cause students to struggle. The publications described 166 common logic errors, which we classified into 11 error categories: Input (2 errors), output (1 error), variable (7 errors), computation (21 errors), condition (18 errors), branch (14 errors), loop (27 errors), array (5 errors), function (24 errors), conceptual (43 errors), and miscellaneous (4 errors). Among those errors, we highlighted 43 that seemed to be the most common and/or troublesome. This survey can help instructors, authors, and tool developers focus on helping learners detect or avoid these common errors in CS1.

In Chapter 4, we discuss our observation that the extensive manual analysis for errors that cause struggle by examining every student's submissions is daunting and time-consuming. Moreover, many CS instructors desire to see more than a student's final submission on a programming assignment, wanting visibility into how the student developed their code. Recently, program auto-graders have grown in usage. A popular cloud-based auto-grader, used in over 2,000 university courses per year, provides a log with an entry whenever a student does a develop run or submission run of their code. Beyond time/date and score, each entry links to a source code snapshot. Using that log file as input, we introduce "code progression highlighting" as a mechanism for instructors to gain visibility into students' code development. The highlighter starts with a statistical summary for each student in roster form, sortable by any statistic such as time spent, number of attempts, code size, a struggle metric, and more. For any student, an instructor can expand to view all code entries from that student, highlighted to show changes from the previous entry (the "progression"), with statistics per entry like time spent, characters changed, current score, etc. The progression highlighter helps instructors to assist students during office hours, detect some cheating not detectable by similarity checkers, or see where students are struggling. We intend to make the progression highlighter free on the web for instructors to use, simply by uploading the log file from the popular auto-grader, or from any system whose log file is converted to that format, and thus can serve the community of CS instructors to gain insights on their students' code development processes.

While we were using the progression highlighter to study student progressions to analyze errors that cause struggle, we noticed a lot of cheating cases in programming assignments (labs). Our research since then has shifted from detecting and preventing struggle to detecting and preventing cheating.

In Chapter 5, we discuss that cheating in programming courses is a well-known problem [17, 18, 19, 20]. We define cheating in a programming course as submitting someone else's code. Cheating is often via plagiarism (copying existing code) or contracting (hiring someone to write code). In some cases, copied code will be similar to classmates' code or known-online sources, so such copying can be detected by traditional code similarity approach detection tools such as Moss [21]. But increasingly, solutions produced by cheating might not be similar, due to contracting, due to a plethora of solutions available online such that a student may find a unique one, or due to students learning (via web tutorials) how to modify copied code to beat code similarity tools [22, 23]. And contracting is becoming affordable and accessible for students to submit unique solutions. For example, a student can take a phone picture of a lab problem and send it to a “tutoring service” to get a unique solution in a few minutes for about $1 (with a monthly subscription), even while the student is in a lab session [24]. Using a cloud-based programming environment that records every code compile/run, we detected code clearly not written by students in our class and noticed running of such code was sometimes preceded by a "drastic change" in their code history -- a run whose code is so dramatically different from the previous run as to be unlikely to have been derived normally from the previous run. Some students submitted such code right away, what we call an "initial leap". Other students tried writing code themselves, gave up, and then copied from an online source or hired a contractor, what we call "gave up". Among either group, we further noticed that some would sequentially try a variety of copied solutions attempting to find one that works, what we call "solution hopping," causing even more instances of drastic changes in the student's code history. Thus, we developed a tool, based on a simple "text diff" algorithm, to detect drastic code changes in student code progressions, and to point instructors to possible cheating cases. We conducted two experiments. The first experiment measured the accuracy of our tool. The tool averaged 100% sensitivity and 100% specificity using real data and synthetic data. The second experiment studied the prevalence of drastic changes in real student programs in our course. The study showed about 32% of students in the initial leaps group, and 5% in the gave up group, which we manually confirmed as actual cheating. Furthermore, 24% of initial leap students and 47% of gave up students subsequently solution hopped. We plan to make our drastic change detection tool available to the CS community as a free web tool.

In Chapter 6, we discuss our observation that students cheating in a programming course often get solutions from different sources for different labs, causing one student's coding style to vary unusually across labs for that same student. We introduce a complementary approach to detect possible cheating. Our approach checks for code style variation for the same student across different labs. We developed a tool to detect code style variation and to point instructors to possible cheating cases. We conducted experiments to measure the accuracy of our tool in three courses that use different programming languages (C++, Java, and Python). The tool's accuracy was near 100% (in both sensitivity and specificity). We plan to make the code style variation detection tool available for the CS community as a free web tool.

In Chapter 7, we describe our observation that copied code often has a code style that deviates from a class' code style, which we call class code style anomalies. We developed an approach to detect possible cheating based on class code style anomalies. We implemented our approach in a tool and used the tool in some of our classes to detect possible cheating. We conducted experiments to evaluate the accuracy of our tool in detecting class code style anomalies. The tool neared 100% accuracy in sensitivity and specificity metrics. We plan to make our tool available to the CS community as a free web tool.

2. An Analysis of Common Errors Leading to Excessive Student Struggle on Homework Problems in an Introductory Programming Course

2.1 Introduction

In Section 2.2, we describe our study of errors leading to struggle on auto-graded homework problems in an introductory C++ course. We introduce the type of homework problem used in the study. We define a quantitative metric for "struggle" and show the struggle rates across all 78 homework problems in our course of 80 students. We highlight the homework problems having the highest struggle rates and summarize our manual investigation of the programming errors that seemed to lead to such struggle.

Knowing those errors may help teachers and publishers develop techniques or content that help students avoid such errors. As an example, we showed that adding a hint to certain homework problems reduced struggle rates by 17%; more aggressive prevention and intervention should see even more reductions.

We believe some struggle is a normal part of learning to program. Our goal is to reduce excessive struggle that goes beyond normal learning and instead may cause frustration and ultimately attrition.

In Section 2.3, we analyze common errors in CS1, in this case for 89 auto-graded C++ coding homework problems provided in widely-used zyBooks. Our class used the C++ zyBook, but similar homework problems exist in the C, Java, and Python zyBooks. While we found many general common errors similar to previous works, we also found many common problem-specific errors. This means if teaching or help systems only focus on general common errors, then many students will still struggle. In fact, via further analysis, we found that "one-off" errors also cause struggle -- errors that are not common but that a particular student got stuck on. This means we need to go even further to detect such situations and provide customized help.

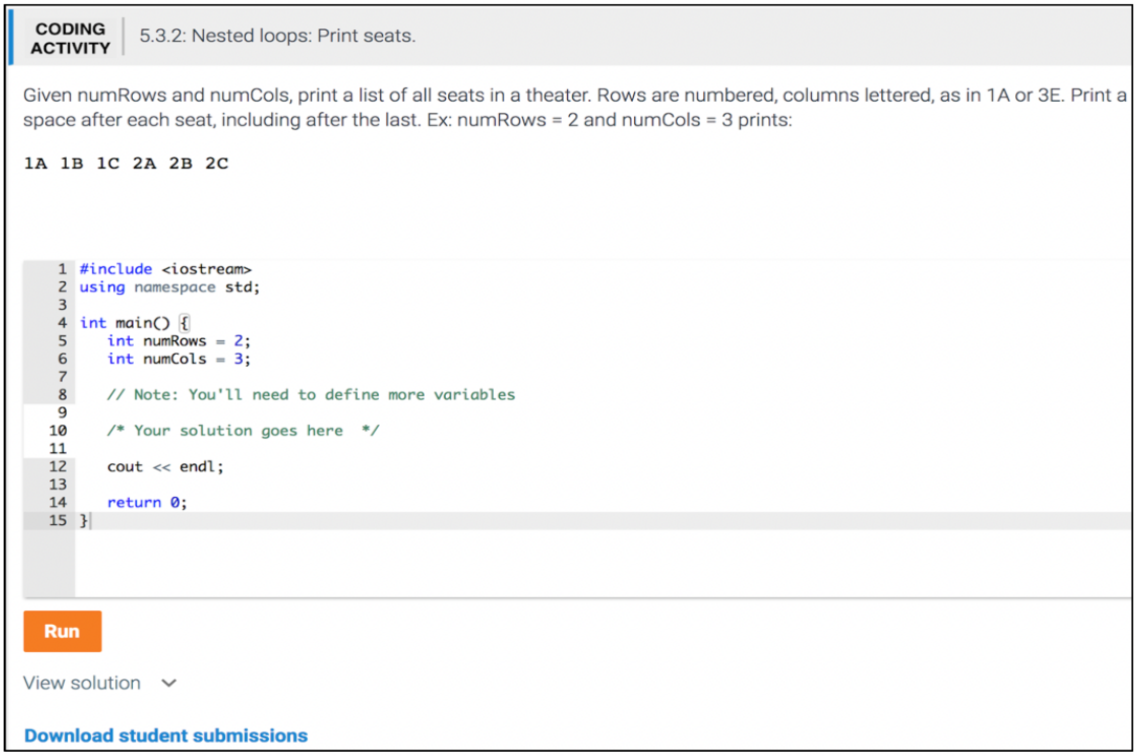
2.2 Studying Struggle

2.2.1 Methodology

2.2.1.1 Homework Problems

We used online content that had several coding homework problems per content section [1]. Each homework problem is called a coding activity (CA). Each CA has the student complete some existing code to achieve a goal, typically only requiring about 1-10 lines of student code.

The student cannot modify the template code except in the region indicated. The student can submit their solution, and the system automatically compiles the program and runs the program with various test values, comparing with the expected output. The student can re-attempt each activity as many times as desired. Figure 2.1 shows an example CA, on the hard end of the spectrum, requiring the student to create nested loops; many students struggled with this activity.



5.3.2: Nested loops: Print seats.

Given numRows and numCols, print a list of all seats in a theater. Rows are numbered, columns lettered, as in 1A or 3E. Print a space after each seat, including after the last. Ex: numRows = 2 and numCols = 3 prints:

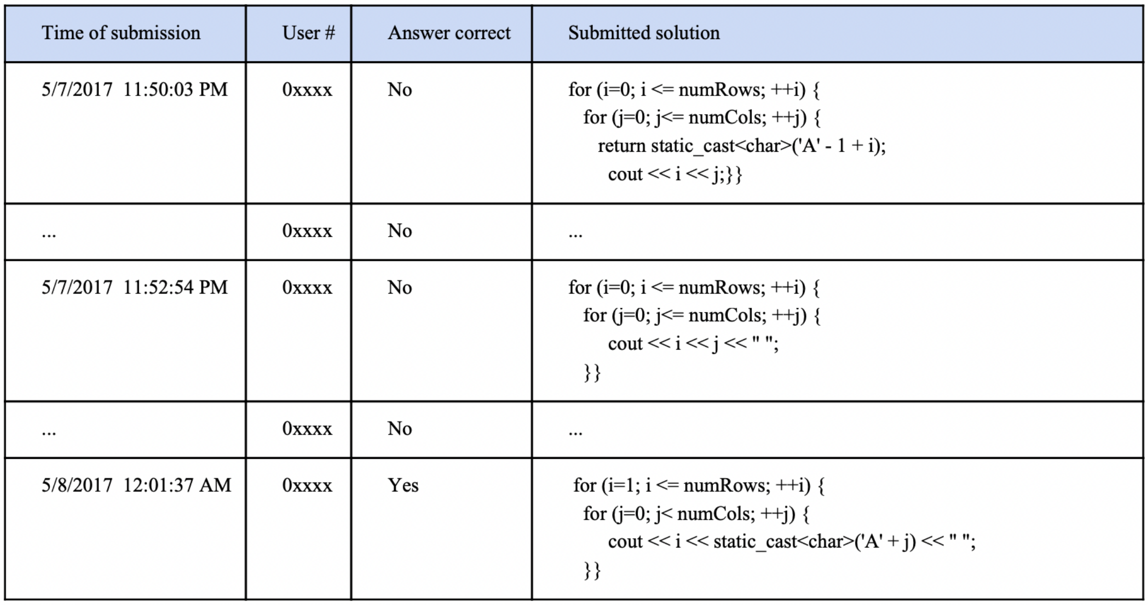
```
1A 1B 1C 2A 2B 2C
```

```
#include <iostream>
using namespace std;

int main() {
   int numRows = 2;
   int numCols = 3;

   // Note: You'll need to define more variables

   /* Your solution goes here  */

   cout << endl;

   return 0;
}
```

Figure 2.1: Coding activity 5_3_2_Nested_loops_Print_seats, on which many students struggled due to the need for nested loops.

Figure 2.1 shows a sample CA, 5_3_2_Nested_loops_Print_seats, with a link “Download student submissions”. A teacher clicks that link to get all student submissions for that CA. Table 2.1 shows a tiny snapshot of that file, showing the corresponding submissions for one particular student for CA 5_3_2_Nested_loops_Print_seats. The downloaded file has 4 columns: Time of submission (timestamp), user ID# (concealed as 0xxxx), whether the answer is correct (Yes/No), and the actual code submitted by the student (including whitespace).

Table 2.1: A snapshot of rows of one particular student's submission for the CA 5_3_2_Nested_loops_Print_seats.



Time of submission	User #	Answer correct	Submitted solution

5/7/2017 11:50:03 PM	0xxxx	No	for (i=0; i <= numRows; ++i) {
			for (j=0; j<= numCols; ++j) {
			return static_cast<char>('A' - 1 + i);
			cout << i << j;}}

...	0xxxx	No	...

5/7/2017 11:52:54 PM	0xxxx	No	for (i=0; i <= numRows; ++i) {
			for (j=0; j<= numCols; ++j) {
			cout << i << j << " ";
			}}

...	0xxxx	No	...

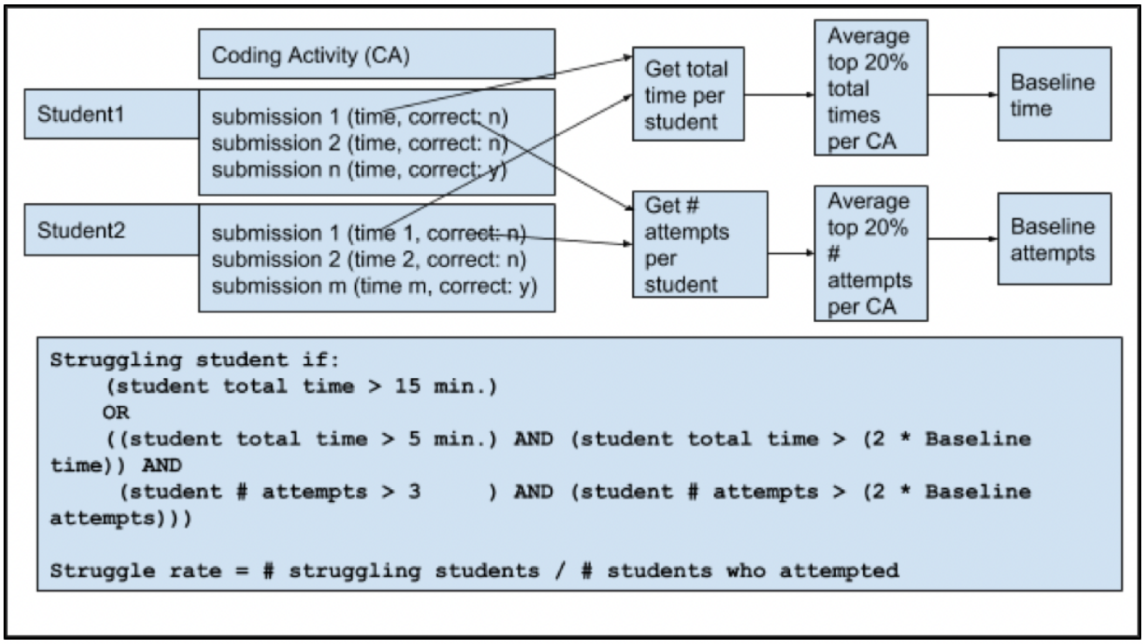
5/8/2017 12:01:37 AM	0xxxx	Yes	for (i=1; i <= numRows; ++i) {
			for (j=0; j< numCols; ++j) {
			cout << i << static_cast<char>('A' + j) << " ";
			}}

The course was CS 1 in C++, having 80 students who were all non-computing majors, mostly in engineering (chemical engineering, bioengineering, etc.) and science (biology, chemistry, physics, math, etc.). The university is a research institution, and the CS department is typically ranked around 50-70 among U.S. CS programs. The content had 78 code activities, distributed through 9 chapters, covered in 9 weeks of a 10-week course. The chapters were: Intro/Input/Output, Variables/Assignments, Branches, Strings/Loops1, Loops2, Functions1, Functions2, Vectors1, and Vectors2. The CAs were due on Sunday at the end of the week the subject was covered in class, and worth 5% of the course grade. On average, student completion of CAs each week was: 98%, 96%, 95%, 85%, 87%, 88%, 88%, 82%, and 74%.

2.2.1.2 Struggle Rate as a Metric

We desired a metric that would highlight CAs that caused students to struggle. Informally, struggle means a student works on a problem inefficiently, using excessive time or repeated attempts, and causing frustration. Ideally, we would measure struggle directly by observing the student and/or asking the student, but such data is not available and hard to obtain. Instead, we need a way to measure the struggle from the data we have, which involves every submission (wrong or right) of every student, each submission's date/time, and the submission's correctness.

One struggle measure is time spent. But, some activities require more time than others, so time alone is insufficient. Another measure is the number of attempts. But, some activities may involve students detecting and correcting simple errors, approaching quickly to a correct solution. Thus, we define a struggle metric that combines time and number of attempts. And, to avoid considering naturally long activities as having struggle, we consider the ratio of time and attempts compared to the "top" 20% of students for each CA. "Baseline time" is the average of the top 20% of student times for a CA. "Baseline attempts" is the average of the top 20% of the student attempts for a CA. Because CA's are designed to take 5-7 minutes, as a catch-all we also define 15 or more minutes as struggle.


Coding Activity (CA)
Student1
submission 1 (time, correct: n)
submission 2 (time, correct: n)
submission n (time, correct: y)
Student2
submission 1 (time 1, correct: n)
submission 2 (time 2, correct: n)
submission m (time m, correct: y)
Get total time per student
Get # attempts per student
Average top 20% total times per CA
Average top 20% # attempts per CA
Baseline time
Baseline attempts

```
Struggling student if:
    (student total time > 15 min.)
    OR
    ((student total time > 5 min.) AND (student total time > (2 * Baseline
time)) AND
     (student # attempts > 3     ) AND (student # attempts > (2 * Baseline
attempts)))

Struggle rate = # struggling students / # students who attempted
```

Figure 2.2: Definition of struggle rate for a particular CA.

Figure 2.2 illustrates our struggle metric calculation. For a CA, we consider each student individually. Student1 has n submissions, each with a time, and the nth being correct. (We ignored any submissions from a student following a correct submission, as that student was likely just experimenting). For Student1, we compute the total time for that student as the nth submission's time minus the first submission's time. Note that this time may be an underestimate, as the time doesn't include the time the student spent reading the instructions and developing the first submission. If two successive submissions are separated by at least 10 minutes, we assume the student was perhaps taking a break (this is not a perfect measure but the best we can do as we cannot directly observe the student), and thus we exclude that time from the total time. For every student (two are shown in Figure 2.2), such total time is computed. We then compute the average of the shortest 20% of such times to yield the baseline time. The same approach is done for the number of attempts per student.

Figure 2.2's bottom part shows how we define that a particular student struggled on a particular CA. The key features are that the student spent more than 2x the baseline time and more than 2x the baseline attempts. Furthermore, to account for very easy CAs, we also require the time to be more than 5 min. And to account for the fact that a few errors on any CA are normal, we require more than 3 attempts. If all four of the above are true, we classify the student as having struggled on that CA. Also, any student spending more than 15 minutes is classified as struggling, since each CA was designed to take about 5-7 minutes. Given that definition of a struggling student, the struggle rate for a particular CA is then just the number of struggling students for that CA divided by the total number of students who attempted that CA.

2.2.1.3 Struggle Rates for the CA's

For reference, Figure 2.3 shows average time and average number of attempts, along with struggle rate, for CAs sorted by struggle rate. While time or number of attempts obviously correlates with struggle rate, the struggle rate metric provides a more pronounced measure. Time alone, or number of attempts alone, may fluctuate due to the inherent complexity of the particular CA, not necessarily indicating student struggle. As we see in Figure 2.3, some CAs have a low number of attempts and spent time but have a high struggle rate. For example, CA 3_3_2_If-else_statement_Fix_errors (which is represented by a blue and yellow dot on the x-axis coding activity 50 value), has low average attempts (less than 3 times) and low average spent time (less than 3 minutes), but has high struggle rate (above 10%). Figure 2.4 shows struggle rates for all CA's. The last bar is the average struggle rate: 12.3%.

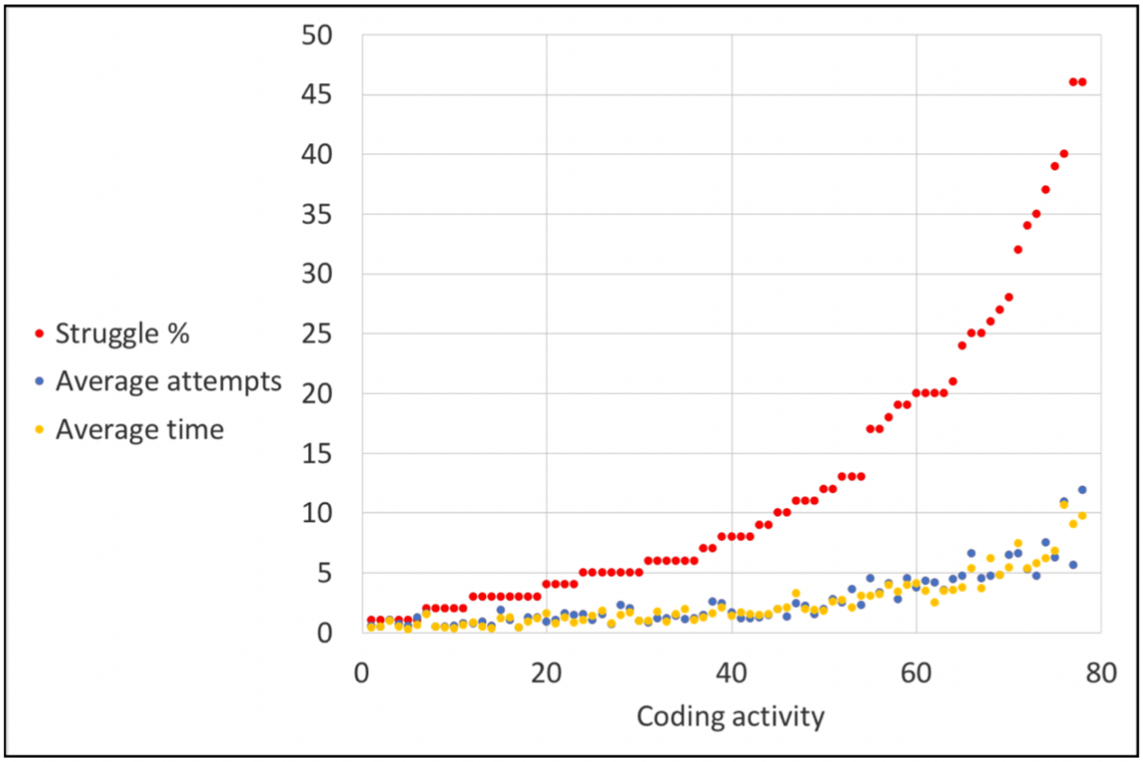


Figure 2.3: Coding activities versus average attempts, average time, and struggle rate (ordered by struggle rate).

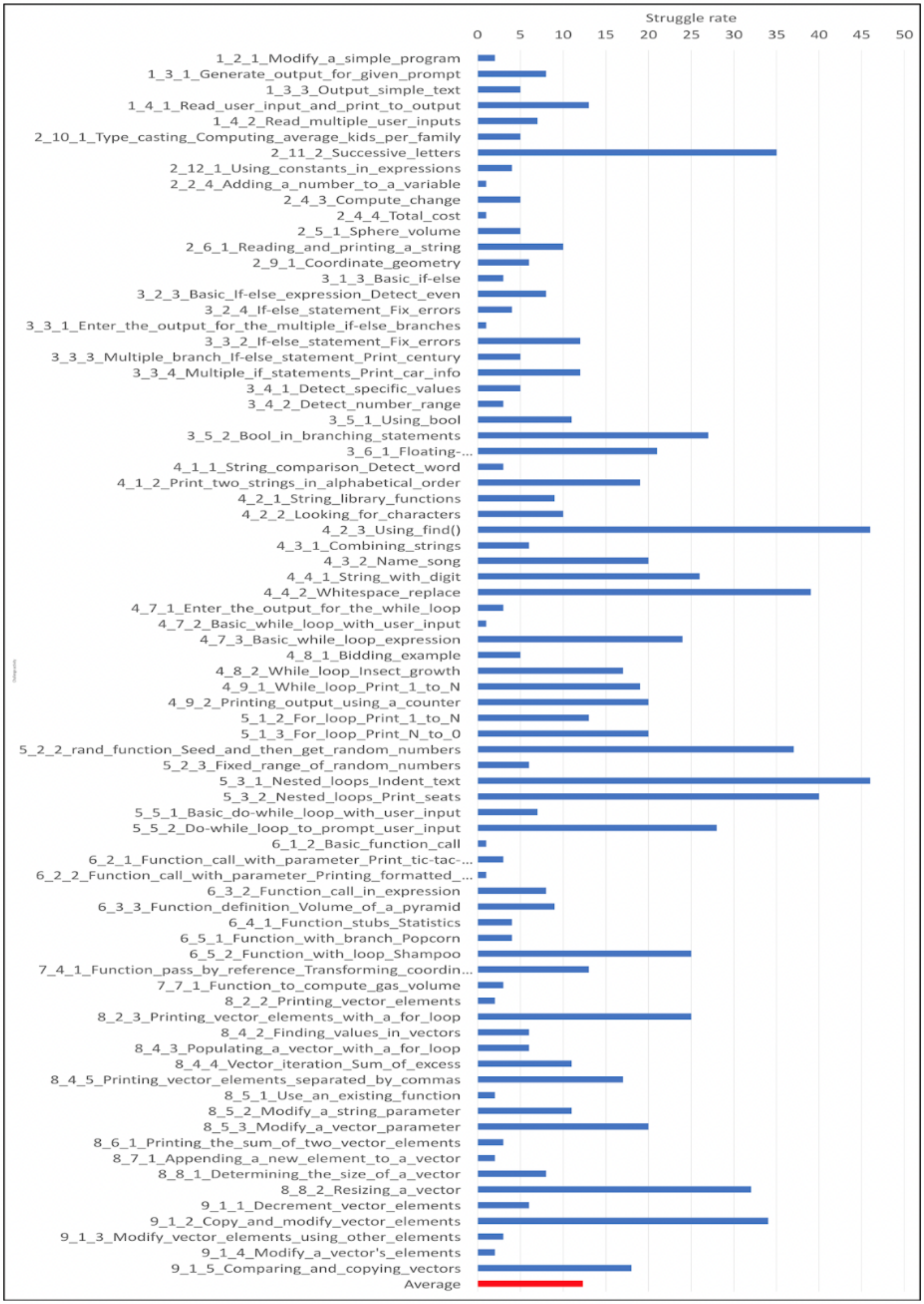


Figure 2.4: Struggle rates for the 78 CAs, in the order the CAs appeared in the 9 chapters of course content.

2.2.1.4. Student Errors for Challenge Activities with the Highest Struggle Rates

Our goal was to understand what errors seemed to cause students to struggle, so that teachers or authors can devise appropriate means for students to avoid or fix those errors and thus reduce struggle (of course while balancing providing help with letting students figure things out themselves).

Table 2.2 highlights the 10 CAs with the highest struggle rates, all being above 25%. For each CA, the table presents the struggle rate, the CA name, the CAs sample correct code, one sample of wrong code from a student, and our conclusion of what errors yielded such struggle based on our manual code analysis of student submissions.

We considered errors as falling into two categories. “Core errors” are errors that likely would occur for a wide variety of courses and programming languages. “Specialized errors” are errors that seem rather specific to a particular function or language feature.

Some common core errors were:

- Nested loops: Students had trouble defining the inner loop’s initialization and expression, and knowing how to produce output within such loops.

Table 2.2: The CAs with the highest struggle rates, and our analysis of the errors leading to the struggle.

%	CA name	Sample correct code	Sample wrong code	Common errors yielding struggle

46%	4_2_3_Using_find()	if (userInput.find("darn") != string::npos) {
		cout << "Censored" << endl;}
		else {
		cout << userInput << endl;}
			if (userInput.find("darn")) {
			cout << "Censored" << endl;
			}
			else {
			cout << userInput << endl;}
				• Missing if statement condition to check if the word is found or not

46%	5_3_1_Nested_loops_Indent_text	for (i = 0; i <= userNum; i++) {
		for (j = 0; j < i; j++) {
		cout << " ";}
		cout << i << endl;}
			for(i = 0; i <= userNum; i++) {
			for(j = 0; j <= userNum; j++) {
			cout << j << endl;}
			cout << " "; }
				• Wrong inner loop condition
				• Wrong cout() locations and arguments

40%	5_3_2_Nested_loops_Print_seats	char currColLet = 'A';
		for (currRow = 1; currRow <= numRows; currRow = currRow + 1) {
		currColLet = 'A';
		for (currCol = 1; currCol <= numCols; currCol = currCol + 1) {
		cout << currRow << currColLet << " ";
		currColLet = currColLet + 1; }}
			char letter = 'A';
			for (i=1;i<=numRows;++i) {
			cout << i ;
			for (j=0;j<=i;++j){
			cout << letter +1 << endl;
			}
			}
				• Missing initialization of "letter" in the outer loop.
				• Wrong initialization of "j" in the inner loop
				• Wrong condition for inner loop
				• Missing "letter" increment

37%	4_4_2_Whitespace_replace	if (isspace(passCode.at(0))) {
		passCode.at(0) = '_';
		}
		if (isspace(passCode.at(1))) {
		passCode.at(1) = '_';
		}
			if (isspace(passCode.at(0))) {
			passCode.replace(0,1,"_");
			}
			else if (isspace(passCode.at(1))) {
			passCode.replace(1,1,"_");}
				• Using else if instead of multiple if

37%	5_2_2_rand_function_Seed_and_then_get_random_numbers	srand(seedVal);
		cout << (rand() % 10) << endl;
		cout << (rand() % 10) << endl;
			srand(time(0));
			cout << (rand() % 9) << endl; cout << (rand() % 9) << endl;
				• Using seed of time(0) (taught earlier) rather than obeying the instruction to use seedVal
				• Using %9 instead of %10

34%	2_11_2_Successive_letters	char letterStart;
		cin >> letterStart;
		cout << letterStart;
		letterStart = letterStart + 1;
		cout << letterStart << endl;
			char letterStartA = 'a';
			char letterStartB = 'b';
			letterStartB = letterStartA + 1;
			cout << letterStartA << letterStartB << endl;
				• Missing cin()
				• Missing cout() after cin()
				• Missing incrementing the only variable
				• Missing printing that variable

34%	9_1_2_Copy_and_modify_vector_elements	for (i = 0; i < SCORES_SIZE; ++i)
		{
		if (i != (SCORES_SIZE-1)) {
		newScores.at(i)= oldScores.at(i+1);
		}
		else{
		newScores.at(i) =oldScores.at(0);}}
			for (i = 0; i < SCORES_SIZE - 1; i++) {
			newScores.at(0) = oldScores.at(oldScores.size() - 1);
			newScores.at(i) =oldScores.at(i - 1);
			}
				• Wrong for-loop condition
				• Missing if statement to update a variable in a loop
				• Wrong use of vector index to update variable

32%	8_8_2_Resizing_a_vector	countDown.resize(newSize);
		
		for (i = 0; i < newSize; ++i) {
		countDown.at(i) = newSize - i;
		}
			for(i = 0; i < SCORES_SIZE - 1; i++){
			if(i == 0){
			newScores.at(i) = oldScores.at(oldScores.size() - 1);}
			else{
			newScores.at(i) = oldScores.at(i - 1);} }
				• Missing resize of the vector
				• Wrong condition for the for-loop
				• Wrong code to change the vector elements

28%	5_5_2_Do-while_loop_to_prompt_user_input	do {
		cout << "Enter a number (<100): \n";
		cin >> userInput;
		} while (!(userInput < 100));
			cin >> userInput;
			do {
			cout << "Enter a number (<100): " << endl;
			cin >> userInput;
			} while (userInput < 100);
				• Wrong condition

27%	3_5_2_Bool_in_branching_statements	if (isBalloon && !isRed) {
		cout << "Balloon" << endl;
		}
		else if (isBalloon && isRed) {
		cout << "Red balloon" << endl;
		}
		else {
		cout << "Not a balloon" << endl;}
			if ((isBalloon != false) && isRed) {
			cout << "Balloon" << endl;
			}
			if ((isBalloon = true) && (isRed = true)) {
			cout << "Red balloon" << endl;
			}
			else {cout << "Not a balloon"<< endl; }
				• Incorrect use of if statement
				• Incorrect condition for the if statement

- Else-if vs. multiple if: Students used else-if in cases where multiple if statements were needed, and vice-versa.
- Random range: Students often used % N rather than % (N+1) to generate random numbers in the range 0 to N.
- Input/output: Students early on had trouble putting output and input statements in the correct order.
- For loop and vector: Students had trouble creating a for loop header when the iteration wasn't through every element.
- For loop and if: Students had difficulty using an if statement inside a for loop when iterating through all elements.
- Vector index: Students had trouble when vector element updates weren't simple assignments while iterating through a vector, using the loop's counter variable as the vector index.
- Negated loop expression: Students often negated a loop expression, using while (x) rather than while (!x).
- Boolean expressions: Students struggled writing simple expressions with Booleans, comparing with false or true unnecessarily, increasing complexity and thus mistakes.

Some common specialized errors were:

- C++ find() function: Students did not understand C++'s unusual return value of the find() function, namely std::npos.

- Character increment: Students had trouble understanding that incrementing a character variable holding a letter yields the next letter in the alphabet.
- Random seeding: Students often improperly seeding a random function, by copying a particular example in our content that seeded with current time, rather than following instructions to seed with a particular variable's value.

2.2.2 Discussion

The above information can be used in many ways, such as spending more time in lectures going over examples focused on avoiding a common error, creating more content for teaching a particular subject matter, improving code auto-graders to detect common errors and provide specific hints (perhaps after the student has spent at least a few minutes trying on their own), etc. The purpose of this work is to summarize the common errors so that others can adjust their teaching/content accordingly.

However, we happened to have some historical data that was relevant to this work, and thus we chose to include that data. In particular, for 10 CAs that we'd noticed in the past that students asked the most questions on (so not based on a struggle metric, which we had not developed at that time), we in January 2017 added a hint link that discussed common errors. For those CAs, we obtained submission data from before the hints were added, and after the hints were added. In fact, we obtained the data from our university which used the same CAs. The analysis compared Spring 2016 (262 students) and Spring 2017 (175 students).

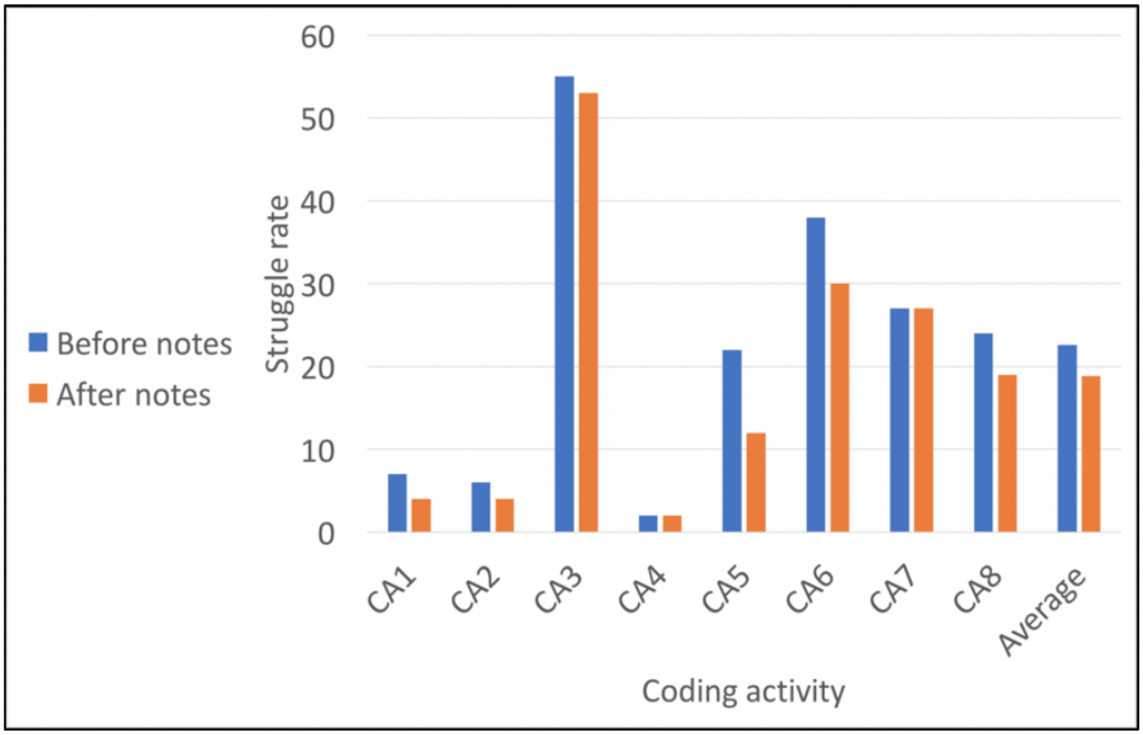


Figure 2.5: CAs versus struggle rate, without hints (blue) and with hints (orange).

Figure 2.5 shows the effectiveness of adding hints to some CAs (CA1 to CA8). The blue and orange bars represent the struggle rate for a CA before and after adding a hint, respectively. The hint drops the struggle rate modestly for most CAs, with a substantial drop for some. On average, the struggle rate for those CAs dropped from 23% to 19%, representing a 17% decrease in the number of struggling students. Note that some CAs had low struggle rates even before the hints were added, indicating that students asking questions about a problem does not necessarily mean that students are struggling on those problems.

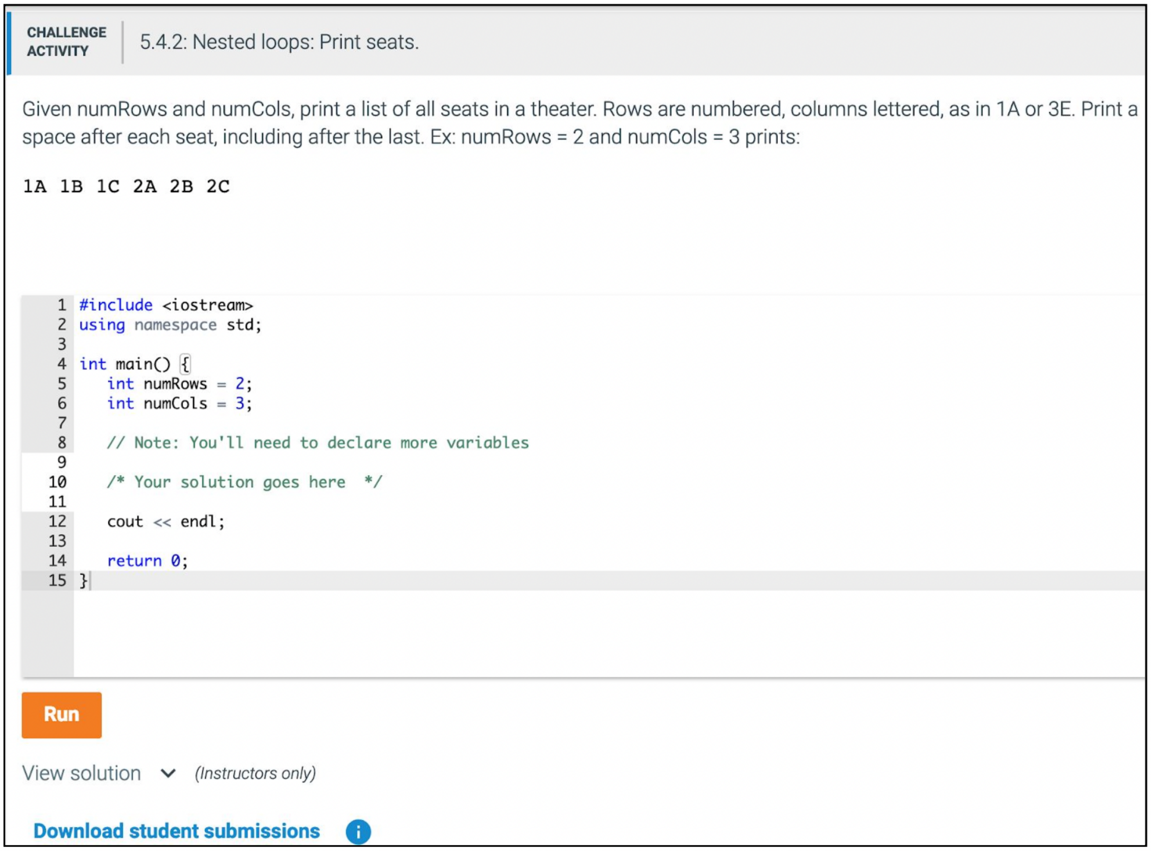
As a reminder, adding a hint link is just one way, and a fairly modest way, to strive to reduce struggle rates. We suggest that more aggressive techniques be utilized.

2.3 Studying Struggle with Detailed Manual Analysis

2.3.1 Methodology

2.3.1.1 Homework Problems

We use a zyBook, an online interactive textbook content for learning programming used by about 500 universities and 200,000 students yearly per the company's information [1]. A zyBook has integrated coding homework problems known as challenge activities (or CAs). Each problem has students finishing a small program by writing a few lines of code, like a for loop as shown in Figure 2.6. Clicking Run compiles and executes the program on a server, tests using various test cases, and shows results. Students can repeat any number of times until passing all test cases. Each problem is designed to take about 3-5 minutes to solve.



5.4.2: Nested loops: Print seats.

Given numRows and numCols, print a list of all seats in a theater. Rows are numbered, columns lettered, as in 1A or 3E. Print a space after each seat, including after the last. Ex: numRows = 2 and numCols = 3 prints:

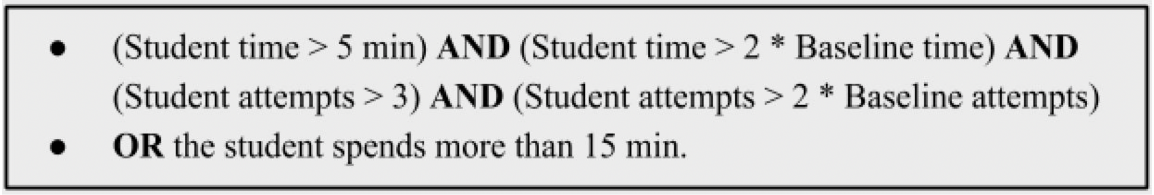
```
1A 1B 1C 2A 2B 2C
```

```
#include <iostream>
using namespace std;

int main() {
   int numRows = 2;
   int numCols = 3;

   // Note: You'll need to declare more variables

   /* Your solution goes here  */

   cout << endl;

   return 0;
}
```

Figure 2.6: A coding activity example: Writing nested loops. This activity has a high struggle rate.

An instructor can download all submissions for any CA, yielding a CSV file having every submission, both incorrect and correct. Each submission has: the timestamp, a user unique ID #, Yes or No (correct or incorrect), and the submitted code.

We downloaded all submissions for all 89 CAs in our zyBook. For each CA, we computed the struggle rate, defined as the % of students who struggled. We used the struggle metric defined by researchers in [2] (other metrics are possible of course):



- (Student time > 5 min) **AND** (Student time > 2 * Baseline time) **AND** (Student attempts > 3) **AND** (Student attempts > 2 * Baseline attempts)
- **OR** the student spends more than 15 min.

Figure 2.7: The struggle metric.

For a gap between a student's submissions of more than 10 minutes, we assume the student stepped away and thus excluded such gaps from time.

2.3.1.2 Manually Finding Errors Experienced by Struggling Students

Using the above automated analysis, we found the 12 CAs with the highest struggle rates. We manually (i.e., human analysis) examined student submissions to determine the common errors, and number of attempts and time spent by students specifically fixing each error.

Human analysis was preferred because submissions commonly had multiple errors. Humans used judgment to determine what error a student was likely working on. For example, the following submissions have three errors: incorrect left-side variable in line 2, incorrect squaring in line 2, and misspelled area variable in line 3.

```
1: int area = 0;

2: r = PI * r * 2;

3: cout << arae << endl;
```

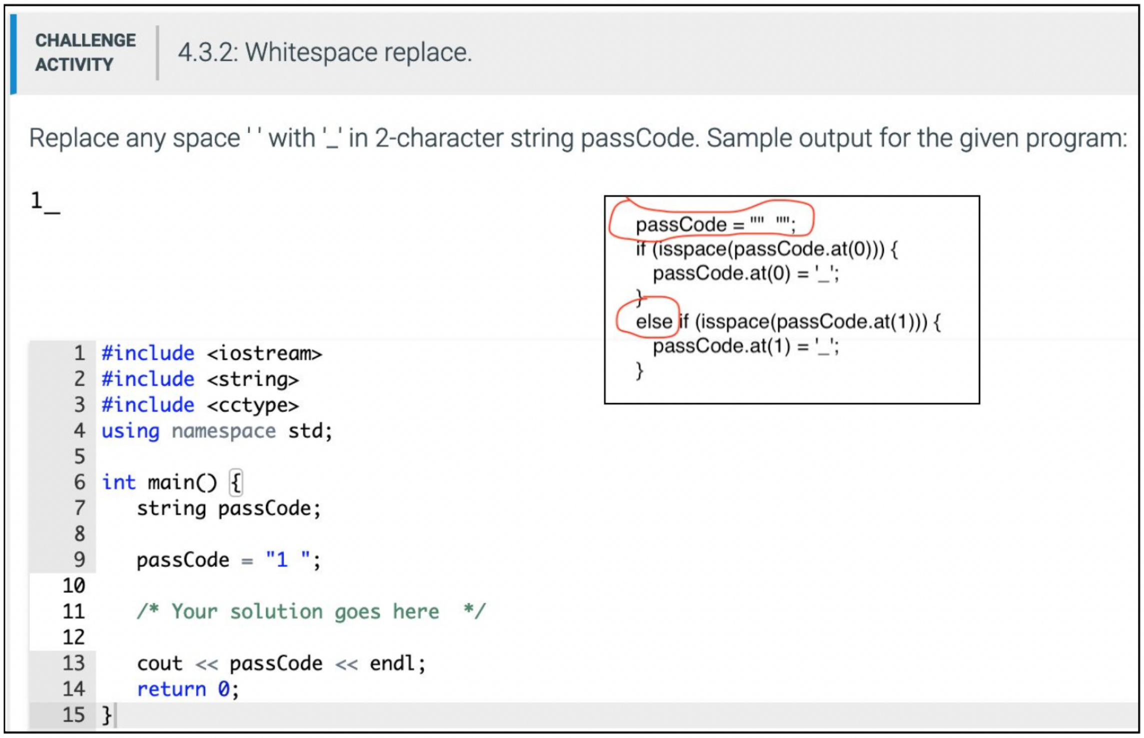
- 1 min later, the student fixes the spelling of area, so we say that 1 min was spent on the spelling error.
- 1 min later, the student adjusts the initial value of area, probably to see the impact on output. We attribute this attempt to the incorrect left-side variable error.
- 1 min later, the student adds a cout of r as well, probably to make sure r's value is as expected. We again attribute this attempt to the left-side error.
- 1 min later, the student changes the left-side to area. We attribute it to the left-side error, which is now fixed.
- In 5 submissions over the next 9 minutes, the student tries changing line 2's expression to PI * 2 * r, then 2 * r * PI, then PI * (r * 2), then PI * r * 2.0, and finally PI * r * r.

Humans can recognize what the errors were, and can attribute 3 attempts and 3 minutes to solve the left-side error and 5 attempts and 9 minutes to the squaring error. Most previous work was automated, and would have considered the squaring error as 8 attempts and 13 minutes because that error existed in each submission, but a human can see that the student was focused on a different error for the first 3 minutes. Such overestimates can be significant in CAs that commonly see multiple errors.

The analyses were carried out by 3 upperclass-student CS majors, with close monitoring by a professor and a postdoctoral researcher. Total time was about 10 hours per CA.

Because our focus is on errors experienced by students who struggle, we manually examine every submission for every struggling student, and count their errors, attempts, and time spent, as in the above example. Our focus was not on non-struggling students, and our resources did not allow manually examining every non-struggling student. Nevertheless, comparing is interesting, so we also manually examined a random subset of non-struggling students and calculated values for their errors, and scaled those values to get estimates for all non-struggling students.

2.3.1.3 Common Errors Among Strugglers



CHALLENGE ACTIVITY | 4.3.2: Whitespace replace.

Replace any space ' ' with '_' in 2-character string passCode. Sample output for the given program:

```
1_
```

```
passCode = "" "";
if (isspace(passCode.at(0))) {
   passCode.at(0) = '_';
}
else if (isspace(passCode.at(1))) {
   passCode.at(1) = '_';
}
```

```
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

int main() {
   string passCode;

   passCode = "1 ";

   /* Your solution goes here  */

   cout << passCode << endl;
   return 0;
}
```

Figure 2.8: Whitespace replace (CA 4.3.2) CA with a student's submission in the top-right corner.

Figure 2.8 shows a sample CA (Whitespace replace CA 4.3.2) with a student's submission in the top-right corner), with student errors circled in red. Figure 2.9 to Figure 2.14 show common errors for 6 out of 12 CAs with the highest struggle rates (56% to

34% struggle rates for those 6 CAs). The struggle rates are calculated according to the struggle metric defined in Figure 2.7. The remaining 6 figures are omitted for space reasons. The results are separated for struggling students and non-struggling students. In the bar charts, a (G) at the end of the error stands for a general error while (S) stands for a problem-specific error. A bar's label x/y means x is the number of students, and y is the median number of submissions containing that error. The time is the average time spent solving that specific error.

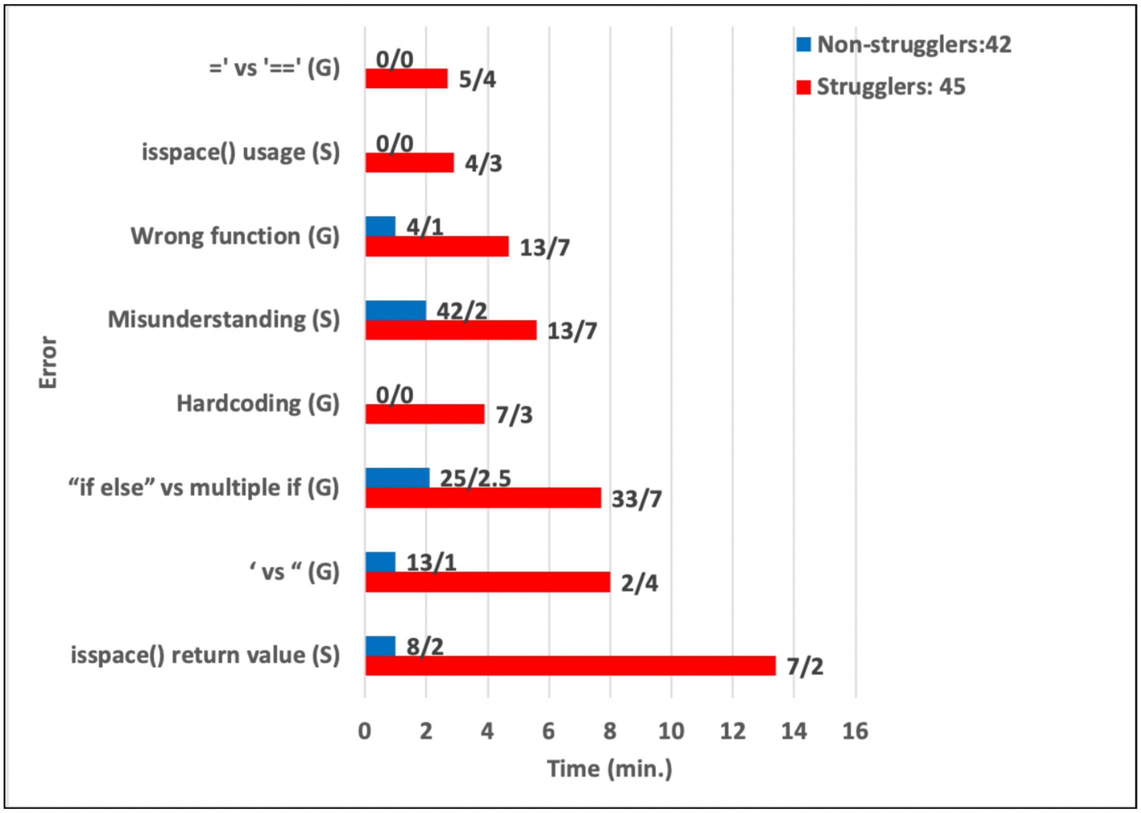


Figure 2.9: Whitespace replace (CA 4.3.2).

Figure 2.9's CA appeared in the fourth week covering branches, and asked students to replace any spaces in a two-character string by an underscore. The error that caused the most struggle was students mis-understanding the return value of library function isspace(), which returns 0 (not a space) or a non-zero value (is a space) -- 7 students compared with true or 1 instead of just using "if (isspace(x))", causing struggle averaging 13.4 minutes. 8 other students did so too, but quickly fixed their mistake. 2 students struggled due to using ' instead of " or vice-versa. 33 students struggled due to using if-else instead of multiple if statements. 7 students struggled due to trying to hardcode a solution based on the test cases (trying to "cheat" the auto-grader). 13 students struggled due to misunderstanding the question, trying to solve a different problem (something that human analysis could determine, but a fully-automated analysis would not have). 13 struggled due to using a wrong library function other than isspace (such as isalpha()).

The CA in Figure 2.10 was also in the fourth week with branches, asking students to set a boolean with true if a 3-character string contained a digit. 33 students struggled due to syntax errors. In fact, 11 used camel-case IsDigit as taught by the textbook, rather than the lowercase of the isdigit() library function, and took much time realizing the error. 24 struggled due to not checking each character individually. We omit further discussion of the errors for this CA, and for subsequent CAs, as the discussions are similar.

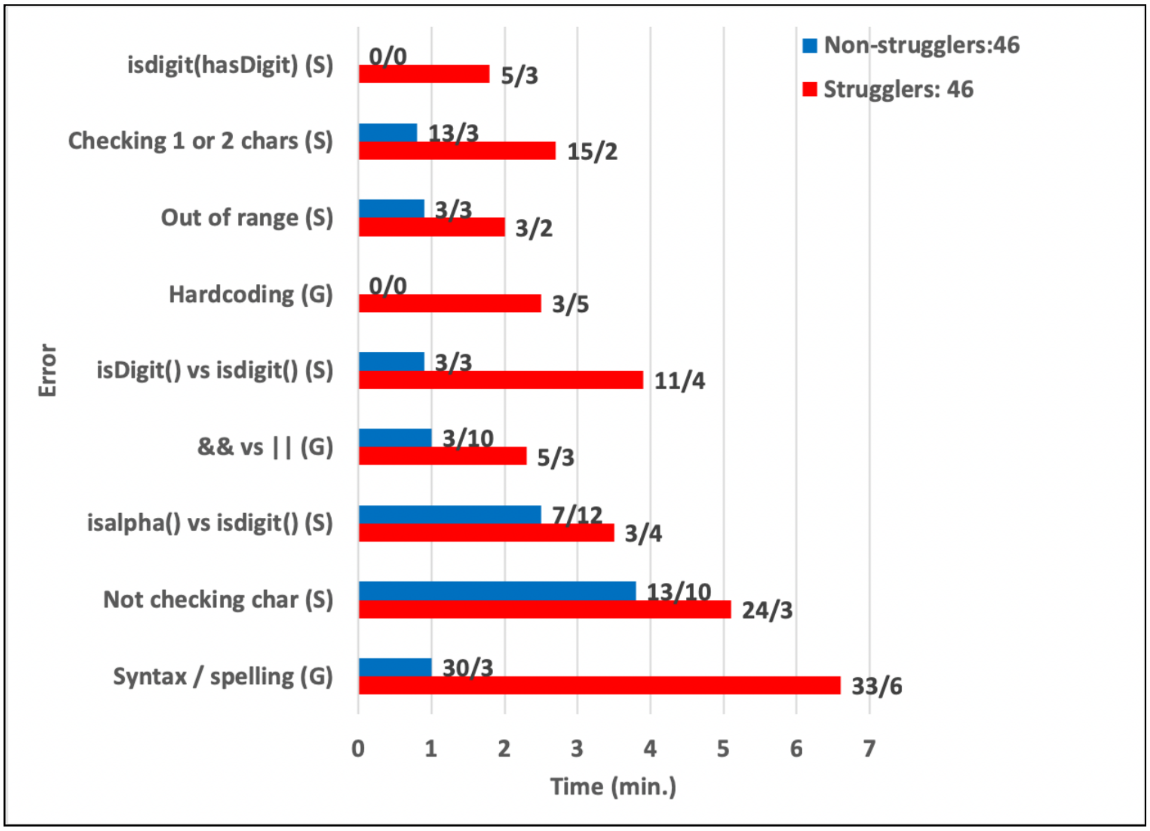


Figure 2.10: String with digit (CA 4.3.1).

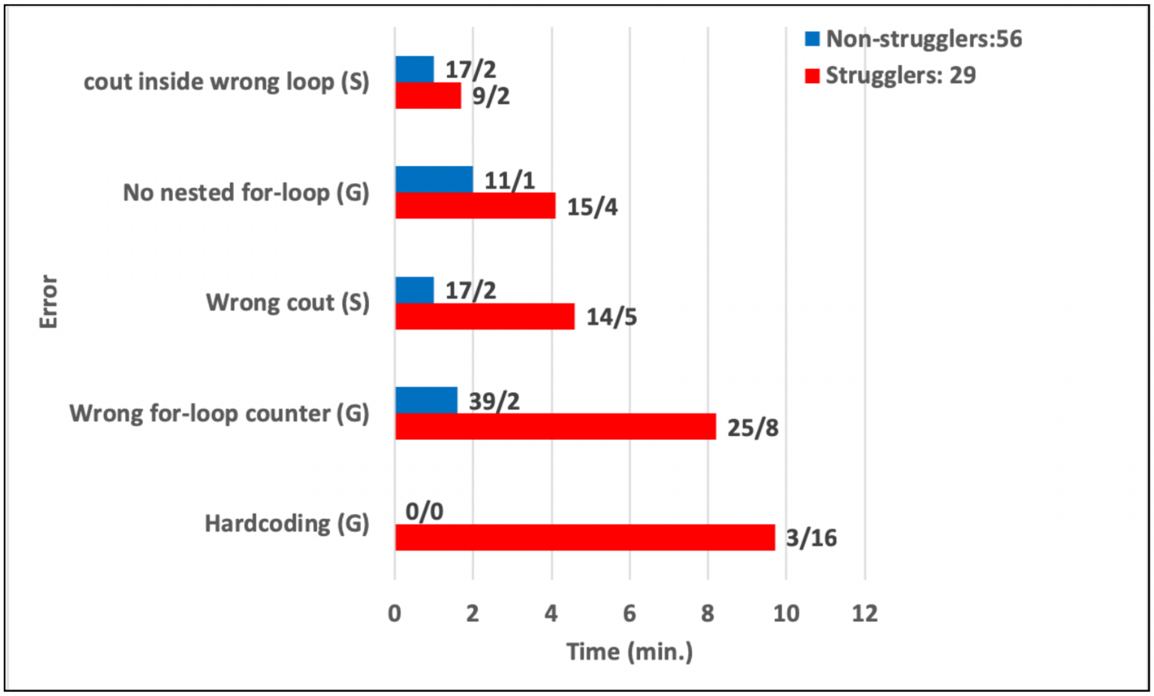


Figure 2.11: Nested loops Indent (CA 5.4.1).

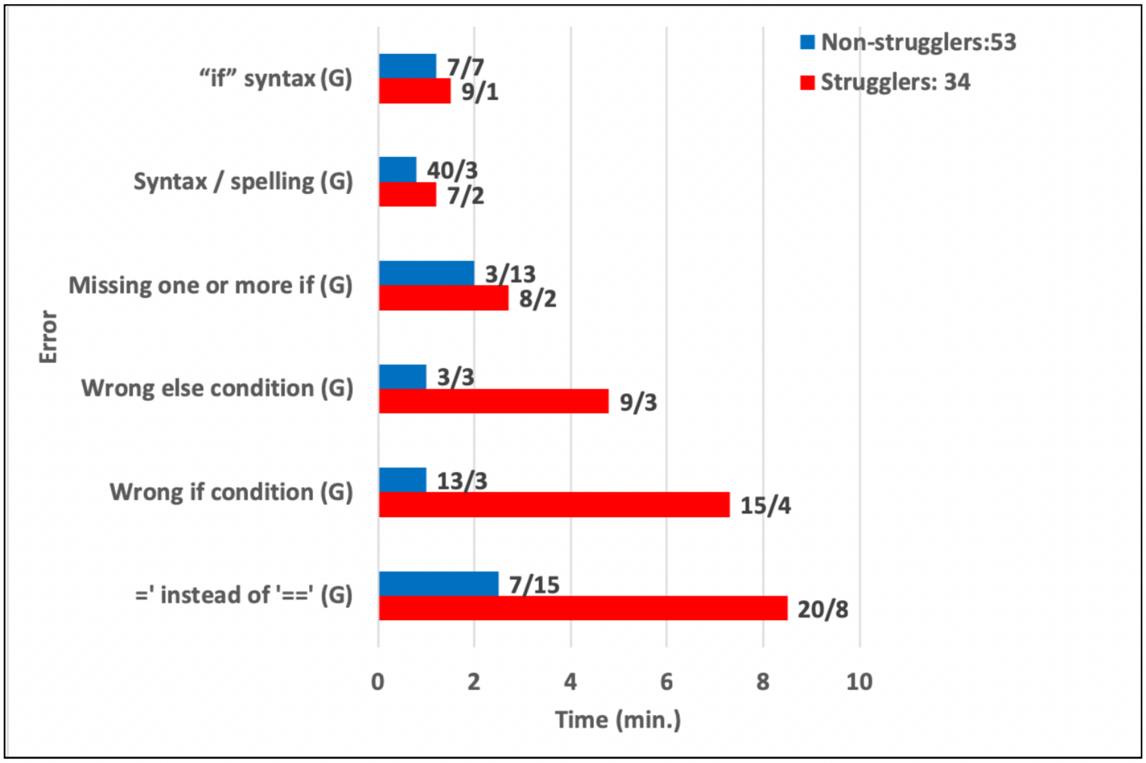


Figure 2.12: Bool in branching statements (CA 3.10.2).

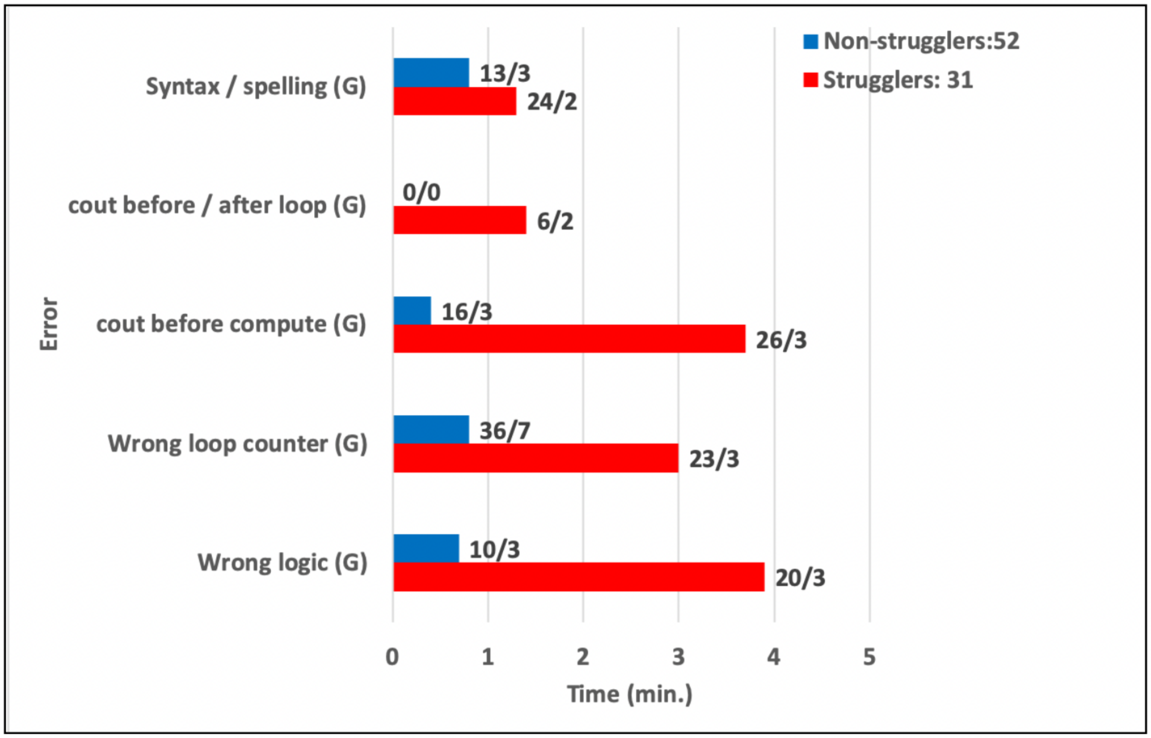


Figure 2.13: Basic while loop expression (CA 4.7.3).

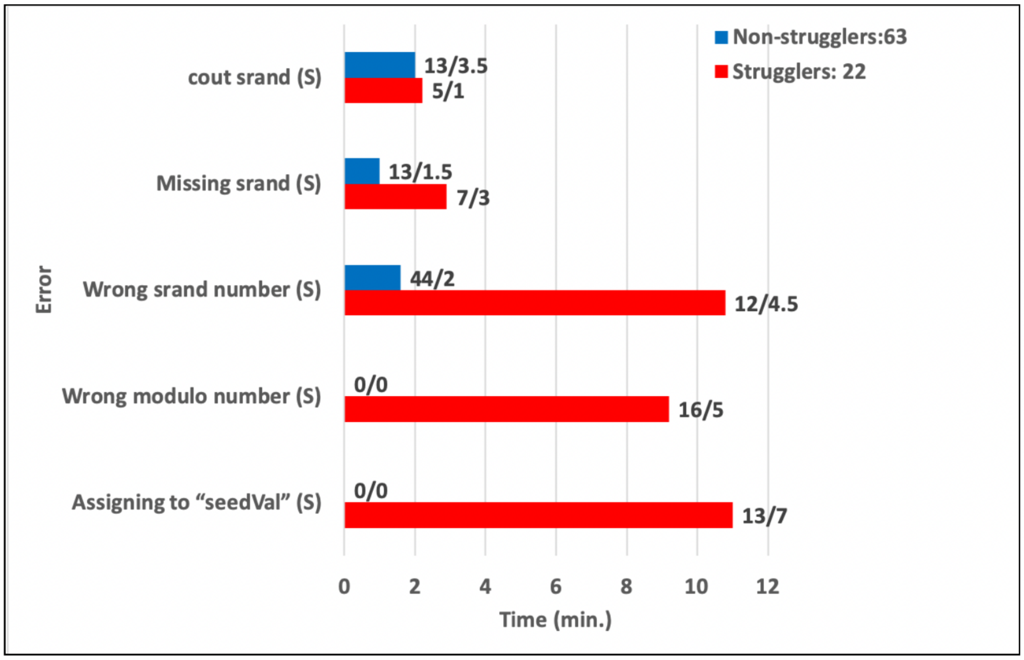


Figure 2.14: Rand function seed and then get random numbers (CA 5.5.2).

2.3.2 Discussion

2.3.2.1 Common Errors Discussion: General vs. Problem-Specific

A key finding of our analysis is that while the top struggle causing errors included "general" programming errors, they also included "problem-specific" errors. General errors include errors like confusing ' and ", confusing if-else and multiple ifs, confusing && and ||, using = rather than ==, or using a wrong if condition.

Table 2.3: Most common general errors.

Error
Using if and else instead of else if
Using = to compare
Syntax / spelling
For loop/while loop conditions/logic (infinite loop, not updating variable, etc)
Wrong if condition logic
Confusing single quotes with double quotes
Cout before cin
Logic error (answer logic, using multiplication, subtraction, division, strings, etc)

In contrast, problem-specific errors were very specific to a particular CA, such as misunderstanding the return value of isspace(), using the wrong library function like using isalpha() instead of isdigit(), placing a cout statement before a computation in a loop (causing values to be off by one iteration versus the desired output), or failing to seed a random number generator per the problem's instructions. These errors are unlikely to show up in a typical analysis of common errors, yet are just as problematic for students. In fact, some errors even depend on where the problem appears: the isDigit() vs. isdigit() error in Chapter 4 (Strings/Loops 1) is due to the CA being the first to use a two-word library function -- Chapter 2 (Variables/Assignments) requires only one-word math functions like pow() and sqrt().

While those problem-specific errors could be generalized as "misusing functions" or "mis-ordering statements", such generalizations do not afford teachers much opportunity to improve and provide little chance for automated hints.

Thus, a key finding of our analysis is that error analysis should include detecting problem-specific errors, so teachers can provide improved problem-specific instructions, and an automated help system should allow adding problem-specific hints. For example, for CA 4.3.2 in Figure 2.10, a hint system could auto-detect "isspace(x) == true" and provide a specific hint like "isspace() returns 0 or non-zero; your code has isspace(x) == true. Should not assume non-zero is true or 1. Instead, use just isspace(x)." Creating an exhaustive list of misuses of all library functions is unreasonable; focusing on the particular errors causing struggle is more feasible. Similarly, a problem that requires seeding a random number generator can detect absence of srand() and give a hint like "This problem requires srand() to seed the random number generator, but your code is missing srand()." Such hints can reduce much struggle while keeping the student learning.

For completeness, we list the most common general errors in Table 2.3, as done in most prior research. Our list has many similarities to previous lists, being a subset due to us focusing on 89 CAs from a zyBook, and also due to us focusing on errors that caused struggle.

2.3.2.2 Top Struggle Situations for Struggle-Causing Errors

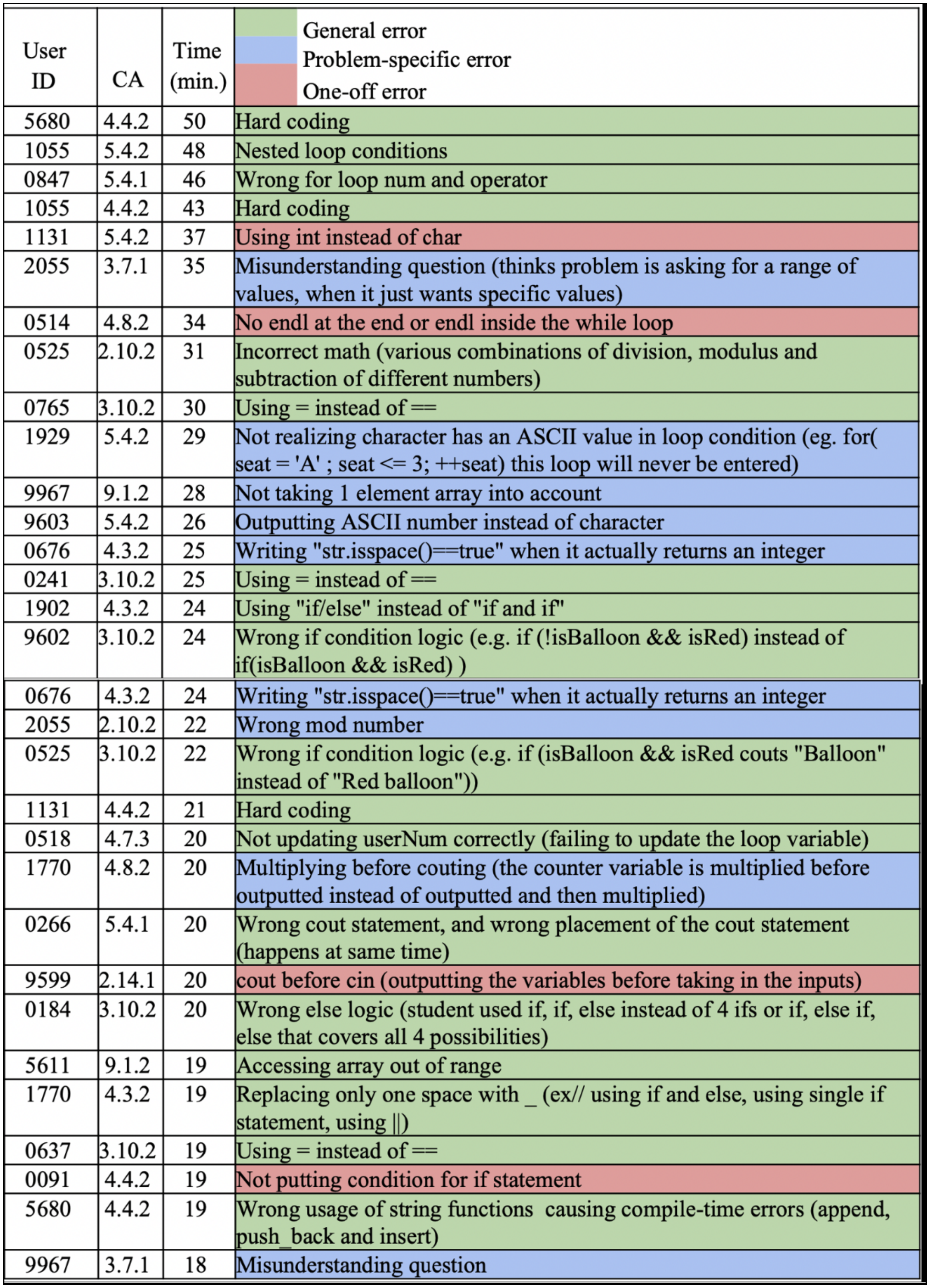
We were curious to know what particular errors caused the worst struggle situations. For the 12 CAs that we manually examined and recorded the time per error for each student, we created a list of the time spent by each student on each CA on each error, and then sorted in descending order. Table 2.4 shows the top 30. We see that the worst scenarios are due largely to both general and problem-specific errors, reinforcing the conclusion that we must focus on both if we wish to help students.

But additionally, we see that some errors that caused the worst struggle scenarios are not common errors, neither general nor problem-specific, but rather are just "silly" mistakes that students made and didn't recognize. We call these "one-off" errors. They appeared throughout the top 100 scenarios as well.

For example, one student accidentally declared a variable as int instead of char, spending 37 minutes because of that error. Another student just couldn't get a newline (endl) placement correct to pass the auto-grading, spending 34 minutes. One student failed to include a condition in an if statement, having an open parenthesis, and simply could not understand the compiler error, spending 19 minutes. Looking further in the list, we saw a semicolon after an else keyword (which is not a syntax error) causing struggle. On the one hand, experiencing and fixing errors is part of learning; on the other hand, no human teacher would leave a student to struggle for 15-30 minutes because of such "silly" mistakes.

Thus, a second conclusion from our analyses is that one-off errors, though uncommon, should be detected if possible (like semicolon after else); and if not possible, students spending excessive time need a way to obtain quick help.

Table 2.4: Top 30 errors.



User ID	CA	Time (min.)	🟩 General error 🟦 Problem-specific error 🟥 One-off error
5680	4.4.2	50	🟩 Hard coding
1055	5.4.2	48	🟩 Nested loop conditions
0847	5.4.1	46	🟩 Wrong for loop num and operator
1055	4.4.2	43	🟩 Hard coding
1131	5.4.2	37	🟥 Using int instead of char
2055	3.7.1	35	🟦 Misunderstanding question (thinks problem is asking for a range of values, when it just wants specific values)
0514	4.8.2	34	🟥 No endl at the end or endl inside the while loop
0525	2.10.2	31	🟩 Incorrect math (various combinations of division, modulus and subtraction of different numbers)
0765	3.10.2	30	🟩 Using = instead of ==
1929	5.4.2	29	🟦 Not realizing character has an ASCII value in loop condition (eg. for(seat = 'A' ; seat <= 3; ++seat) this loop will never be entered)
9967	9.1.2	28	🟦 Not taking 1 element array into account
9603	5.4.2	26	🟦 Outputting ASCII number instead of character
0676	4.3.2	25	🟦 Writing "str.isspace()==true" when it actually returns an integer
0241	3.10.2	25	🟩 Using = instead of ==
1902	4.3.2	24	🟩 Using "if/else" instead of "if and if"
9602	3.10.2	24	🟩 Wrong if condition logic (e.g. if (!isBalloon && isRed) instead of if(isBalloon && isRed))
0676	4.3.2	24	🟦 Writing "str.isspace()==true" when it actually returns an integer
2055	2.10.2	22	🟦 Wrong mod number
0525	3.10.2	22	🟩 Wrong if condition logic (e.g. if (isBalloon && isRed couts "Balloon" instead of "Red balloon"))
1131	4.4.2	21	🟩 Hard coding
0518	4.7.3	20	🟩 Not updating userNum correctly (failing to update the loop variable)
1770	4.8.2	20	🟦 Multiplying before couting (the counter variable is multiplied before outputted instead of outputted and then multiplied)
0266	5.4.1	20	🟩 Wrong cout statement, and wrong placement of the cout statement (happens at same time)
9599	2.14.1	20	🟥 cout before cin (outputting the variables before taking in the inputs)
0184	3.10.2	20	🟩 Wrong else logic (student used if, if, else instead of 4 ifs or if, else if, else that covers all 4 possibilities)
5611	9.1.2	19	🟩 Accessing array out of range
1770	4.3.2	19	🟩 Replacing only one space with _ (ex// using if and else, using single if statement, using ||)
0637	3.10.2	19	🟩 Using = instead of ==
0091	4.4.2	19	🟥 Not putting condition for if statement
5680	4.4.2	19	🟩 Wrong usage of string functions causing compile-time errors (append, push_back and insert)
9967	3.7.1	18	🟦 Misunderstanding question

2.4 Conclusion

In this chapter, we measured student struggle rates on homework problems (coding activities / CAs) in an introductory CS 1 programming course. We found struggle rates to be 10-15%, and as high as 30-40% for some CAs. We listed common errors that led to struggle, such as errors related to nested loops, use of else-if rather than multiple ifs, etc. We analyzed historical data showing that adding hints to particular CAs could reduce struggle rate. We hope that publishing these common errors on homework problems of a CS 1 class will aid teachers and publishers to devise aggressive approaches to help students avoid or fix such errors. To be clear, we believe some struggle is necessary to learn programming; our goal is to address errors that lead to excessive struggle. Hints are one way to reduce struggle but other ways exist. Also, we do not believe that automatically providing hints for every student error, as done in some auto-grading homework systems, is the best approach, as that approach may lead to excessive dependency and may hamper critical thought or effort. Our goal instead is to focus on the specific errors that yield excessive struggle. In future work, we will build debugging activities focused on reducing the errors found in this work. We also hope to automatically detect these common errors and to provide custom hints. Both approaches we hope will reduce student struggle.

In section 2, we analyzed our students' submissions on 89 auto-graded coding Challenge Activities (CAs) from a zyBook used in our CS 1 class. For the 12 CAs with the highest struggle rates, we manually analyzed what errors caused struggle. We found many were due to common general errors, as found in various previous works. However, we also found that many errors were specific to a particular CA ("problem-specific"), such as misusing a particular library function, or misunderstanding instructions. We conclude that problem-specific errors are as important to detect as general errors, so that teachers can focus their teaching or modify instructions to reduce those errors. Moreover, automated hint systems should allow creating problem-specific hints based on such errors. Furthermore, we noticed that one-off errors -- errors that were uncommon -- were causing some of the worst struggle scenarios for students. These errors are neither general nor problem-specific errors. We thus also conclude that help systems should strive to detect as many one-off errors as possible and provide hints for those (the list may be huge), and that students struggling for more than some period of time should have a way to get quick help. We intend to make use of these findings to improve our own teaching and content, and to begin developing an automated help system for coding homework problems.

3. Common Logic Errors for Programming Learners: A Three-Decade Literature Survey

3.1 Introduction

One contributor to poor CS1 performance is students struggling with programming errors. Thus, numerous researchers over the past decades have published errors made by students learning programming, hoping to aid instructors, authors, and tool developers in helping students detect or avoid such errors. However, publications report different subsets of errors, due to variations in the language used, in the assignments students worked on, in the tools and instructional materials used, and in the help provided to students. We thus reviewed the publications to develop a more comprehensive summary of the common errors made by novice programmers. We focus on logic errors rather than syntax errors. A syntax error is a program error that violates language rules and thus prevents execution. For compiled languages, a syntax error results in a compiler message, typically pointing to the erroneous program line. An example message is "Line 23: Missing semicolon". Syntax errors may annoy students and cause some struggle, but our experience is that logic errors cause more struggle. Syntax errors are covered by other works, such as Hristova [1] or Denny [2]. In contrast, a logic error appears in a syntactically-correct program that compiles and runs, but incorrectly attempts to solve the assignment given to the student programmer. An example is a loop that should iterate through an array but incorrectly stops one short of the array's last element. In our teaching experience, logic errors can be harder to detect and find than (most) syntax errors, and are a more common cause of substantial student struggle.

During our review, we found two publications, [4, 5], to be of particular interest due to not just reporting common errors, but also indicating the time required by students to find and fix the errors. Time is important, because some common logic errors are straightforward to find and fix, such as dividing by 0, as in sumItems / numItems where numItems is 0. That error may result in a runtime message that guides students directly to the offending program line and helps students immediately realize the problem, and whose fix may be a relatively straightforward check for 0 before dividing. On the other hand, some common logic errors are much harder to find and fix, such as performing integer division when intending for floating-point division, as in f = (9/5)*c + 32 (in Java, C, C++, etc.). That error may cause incorrect output, but the student doesn't know that the 9/5 (should be 9.0/5.0) or even that code line is the problem, and thus may try many different things, spending a lot of time and leading to struggle.

Altadmri [4] automatically analyzed 37 million compilations from 250,000 students learning Java using BlueJ to find runtime errors, considering frequency and time-to-fix, yielding a list of time-consuming errors like confusing &&/|| and &/|, using == instead of .equals for strings, ignoring a method's return value, putting a semicolon after if, and dozens more. Median time-to-fix for the above was 17 min to 7 min. Ettles [5] analyzed 51,000 submissions from 809 students solving 10 problems in C using CodeWrite, yielding time-consuming errors of: accessing an invalid array element, off-by-one errors, boundary errors, and more, with median times ranging from 20 min to

8 min. As will be seen, we give special weight to the time-consuming errors found by these two publications.

Our goal is to assist instructors, authors, and tool developers who wish to adapt their teaching techniques, learning content, languages, tools, and automated help systems to assist students in detecting or avoiding common logic errors.

3.2 Methodology

3.2.1 Literature Review Method

To find publications relating to common errors, we carried out two tasks. (1) We searched on Google Scholar from 1985 to 2018 for a reasonable combination of these relevant (i.e., related to common novice-programmer logic errors) keywords: program, programmer, CS1, error, mistake, bug, fix, problem, novice, difficult, misconception, and manually examined titles of the search results to find relevant publications. (2) We manually examined the titles of every paper published from 2008 to 2018 in the following 8 conferences and journals, which include a focus on CS education topics: the American Society for Engineering Education Annual Conference (ASEE), the ASEE Computers in Education Journal (CoED), the ACM Global Computing Education Conference (CompEd), the Frontiers in Education Conference (FIE), the International Computing Education Research Conference (ICER), the Innovation and Technology in Computer Science Education Conference (ITiCSE), the Special Interest Group on Computer Science Education Conference (SIGCSE), and the ACM Transactions on Computing

Education journal (TOCE). For both tasks, if a title seemed relevant (i.e., related to common novice-programmer logic errors), we manually examined the publication to see if the publication reported on common novice-programmer logic errors, and ultimately found 47 relevant publications.

For these 47 publications, we logged the errors described in each publication. The result was a raw list of 400 errors, but after processing these errors (remove duplication, remove non-CS1 related such as OOP, pointers, etc., and not counting generic errors such as "branching errors", "selection errors", etc.), we reduced the list to 166 errors. We classified the remaining specific 166 errors into 11 error categories based on commonality: input (2 errors), output (1 error), variable (7 errors), computation (21 errors), condition (18 errors), branching (14 errors), loop (27 errors), array (5 errors), function (24 errors), conceptual (43 errors), and Miscellaneous (4 errors), as shown in Table 3.1. Of the 166, we highlighted (in bold) 43 that appear in more than one paper. Of those 43, we further highlighted (with an asterisk) 11 that were reported to be time-consuming by either Altadmri et al. [4] or Ettles et al. [5]. Section “Common Errors” shows the errors within each category in a tabular format in Table 3.2 to Table 3.12.

For the categorization, we did not use any algorithm or formal process to develop the categories. We used our own judgments based on our own experiences to group together related errors based on commonality in 11 categories. The classification was done by the first author and was reviewed as needed by the second author. For verification and correctness, the first author redid the categorization multiple times. Different categorizations are possible of course; no one categorization is exclusively "correct".

Table 3.1: 11 error categories for the 166 errors in the of the most common error in each category.

No.	Category	# errors	Most common errors
1	Input	2	Erroneous prompting
2	Output	1	Order of output statements
3	Variable	7	Uninitialized variables
4	Computation	21	Integer division
5	Condition	18	&& and || operators
6	Branching	14	Multiple If vs If-else
7	Loop	27	Loop counter
8	Array/String	5	Indexing
9	Function	24	Return value
10	Conceptual	43	Lack of plan
11	Miscellaneous	4	Typos
Total no. of errors			166

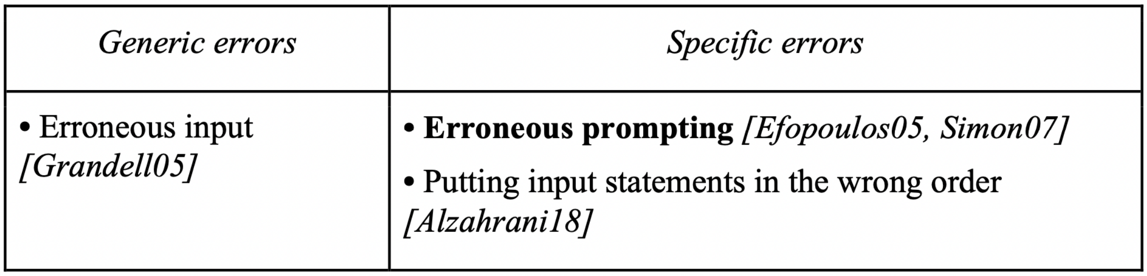
Our study focuses on logic errors typically found in CS1 courses; therefore, we exclude errors commonly found in later courses (or late in some CS1 courses) like object-oriented programming (classes, objects, interfaces, inheritance, overriding, constructors, accessors/modifiers, etc.), recursion, pointers/references, exception handling, data structures beyond arrays and strings such as linked lists, trees, and graphs, GUI and event-driven programming, etc.

Obviously, we cannot provide explanations and examples for all 166 logic errors. Instead, we highlighted in bold errors reported in multiple publications, and highlighted with an asterisk errors that [4, 5] found to be the most time-consuming, yielding 43 highlighted errors. References are included for all 211 errors, however, so that a reader can find details in previous publications of any error of interest.

3.2.2 Common Errors

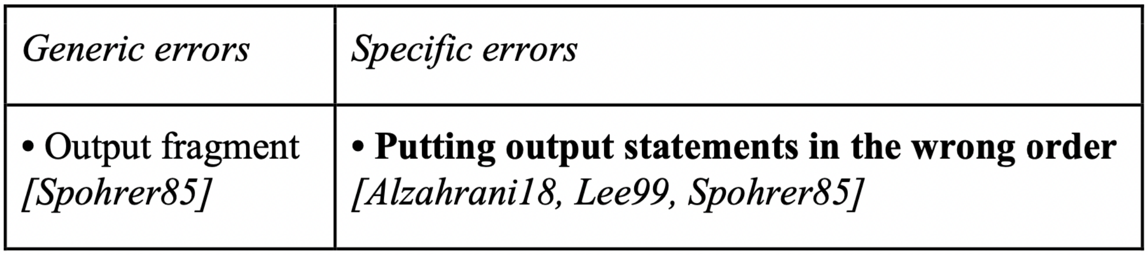
Table 3.1 lists the 11 error categories we defined for the 166 errors, with a column showing the number of errors in each category and another column for the most common error in each category. The sections below provide further details on those 166 errors. For each error category, we use a tabular format (Table 3.2 to Table 3.12) to decompose generic and specific errors that we defined, each with a bulleted list of errors. Generic and specific errors include citations (in the form of the first author last name and the last two digits of the year of publication) of several publications (but not necessarily all) that discussed the item. Note that some publications only discussed errors generically (like "Input errors") while others described specific errors (like "Waiting for input without prompting"). For each table, we highlighted common errors in bold and with an asterisk (as we discussed at the end of the previous section).

Table 3.2: The 2 input specific errors.



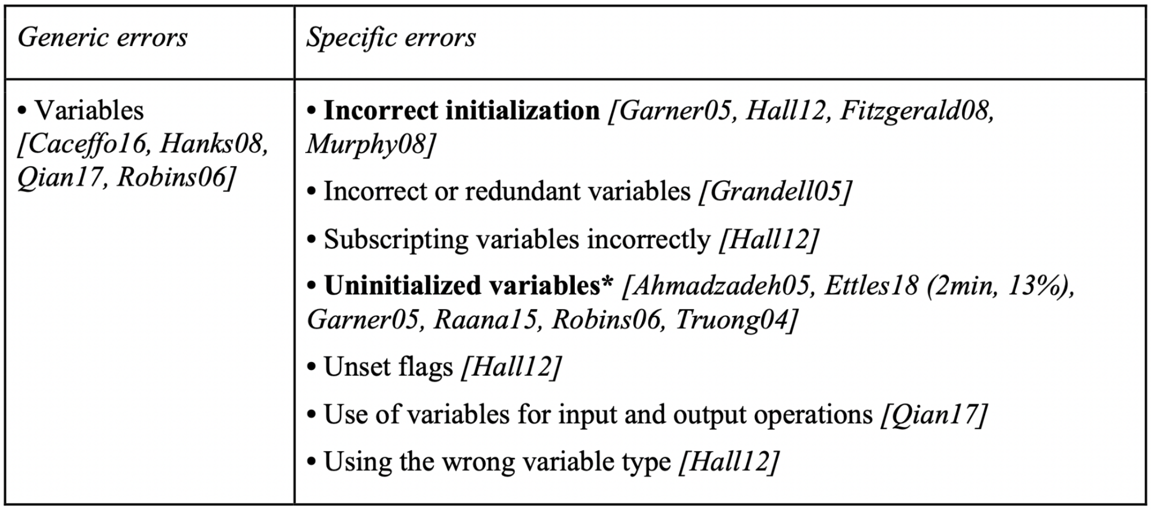
Generic errors	*Specific errors*
• Erroneous input *[Grandell05]*	• **Erroneous prompting** *[Efopoulos05, Simon07]* • Putting input statements in the wrong order *[Alzahrani18]*

Table 3.3: The 1 output specific error.



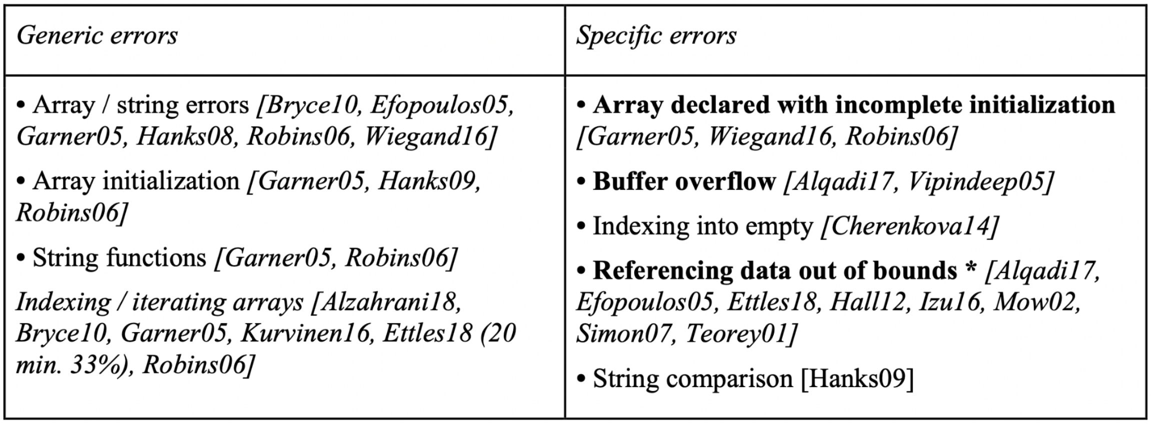
Generic errors	*Specific errors*
• Output fragment *[Spohrer85]*	• **Putting output statements in the wrong order** *[Alzahrani18, Lee99, Spohrer85]*

Table 3.4: The 7 variable specific errors.



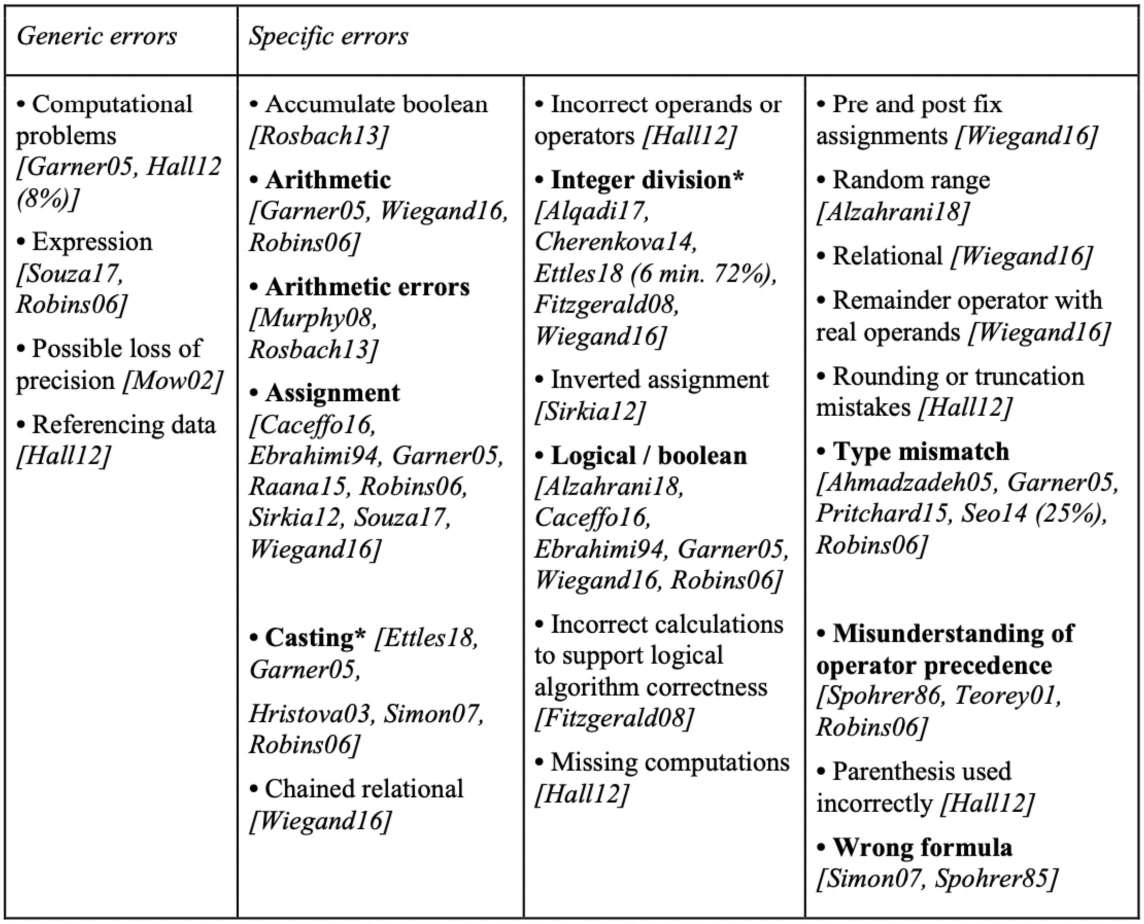
Generic errors	*Specific errors*
• Variables *[Caceffo16, Hanks08, Qian17, Robins06]*	• **Incorrect initialization** *[Garner05, Hall12, Fitzgerald08, Murphy08]* • Incorrect or redundant variables *[Grandell05]* • Subscripting variables incorrectly *[Hall12]* • **Uninitialized variables*** *[Ahmadzadeh05, Ettles18 (2min, 13%), Garner05, Raana15, Robins06, Truong04]* • Unset flags *[Hall12]* • Use of variables for input and output operations *[Qian17]* • Using the wrong variable type *[Hall12]*

Table 3.5: The 21 computation specific errors.



Generic errors	*Specific errors*
• Array / string errors *[Bryce10, Efopoulos05, Garner05, Hanks08, Robins06, Wiegand16]* • Array initialization *[Garner05, Hanks09, Robins06]* • String functions *[Garner05, Robins06]* *Indexing / iterating arrays [Alzahrani18, Bryce10, Garner05, Kurvinen16, Ettles18 (20 min. 33%), Robins06]*	• **Array declared with incomplete initialization** *[Garner05, Wiegand16, Robins06]* • **Buffer overflow** *[Alqadi17, Vipindeep05]* • Indexing into empty *[Cherenkova14]* • **Referencing data out of bounds *** *[Alqadi17, Efopoulos05, Ettles18, Hall12, Izu16, Mow02, Simon07, Teorey01]* • String comparison [Hanks09]

Table 3.9: The 5 array specific errors.



Generic errors	*Specific errors*		
• Computational problems *[Garner05, Hall12 (8%)]* • Expression *[Souza17, Robins06]* • Possible loss of precision *[Mow02]* • Referencing data *[Hall12]*	• Accumulate boolean *[Rosbach13]* • **Arithmetic** *[Garner05, Wiegand16, Robins06]* • **Arithmetic errors** *[Murphy08, Rosbach13]* • **Assignment** *[Caceffo16, Ebrahimi94, Garner05, Raana15, Robins06, Sirkia12, Souza17, Wiegand16]* • **Casting*** *[Ettles18, Garner05,* *Hristova03, Simon07, Robins06]* • Chained relational *[Wiegand16]*	• Incorrect operands or operators *[Hall12]* • **Integer division*** *[Alqadi17, Cherenkova14, Ettles18 (6 min. 72%), Fitzgerald08, Wiegand16]* • Inverted assignment *[Sirkia12]* • **Logical / boolean** *[Alzahrani18, Caceffo16, Ebrahimi94, Garner05, Wiegand16, Robins06]* • Incorrect calculations to support logical algorithm correctness *[Fitzgerald08]* • Missing computations *[Hall12]*	• Pre and post fix assignments *[Wiegand16]* • Random range *[Alzahrani18]* • Relational *[Wiegand16]* • Remainder operator with real operands *[Wiegand16]* • Rounding or truncation mistakes *[Hall12]* • **Type mismatch** *[Ahmadzadeh05, Garner05, Pritchard15, Seo14 (25%), Robins06]* • **Misunderstanding of operator precedence** *[Spohrer86, Teorey01, Robins06]* • Parenthesis used incorrectly *[Hall12]* • **Wrong formula** *[Simon07, Spohrer85]*

Table 3.6: The 18 condition specific errors.

Generic errors	*Specific errors*		
• Boundary case condition *[Spohrer86, Robins10, Rosbach13, Spohrer85]* • Conditionals *[Cherenkova14, Garner05, Qian17, Robins06]* • Relational operator *[Spohrer85]*	**• = vs ==** *[Alqadi17, Ebrahimi94, Hanks08, Kiran15, Raana15, Simon07, Sirkia12]* • Accidentally including sentinel values in a computation *[Simon07]* • Checking the wrong variable *[Hall12]* • Comparison *[Kurvinen16]* • Condition on rule wrong *[Winikoff14]* **• Condition variable has not been updated** *[Alqadi17, Rosbach13]*	**• Missing && and || operator *** *[Alqadi17, Altadmri15, Alzahrani18, Fitzgerald08, Simon07, Spohrer86]* • Missing condition tests *[Hall12]* • Not checked border value *[Cherenkova14]* • Numerical values are used as boolean operands *[Wiegand16]* • Perform unnecessary checking with Boolean expression *[Truong04]* • Reversed comparison operator *[Cherenkova14]*	**• Truth tables** *[Caceffo16, Garner05, Robins06]* • Unexpected cases problem. Boundary cases may not be considered [Robins10] **• Using == instead of equals() to compare strings *** *[Altadmri15 (17 min.), Brown14, Murphy08, Simon07]* • Wrong condition *[Rosbach13]* • Wrong False [Sirkia12] • Zero is excluded [Spohrer85]

Table 3.12: The 4 miscellaneous specific errors.

Generic errors	*Specific errors*
• Typo *[Garner05]*	**• Duplicate tail digit problems involve dropping the final digit from a constant with duplicated tail digits** *[Spohrer85, Spohrer86]* **• Empty statement blocks introduced with a misplaced semicolon** *[Simon07, Raana15]* **• Trivial typos - mistakes of typing (e.g. - for +) not caught by the compiler** *[Winikoff14]* • Wrong constant *[Spohrer85]*

Table 3.7: The 14 branching specific errors.

Generic errors	*Specific errors*	
• If statements *[Ebrahimi94, Garner05, Robins06]* • Jump *[Souza17]* • Selection *[Garner05, Souza17, Wiegand16, Robins06]* • Switch statements *[Alqadi17, Bryce10, Garner05, Wiegand16, Souza17, Robins06]*	• Break *[Souza17]* • Continue *[Souza17]* • Dangling else *[Teorey01]* • Failing to jump upon selection *[Sirkia12]* • Forgetting cases or steps *[Hall12]* • Identifying the output of an "if-else" statement with condition and nested "if" statements *[Wiegand16]* **• Missing "break" keywords in "switch" statement** *[Alqadi17, Raana15, Truong04, Wiegand16]*	• Wrong branch *[Sirkia12]* • Missing implication of if/else placing code outside the begin/end block *[Spohrer85]* **• Omitted "default" case in a "switch" statement** [Bryce10, Truong04] • Return *[Souza17]* • Swapping conditional block bodies in an "if" statements *[Fitzgerald08]* • Too many conditional statements *[Truong04]* **• Using multiple "if" flow structure instead of "if-else"*** *[Alzahrani18, Ettle18 (5 min. 11%), Souza17, Rosbach13]*

Table 3.8: The 27 loop specific errors.

Generic errors	*Specific errors*
• Loop contents *[Garner05, Lee99, Robins06]* • Loop errors *[Alzahrani18, Bryce10, Caceffo16, Cherenkova14, Ebrahimi94, Garner05, Hanks08, Izu16, Qian17, Robins06, Rosbach13, Simon07, Souza17, Wiegand16]* • Loop with conditionals *[Cherenkova14, Garner05]*	• Classic logic errors in searches *[Simon07]* • Code inside a loop that does not belong there *[Teorey01]* • Code that appears and is executed after the loop exits *[Wiegand16]* • Conditional into loop control variable *[Sirkia12]* • Empty loop *[Izu16]* • Erroneous incrementing of a loop counter variable (i.e., outside the loop) *[Efopoulos05]* **• For loop is not inclusive*** *[Ettles18 (2 min. 13%)]* • How and when to terminate loops *[Ebrahimi94]* **• Improper / malformed loop** *[Caceffo16, Hall12, Lee99, Murphy08, Teorey01, Wiegand16]* • Incorrect (including none) initialization of loop control variable *[Lee99]* • Incorrect update of the control variable *[Lee99]* • Indices in loops *[Kurvinen16]* **• Infinite loop** *[Bryce10, Izu16]* •Initialization of loop control variable is incorrectly placed *[Lee99]* • Loop containing "continue" statement *[Wiegand16]* • Loop has no body, extra semicolon *[Alqadi17]* **• Loop headers** *[Alzahrani18, Garner05, Robins06]* • Loop whose condition is an assignment statement or a conjunctive logical expression *[Wiegand16* • Missing input statement inside the loop, resulting in only one set of data read *[Lee99]* • Nested loop initialization, expression *[Alzahrani18]* **• Off by 1 *** *[Alqadi17, Bryce10, Cherenkova14, Ettles18 (8 min. 19%), Fitzgerald08, Izu16, Spohrer85, Teorey01, Vipindeep05]* • Stop incrementing sum *[Cherenkova14]* • Too many loop and conditional statements *[Truong04]* • Unedited loop *[Izu16]* • Unnecessary output statement within the loop *[Lee99]* • Wrong semantics of nested loops *[Izu16]*

Table 3.10: The 24 function specific errors.

Generic errors	*Specific errors*		
• Definition, data flow and header mechanics *[Garner05, Hanks08, Robins06]* • Function errors *[Qian17, Wiegand16]*	• **Always return -1*** *[Ettles18 (17 min. 12.8%)]* • **Call by reference vs call by value semantics** *[Caceffo16, Wiegand16]* • Data type of the value in the return statement is incompatible with the return type of the function *[Wiegand16]* • **Flow reaches end of non-void method** *[Altadmri15, Hristova03]* • Function name and scope *[Caceffo16]* • Incompatible types between method return and type of variable that the value is assigned to *[Altadmri15]*	• Incorrect / redundant variables or subroutines *[Grandell05]* • Initialization of formal parameters *[Caceffo16]* • Inverse nesting *[Sirkia12]* • Mismatch return type with its invocation *[Hristova03]* • Misplacing main method *[Simon07]* • Misplacing return value *[Sirkia12]* • Missing method call *[Rosbach13]* • Multiplying with a function call *[Sirkia12]* • Parameter values return value *[Qian17]*	• Parameters as local variables *[Caceffo16]* • Re-calling a function *[Sirkia12]* • **Return ignored*** *[Altadmri15 (15 min.), Brown14, Hristova03, Simon07, Sirkia12]* • Return statement is missing in the definition of a non-void function *[Wiegand16]* • **Returning 0 instead of -1*** *[Ettles18 (5 min. 22%)]* • **Unnecessary / not enough / too large method** *[Hall12, Truong04]* • **Wrong arguments** (out of order / type mismatches) *[Ahmadzadeh05, Altadmri15, Caceffo16, Hristova03, Simon07]*

Table 3.11: The 40 conceptual specific errors.

Generic errors	*Specific errors*		
• Conceptual errors *[Hall12 (58%)]* • Misunderstanding / misinterpretation *[Spohrer86, Robins10]* • Problem solving *[Bryce10, Pillay06]*	• Action definition wrong *[Winikoff14]* • Action(s) of rule wrong (but legal) [Winikoff14] • Additional (wrong) rule *[Winikoff14]* • Cognitive load problem *[Spohrer86]* • Composition problem *[Spohrer86]* • Duplicating logic *[Hall12]* **• Expectation and interpretation problem** *[Bryce10, Spohrer85, Spohrer86]* • Fault in domain knowledge *[Winikoff14]* • Fault in initial beliefs / goals *[Winikoff14]* • Improper location of the assignment expression *[Ebrahimi94]* • Incorrect / missing algorithm *[Grandell05]* • Incorrect grouping *[Rosbach13]* • Incorrect identification of the control structure needed *[Pillay06]*	• Incorrect transfer of knowledge *[Pillay06]* • Inefficient problem solving approach *[Pillay06]* • Interpretation problem [Robins10] **• Lack of conceptualization of the execution of the problem** *[Kurvinen16, Qian17, Pillay06]* • Lack of knowledge or understanding of the programming language *[Pillay06]* • Lack of understanding of control structure *[Pillay06]* • Lack of understanding of the application domain *[Pillay06]* • Malformed right place (incorrect, but in the right place) *[Cunniff86]* • Misplaced (necessary but in wrong place) *[Cunniff86]* • Missing (required but not omitted) *[Cunniff86]* • Missing action in a rule *[Winikoff14]* • Missing rule *[Winikokk14]*	• Mixed up of constructs (if and while) *[Grandell05]* • Natural-language problem *[Robins10]* • Not supported *[Spohrer86]* • Not using a compound statement when one is required *[Teorey01]* **• Optimization problem** *[Spohrer86, Robins10]* **• Plan dependency problems** *[Spohrer85, Spohrer86]* **• Previous experience** *[Spohrer86, Robins10]* • Related knowledge interference *[Spohrer86]* • Specialization problem *[Robins10]* • Spurious (not needed) *[Cunniff86]* **• Summarisation problem** *[Robins10, Spohrer86]* • Swap two variables without using a helper variable *[Kurvinen16]* • Unnecessarily complicated *[Rosbach13]*

3.3 Discussion

Table 3.1 summarizes the most common logical errors in each category. Instructors, authors, and tool developers could adapt their teaching techniques, learning content, languages, tools, and automated help systems to assist students in detecting or avoiding these common logic errors. For example, instructors could teach students to always initialize variables before students use them. Similarly, tool developers may adapt their programming tools to give warnings if variables are declared without proper initializations.

The data of the survey (in Table 3.2 to Table 3.12) helped us to identify the most difficult errors that could lead to student struggle in each of the 11 categories. The data shows that errors are more related to advanced general programming concepts such as algorithms, loops, functions, etc. The data also did not show that more errors are related to the syntax and semantics of a programming language (specific- language programming errors). Therefore, instructors might wish to focus on generic programming errors over specific programming errors when providing interventions to help students.

This survey has several limitations. One of the main limitations we faced while conducting this survey is the common lack of the study setup and research methodology in the surveyed publications. For example, many publications did not mention the year when the data were collected, the course setting (e.g., number of instructors, number of TAs, and type of learning material, type of university, CS1 or CS2, etc.), the activity setting (e.g., programming language, number/nature of activities, at-home or in- class, with or without instructor/TA help, etc.), the population setting (e.g., number of students, age, gender, ethnicity, etc.), and/or the outcome measured (e.g., struggle-causing errors using time-to-fix average, etc.). The lack of such data made it difficult to compare between the publications. Also, such a lack of details made it difficult to assess the confidence in the publications' results such as how the authors decide something is an error and when it counts as fixed.

Another limitation is the definition of CS1. We defined, early in section "Literature review method", the CS1 topics that we covered in this study, but some researchers might not agree with our definition of CS1, and thus might not agree with the coverage of the errors in this study based on that definition. For example, if there is a publication about OOP challenges in CS1, we just picked up the non-OOP related errors from that publication. Also, our error categorization did not use a systematic approach based on clearly defined guidelines, but was subjective and based on the authors' own judgments. Some researchers might not agree with such a classification. Also, the errors mentioned in this study might not apply to all programming languages and might be language-dependent or/and language-specific. Furthermore, the time-to-fix parameter, which we used in this study to highlight errors that might cause struggle, might not be accurate due to multiple factors such as students stepping out rather than working on solving the bug.

Moreover, we did a Google Scholar search for publication in the period from 1985 to 2018 (3 decades) since the search was easy to do using Google search. However, the manual database search covered only 2008 to 2018 because it was difficult to do the search manually in 8 different databases for the last 3 decades. Another limitation is the keywords that we used in the search for related publications as explained in Section “Literature review method”. We used a limited number of 11 keywords that might not be inclusive for all related publications in the last 3 decades. Also, some researchers might disagree with the keywords that we used. Lastly, we manually searched only 8 databases for related publications, so omissions of some other relevant databases might be a limitation and some researchers might disagree with having such limited database search. Even with the above limitations, we believe this survey is still helpful and more efficient than reading 47 publications.

3.4 Conclusion

We highlighted various errors that were indeed problematic and thus instructors, developers, and authors can focus on reducing those as well as the others too. As the data showed, the literature focused mainly on frequent errors but not on errors that caused struggle. For example, out of 47 literature materials, only 2 (4%) papers focused on errors that are difficult to fix. A frequent error is not necessarily problematic if easily detected and fixed by students, and in fact some would argue that such detecting/fixing is an important part of the learning process. In contrast, an error that causes struggle may lead to frustration and de-motivation without justifiable learning benefit. Detecting struggle was harder in the past due to a lack of online logging of student activity, but is more possible today with newer tools being used in CS1 classes.

4. Progression Highlighting for Programming Courses

4.1 Introduction

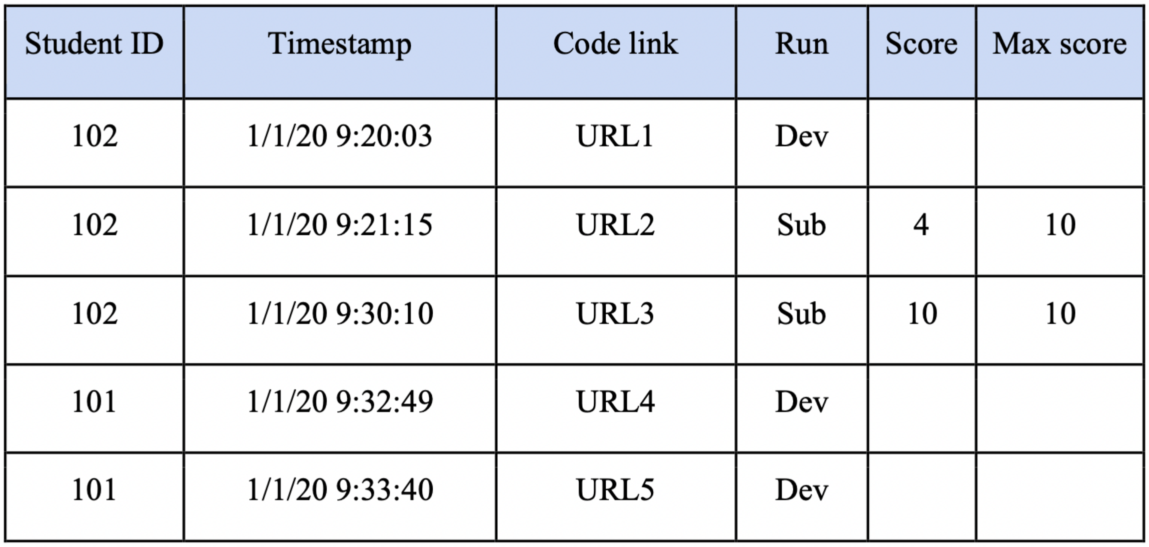
The growth in usage of program auto-graders enables new possibilities for CS instructors. Previously, most schools graded manually, while some schools used custom-built auto-graders, or made use of the freely-available Web-CAT tool [1]. However, in the past few years, several cloud-based commercial auto-graders have appeared, such as zyBooks [2], Gradescope [3], Mimir [4], Vocareum [5], CodeLab [6], and MyProgramming-Lab [7], many emphasizing ease of use and immediate score feedback to students. Based on public information from and direct discussions with those companies, we can conservatively state that at least 500 universities and at least 1,000 courses have switched from manual grading to auto-grading in recent years, impacting well over 250,000 students per year. According to a recent whitepaper from zyBooks, there is a steep rise in the use of auto-graders in recent years. For example, courses that used zyLabs (their program grader) grew from 284 in 2016 (the year zyLabs was first released) to 2,175 in 2020, the number of students from 24,216 to 132,121, the number of instructors from 364 to 2,866. They indicate 20 million submissions were graded in 2020. This rise in auto-grader usage enables instructors to spend more time engaging with students and improving the education process. For example, instructors stated that switching to the auto-grader on average saved instructors 9 hours per week [8].

These new cloud-based auto-graders not only provide immediate score feedback to students, versus auto-graders run in batch mode by instructors after all programs are submitted, but some also provide built-in development environments where students develop their code, versus using an external integrated development environment or IDE. We conducted an online survey in Spring 2021 in one of our online CS1 C++ courses where 117 students participated. Most students were positive about auto- graders. For example, 98% liked the immediate score feedback, 80% disagreed that immediate score feedback caused stress and/or confusion, and 92% preferred an auto-grader over a human grader with feedback a week later.

The cloud-based auto-grader IDE provides a new opportunity of giving instructors a view into how their students develop their code -- when students started, how long they spent, how many times a student ran their code on their own ("develop" runs), how many times they submitted for grading and what score was received ("submission" runs), and so on. Auto-graders typically grade using input/output test cases, most of which are visible to the student, so students know how their grade is being determined, and can correct their program's response.

We use the zyBooks program auto-grader. They recently began providing instructors with a detailed log of students' develop and submission runs for any programming assignment, depicted in Figure 4.1 in simplified form. Each entry in the log includes the student's identifying info, the date and time, and a link to the run's source code. For submission runs, the log also shows the score received, and the max possible score.

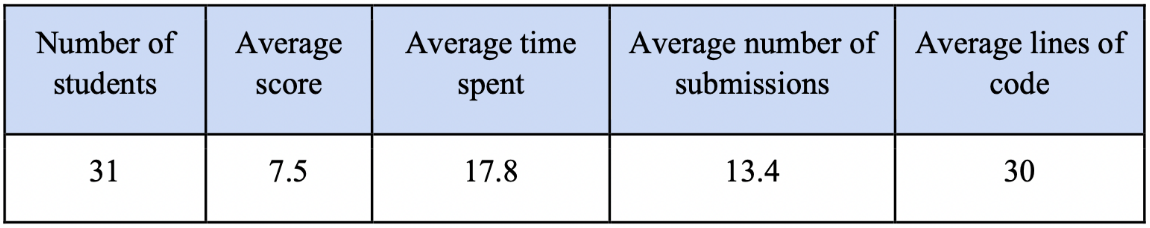
Table 4.1: Example log file from an auto-grader (abbreviated).



Student ID	Timestamp	Code link	Run	Score	Max score
102	1/1/20 9:20:03	URL1	Dev		
102	1/1/20 9:21:15	URL2	Sub	4	10
102	1/1/20 9:30:10	URL3	Sub	10	10
101	1/1/20 9:32:49	URL4	Dev		
101	1/1/20 9:33:40	URL5	Dev		

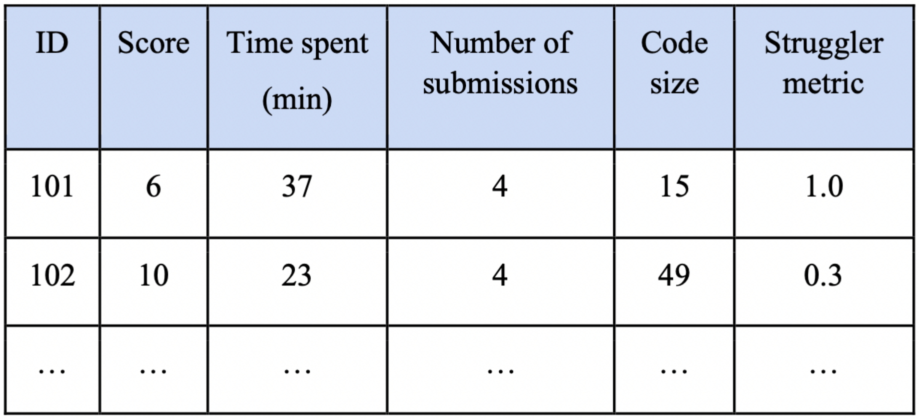
As instructors of a CS1 course that serves about 1,500 students per year, we switched from our custom auto-grader to the zyBooks auto-grader several years ago -- but we note that our approach in this chapter can be applied to any auto-grader's log files if the info is similar. We began developing scripts to process the provided log files to gain further insights into our students' behavior, such as script that estimates how long students spent on each program (and how long relative to the rest of class), counts the develop and submission runs, estimates whether detected a student is struggling (excessive submit runs or time spent), etc., as illustrated in Tables 4.2 and 4.3. Importantly, the instructor can sort by any column, such as lines of code, time spent, etc. This ability allows an instructor much flexibility in how to prioritize viewing of students. For example, an instructor might sort by time, and then examine one end of the roster to see why some students completed the program in almost no time, which is sometimes due to cheating (copy-paste of a solution obtained from a friend or online "tutor"), a student not following the requirement of submitting frequently and/or developing within the auto-grading environment, a highly- experienced student who simply got the program right quickly, or a solution that passes the test cases with much less code than the instructor expected (which could mean a problem with the test cases, or that the student found a clever simple solution).

Table 4.2: Class statistics (abbreviated).



Number of students	Average score	Average time spent	Average number of submissions	Average lines of code
31	7.5	17.8	13.4	30

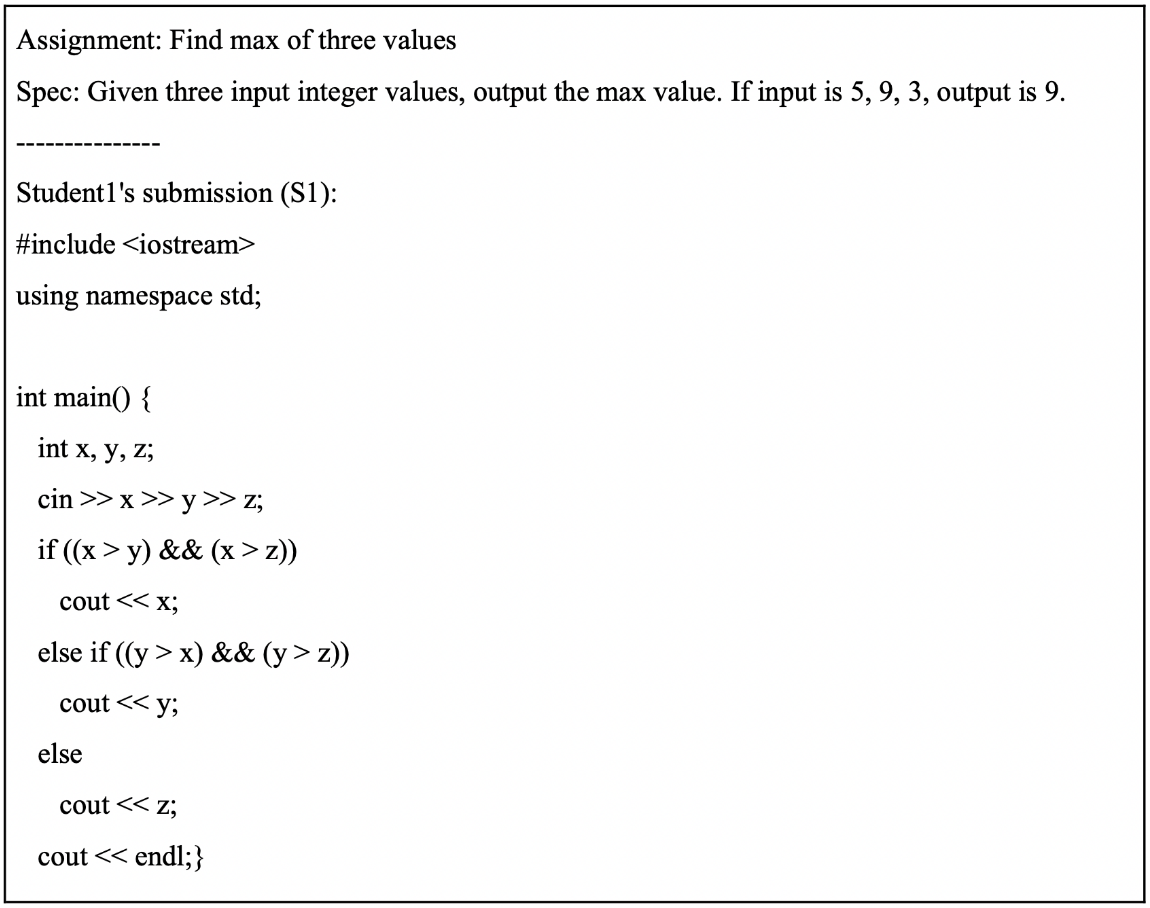
Table 4.3: Student roster view with statistics (abbreviated).



ID	Score	Time spent (min)	Number of submissions	Code size	Struggler metric
101	6	37	4	15	1.0
102	10	23	4	49	0.3
…	…	…	…	…	…

While these statistical summaries were useful and form a part of our contribution in this chapter, over time, we also desired a way to see the progression of a student's coding. The auto-grader’s web interface does provide links to all submitted code, but that was insufficient -- manually examining each code run didn't provide a quick or easy way to gain insight on what the student changed in the code.

This chapter describes a tool we developed to automatically highlight differences between each run, and to provide statistical data for those runs, forming the main part of our contribution. The current tool implementation is language independent. We plan to make the tool freely available on the web (both the roster/statistics and the progression highlighting), to serve CS educators who may wish to gain insights on their students and/or to conduct research on programming assignments. The tool takes a log file with a simple format as described above, which is the default for zyBooks (and hence immediately usable for 2,000+ courses), but any auto-grader, commercial or custom, can have their log files auto-converted to that format for importing to our tool as well.



Assignment: Find max of three values

Spec: Given three input integer values, output the max value. If input is 5, 9, 3, output is 9.

Student1's submission (S1):

```
#include <iostream>
using namespace std;

int main() {
  int x, y, z;
  cin >> x >> y >> z;
  if ((x > y) && (x > z))
    cout << x;
  else if ((y > x) && (y > z))
    cout << y;
  else
    cout << z;
  cout << endl;}
```

Figure 4.1: Example of a simple programming assignment.

4.2 Methodology

4.2.1 Code Progression Highlighting Tool

A programming assignment specifies a programming task for students. Figure 4.1 is a sample programming assignment, followed by an example of a submission from a student whose name is shown as S1; real assignments are commonly much larger.

Table 4.4 provides an example of code progression highlighting. The input is the log file, which includes all develop and submit runs.

Table 4.4: Overview of the code progression highlighting tool, to help instructors quickly see a student's changes. The change from y to x is highlighted in the second run, and from > to >= in the third run.

Code	Run type	Score	Min. since prev	Total min	Timestamp	Ins(+)
						Change(^)
						Del(-)

if ((x > y) && (y > z))
  cout << x;
else if ((y > x) && (y > z))
  cout << y;
else
  cout << z;
	Dev		0	0	1/1/20 9:20:03	

if ((x > y) && (x > z))
  cout << x;
else if ((y > x) && (y > z))
  cout << y;
else
  cout << z;
	Sub	4	0.5	0.5	1/1/20 9:21:15	^1

if ((x >= y) && (x > z))
  cout << x;
else if ((y > x) && (y > z))
  cout << y;
else
  cout << z;
	Sub	5	6.9	7.4	1/1/20 9:30:10	+1

Given the earlier mentioned roster view with statistics, if an instructor has decided to look further into a student's code, the instructor can click on a link to be taken to a page that provides highlighted code for every develop or submit run by that student, shown in Table 4.4.

The progression highlighting tool highlights any code that was added, changed or deleted from the previous run, so that instructors can quickly see the changes made.

For each code run, the tool also provides statistics, as shown in Table 4.4. The statistics include type of run (sub or dev), the number of characters added, changed, and deleted; the time spent between this run and the previous run, elapsed time (time spent so far since the first run), timestamp for each run, score for each run. If the time exceeds ten minutes between runs, we assume the student stepped away, and ignore that time.

We note that highlighting, and some statistics, are available from program version control systems like Git [9], Subversion [10], CVS [11], and Mercurial [12]. However, our approach differs in several ways. First, we don't require the use of such version control tools; students simply develop and submit using their class auto-grader. This simplicity can be especially important in early CS classes like CS1 or CS2. (Version control may be taught later in a class or course sequence). Second, our statistics are specifically intended for instructors, so include items not found in most version control systems. Third, we estimate time spent, which is lacking in version control systems.

The code progression highlighting tool is a web tool written in Python 3 using the Common Gateway Interface (CGI) specifications [13]. The tool is about 5K lines of code. The tool consists mainly of two parts: the first part is the student roster builder and the second part is the code progression highlighter. Moreover, each part consists of multiple subroutines and dictionary data structures. For example, for the student roster builder, there is a subroutine to read the log file and build a dictionary that contains the students (indexed by the students' IDs) with their codes and their codes' meta-data (such as timestamp, score, type of run, etc.). Another dictionary builds a class statistic, and another one for top quartile students. Likewise, the code progression part contains multiple subroutines and dictionaries. For example, one subroutine finds the difference between consequent codes and highlights them in yellow.

4.3 Discussion

4.3.1 Applications

The progression highlighter can be applied by instructors in many ways, some of which we have begun doing in our own classes. Note: The progression highlighter and in fact any new analysis tool works best if students are required to do all development in the auto-grader's environment. In fact, for our CS1, we long ago started using the auto-grader's built-in development environment for most programs, thus avoiding issues with new students installing/learning external IDEs, and we are aware of hundreds of schools that do the same using that same auto-grader. However, for courses that want students using an external IDE, instructors can simply require frequent submissions, such as every 20 minutes or every 20 lines of code (for example). Again, we have found many courses already have such requirements, to encourage students to start early, to develop incrementally, and to decrease the likelihood students will just copy-paste someone else's solution.

Table 4.5: One example of the 5 selected programming assignments.

Problem statement	Solution

Numerous engineering and scientific applications require finding solutions to a set of equations. Ex: 8x + 7y = 38 and 3x - 5y = -1 have a solution x = 3, y = 2.
 	
Given integer coefficients of two linear equations with variables x and y, use brute force to find an integer solution for x and y in the range -10 to 10.
	int main() {
	  int a1, b1, c1;
	  int a2, b2, c2;
	  int x, y;
	  bool eqn1Solved, eqn2Solved;
	  bool solutionFound = false;
	  int xSolution, ySolution;
	  cin >> a1 >> b1 >> c1;
	  cin >> a2 >> b2 >> cin >> c2;
	  for (x = -10; x <= 10; ++x)
	    for (y = -10; y <= 10; ++y)
	      eqn1Solved = ((a1*x + b1*y) == c1);
	      eqn2Solved = ((a2*x + b2*y) == c2);
	      if (eqn1Solved && eqn2Solved) {
	        solutionFound = true;
	        xSolution = x;
	        ySolution = y;
	      }
	  if (solutionFound)
	    cout << xSolution << " " << ySolution << endl;
	  else
	    cout << "No solution" << endl;
	}

One application is to bring up a student's highlighted progression for a student who has come to office hours in need of help. The progression gives the instructor a powerful view of the student's effort history. Did the student start early or late? Are they developing incrementally or writing large pieces of code all at once? Are they testing their created functions first or just using them untested in their larger code? When faced with an erroneous program, are they properly debugging or just making random changes to see what happens, or are they focused on the wrong region of code? We have found the progression highlighter exceptionally useful for such purposes, providing insights that we could not see before.

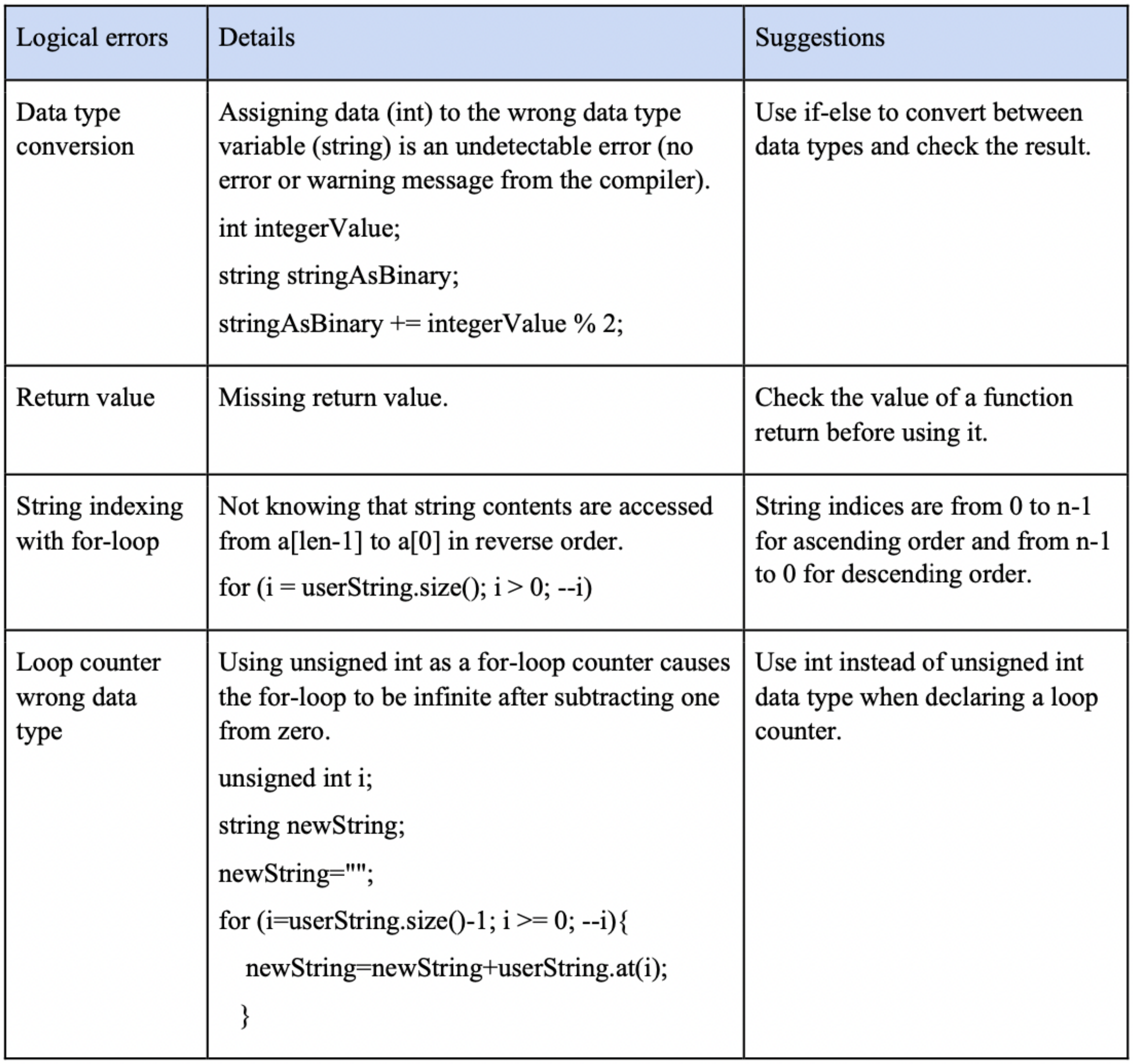
Another application is cheating detection, or even better cheating reduction/prevention. An instructor can sort students by number of runs or time spent, and investigate those students who spent the least time or runs, to see if they simply copy-pasted a solution. With the develop/submission requirement above, and by showing students the progression highlighter tool in class (perhaps in a fun manner during class like seeing how students did on a low-stakes programming task, and/or showing the tool with student names hidden), students may be less tempted to simply copy-paste a solution – students regularly observe that instructors can and do look at their progression, and thus simply pasting a solution will be a smoking gun. For in-class use, we include an “anonymize” button that hides student names/email from the display. Furthermore, if an instructor by other means suspects a student of plagiarizing code (perhaps via similarity detection from a tool like Moss [14], or by dramatically higher programming assignment scores than exam scores), the instructor can examine that student's highlighted progression for further confirmation that the student did not develop the code. Note that statistics alone, like number of runs or time spent, can be more easily faked (via bogus runs before eventually copy-pasting a solution). But faking a code *progression*, though still possible, begins to approach the skill of just writing the code itself.

Many other applications exist, such as quickly gaining insight on what errors are causing a class to struggle, as we report on next.

4.3.2 Experiences Using the Tool to Find Causes of Struggle

The progression highlighter helped us for a particular application. We sought to determine what coding mistakes caused students to struggle in our CS1 class. Previous research described common mistakes among learners [15, 16], but just because a mistake is common does not make the mistake bad; much learning comes from making and then fixing mistakes. In contrast, some mistakes cause students to struggle, which means the student cannot find their error, resulting in numerous runs, excessive time spent, and frustration. Frustrated students are more likely to resort to cheating, or to give up on the assignment, which may eventually lead to failing grades or dropping the class as well.

Table 4.6: Errors by strugglers in the programming assignment “Convert to binary - functions”.



Logical errors	Details	Suggestions

Data type conversion	Assigning data (int) to the wrong data type variable (string) is an undetectable error (no error or warning message from the compiler).
	int integerValue;
	string stringAsBinary;
	stringAsBinary += integerValue % 2;
		Use if-else to convert between data types and check the result.

Return value	Missing return value.	Check the value of a function return before using it.
String indexing with for-loop	Not knowing that string contents are accessed from a[len-1] to a[0] in reverse order. for (i = userString.size(); i > 0; --i)	String indices are from 0 to n-1 for ascending order and from n-1 to 0 for descending order.

Loop counter wrong data type	Using unsigned int as a for-loop counter causes the for-loop to be infinite after subtracting one from zero.
	unsigned int i;
	string newString;
	newString="";
	for (i=userString.size()-1; i >= 0; --i){
	newString=newString+userString.at(i);
	}
		Use int instead of unsigned int data type when declaring a loop counter.

For each of 5 selected programming assignments that we assigned in Spring 2017, we uploaded the auto-grader’s log file into our tool. One example is shown in Table 4.5. Next, we sorted by time, which is one measure of struggle. (Number of runs is another). For the students who spent more than 30 minutes, we clicked on the link to be taken to the progression highlighter, and then scrolled through to quickly and easily see where the student was spending their effort, and could see the error they were making as shown in

Table 4.6. We discovered that the following errors were common logical errors among students that cause struggle (the main list is the programming assignment while the sub list are the errors per a program assignment):

- Brute force equation solver
    - Typos in formula
    - Convert to binary - functions
    - Data type conversion
    - Return value
    - String indexing
    - Loop counter
- Palindrome
    - Lack of plan
    - String indexing
- Count characters
    - Lack of plan
    - Using cin() instead of getline() to read a phrase
    - String indexing
- Leap year - functions
    - If vs, if-else

Table 4.7: Strugglers in the five programming assignments. A struggler is a student who spends 30 min. or more in a programming assignment.

Program Assignment	strugglers / students	Total runs by strugglers	Avg. score out of 10	No. Errors	Teacher manual inspect time (min)
Brute force equation solver	3 / 40	199	8	1	12
Convert to binary - functions	11 / 56	414	6	4	50
Palindrome	7 / 49	476	7	2	38
Count characters	17 / 74	790	8	5	47
Leap year - functions	3 / 71	96	9	1	04

The summary of our findings is shown in Table 4.7. The time spent column on the right shows that we only needed tens of minutes per program to find common errors. In contrast, in the previous summer we did a similar analysis, but that required several weeks, since we had to manually examine code without the benefit of the progression highlighter, which not only took much longer but was also quite tedious and tiring.

4.4 Conclusion

This paper introduces an enabling technology for instructors of programming classes that emphasizes viewing a student's *progression* through the program development process. The technology is a tool that takes as input a log file from a program auto-grader used by thousands of classes (but could be used with other tools too if their logs were converted to the format). The tool provides instructors with a unique ability to sort their class roster according to various statistics like time spent, code size, or number of runs. Then, importantly, the tool allows instructors to view any student's highlighted progression through the program development process, seeing each run's code, with highlights showing additions/changes from the previous run, plus statistics for that code like time spent, number of additions/deletions, and score. The technology enables various new capabilities for instructors. For example, an instructor can open a student's highlighted progression for a student in office hours, to help better understand the student's approach to a program and to provide advice. Or, an instructor can sort by number of runs or time spent to detect students who might be struggling, choosing students with low scores at the top of that list, and examining their progressions to see what errors prevented them from scoring well, which we demonstrated in this paper can be done in just tens of minutes, yielding powerful insights that can improve an instructor's teaching. Or, an instructor can detect potential cheating by detecting students who got full credit with little effort (few runs or little time) -- or even better, to reduce/prevent cheating by showing students the tool (ideally in a fun way, perhaps on low-stakes programming tasks) so students realize their programming process is being viewed, and not just their final submission. Dozens of possibilities exist, and we look forward to seeing how instructors use this technology to improve their classes and to conduct future research. Future work may include providing support for sets of labs (like all labs for a week), supporting team projects, and more.

5. Detecting Possible Cheating In Programming Courses Using Drastic Code Change

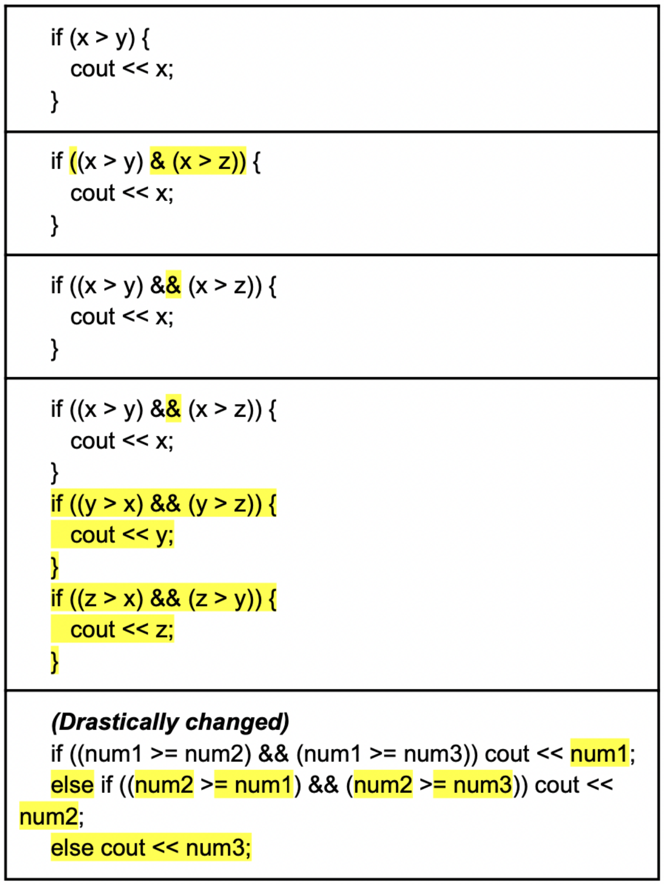
5.1 Introduction

A common form of cheating on programming assignments involves a student initially trying, then struggling, and eventually giving up and copying a solution from elsewhere. Regarding such cheating, Malan [1], who teaches Harvard's CS50, notes "All too often were students' acts the result of late-night panic". Table 5.1 provides an example from real code developed by a student at our university, recorded by the cloud platform we require students to use, with changes between successive code pairs highlighted. The student was required to write code that finds the maximum of three input values. The first four code snippets show the student's code being developed along a normal progression. But, the student eventually gets stuck, having failed to consider cases where input values are equal. Giving up, the last snippet shows code the student copied from an online source (which we found).

Another common form of cheating involves students not even trying to write the program themselves, but going straight to online solutions or contractors.

Many programming courses, especially classes with large numbers of students, use automated code similarity detection tools such as Moss [2] to help detect potential cheating. Such tools focus on detecting similar submissions from multiple students. However, for the above situation, similarity detection fails to detect cases where a student submits a copied solution that is unique, such as code obtained from today's popular online tutors who provide solutions for just a few dollars and often within just tens of minutes [3].

Table 5.1: The first four code snippets follow a normal progression, but the fifth code snippet labeled (Drastically changed) shows a drastic change, involving the arbitrary change in program identifiers, the switch from brace use to no braces, and a different algorithm.



```
if (x > y) {
   cout << x;
}
```

```
if ((x > y) & (x > z)) {
   cout << x;
}
```

```
if ((x > y) && (x > z)) {
   cout << x;
}
```

```
if ((x > y) && (x > z)) {
   cout << x;
}
if ((y > x) && (y > z)) {
   cout << y;
}
if ((z > x) && (z > y)) {
   cout << z;
}
```

(Drastically changed)

```
if ((num1 >= num2) && (num1 >= num3)) cout << num1;
else if ((num2 >= num1) && (num2 >= num3)) cout <<
num2;
else cout << num3;
```

Fortunately, today a different type of automation is possible due to the trend of instructors having access to the history of a student's code. For example, some instructors require students to frequently commit code to GitHub [4]. Thousands of courses and hundreds of thousands of students now use auto-graders [5], and many such auto-graders record the code every time the code is submitted for auto-grading [6]. Some instructors even have the students develop code in IDEs that auto-save the code at regular intervals, or that save the code every time the student runs the code [7].

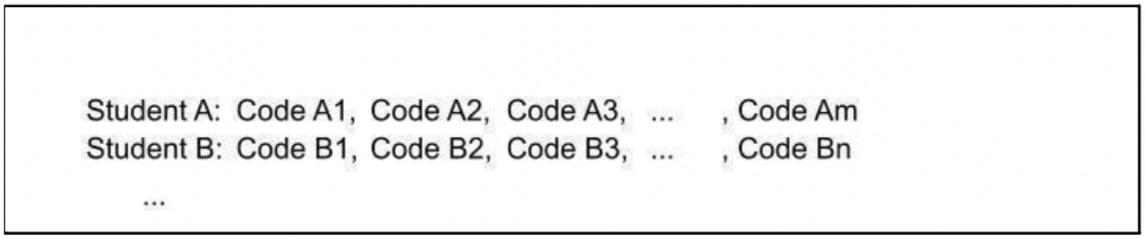
Thus, to detect some forms of cheating, a new approach complementary to similarity checking is possible today. The approach looks across the progression of a student's saved code, trying to detect the case where a student was submitting a normal code progression, but then (perhaps in "a late night panic") suddenly submits very different code -- what we call a "drastic change". As with similarity checking tools, a drastic change detector does not label anything as cheating, but rather simply points instructors to cases that the instructor (or TAs) may wish to investigate. Thus, a key challenge is to build a highly-accurate drastic change detector, to avoid wasting an instructor's time with false positives, and to avoid missing drastic changes as well (false negatives).

This chapter presents a tool we have developed, and used in our courses, for examining code progressions to detect a "drastic" change suggestive of cheating. Our goal is to make this tool freely available to all programming instructors, akin to Moss' availability. A key challenge is to eliminate false positives, because students often make substantial (but not drastic) changes to code during a normal development process. The paper provides experiments on both synthetic and real code progressions, showing the algorithm yields good accuracy. The paper also presents an experiment on the prevalence of drastic change in real student code.

5.2 Methodology

5.2.1 Problem Statement

The input to our tool is a source code progression for each student on a programming assignment, as in Figure 5.1. Student A has m code runs and Student B has n code runs. Code A1 is a link to the entire source code for Student A's first code run.



Student A: Code A1, Code A2, Code A3, ... , Code Am
Student B: Code B1, Code B2, Code B3, ... , Code Bn
...

Figure 5.1: Code progressions for two students on a programming assignment. For simplicity, we sometimes refer to each code instance as a "run".

We get the code progressions from zyLabs offered by zyBooks [8], which records source code for every program submitted for auto-grading, and if the instructor requires the use of the built-in IDE, also records the source code every time a student runs their program during development. zyLabs allows an instructor to click a button to download a log file (a lab file) of all student code progressions for any programming assignment. Our approach also applies to any other tool or scenario where an instructor can access student code progressions, such as when using other commercial or custom auto-graders and/or

IDEs that make available a history of student code, or when requiring students to commit code regularly to GitHub.

In our problem definition, the approach examines every pair of sequential code instances in the progression, comparing the pair's first code (Code1) with the pair's second code (Code2). In Figure 5.1, the approach compares Code A1 and Code A2, then Code A2 and Code A3, etc. (In the future, we may consider a broader definition that at once considers a larger region of the code progression).

For any pair (Code1, Code2), we assign a "drastic change" value between 0 and 1, where 0 means almost no change, 1 means a very drastic change, and a value 0.5 or above means an instructor may wish to examine the code for possible cheating. The approach never concludes cheating itself, but rather, like Moss, directs instructors to suspicious cases. A "drastic change" is informally defined as a suspiciously large change that an instructor may want to inspect for possible cheating. Not all large changes are suspicious, so the key here is to try to match what instructors would consider possible cheating. We also classify drastic changes into two categories: initial leaps and gave ups. An initial leap occurs when the first code run size (lines of code) is bigger than 50% of the first highest-score code run (the first code run with a full score) size in a student code progression. We skip the initial leap detection if the first highest-score code run size is less than 15 lines of code or if a student does not have a code run with a full score. A gave up is a drastic change anywhere in the code progression except in the first code run. We also have solution hopping as a sub-category for each drastic change category. Solution hopping is a drastic change in addition to an initial leap or a gave up. Our tool detects all these categories as a “drastic change”.

5.2.2 Text Comparison Based Solution

Experienced programmers can readily notice drastic changes in code. One indicator might be a sudden and arbitrary change in program identifiers, such as changing x, y, z to num1, num2, num3 in Table 5.1. Another is changing code style, such as changing from consistent brace use to no braces in Table 5.1. Another is using a different algorithm, as in switching from a multiple-if structure to an if-else structure in Table 5.1. Any one such change may not be suspicious, but multiple such changes between two runs increases suspicion.

We wish to detect drastic changes automatically. Perhaps the most obvious approach to automatically detecting drastic change (for gave ups and solution hoppings) is to just use a "diff" tool on each code pair in a student's progression. Diff is a text comparison tool that finds added, changed, and detected characters between files, and is a built-in utility in most operating systems [9]. We used the diff library that is part of the Python language [10]. We noticed (from synthetic data analysis, explained later in the experiments section) that drastic change mostly occurs when 1 - diff’s returned value (the similarity ratio or simRatio for short) is less than or equal to a threshold of about 0.8. Therefore, we considered any code progression with a simRatio less than or equal to the threshold as a drastic change, and return 1 - simRatio as that drastic change value. For simRatio above 0.8, we set the drastic change value to 0, meaning the code pair is quite similar. Our tool is orthogonal to the used programming language in programming assignments, and supports any programming language.

5.2.3 Experiments

We ran two experiments to answer two research questions: (1) What is the accuracy (in terms of sensitivity and specificity) of our tool in detecting drastic changes?, and (2) What is the prevalence of drastic changes (in terms of initial leaps, gave ups, and solution hoppings percentages)?. We used a quantitative approach, probability sampling, and statistical analysis (sensitivity/specificity for the first experiment, and percentage for the second experiment) to conduct both experiments, but used experimental design for the first experiment, and descriptive design for the second experiment.

In the first experiment to test the efficacy of our tool in detecting drastic changes (Experiment 1, the tool's efficacy experiment), we carried out the experiment on synthetic progressions, where we injected drastic changes into real progressions, and on unmodified real progressions where we manually discovered actual drastic changes in student code.

In the second experiment to find the prevalence of drastic changes (Experiment 2, the prevalence of drastic changes experiment), we carried out the experiment on real data.

In each experiment, we used 2 zyBook lab files, one lab file for a 240-student C++ CS1 section (176-line instructor solution) and and the other lab file for a 430-student Python CS1 section (121-line instructor solution) in Fall 2019 at two large research universities.

In the tool's efficacy experiment, for a chosen lab, we sorted the student roster in descending order by their student ID and picked 20 students who received a full score (10 points) and had no drastic change in their code progressions after examining their progressions manually.

We randomly divided the 20 students into two groups of 10 students, Group 1 and Group 2. For each student in Group 1, we replaced the first and the middle code-runs of one student with the first full-score-code-run from the corresponding student in Group 2. The reason for doing so is to introduce drastic change as an initial leap and as a gave up for each student in Group 1.

For each student in Group 1, we manually examined each code pair in their progression for drastic changes, and manually recorded the results. The results were 10 students with drastic change. Then, we ran our drastic change detection tool on both groups to detect drastic changes with a drastic change value of 0.5 and above. The tool detected 10 students with drastic change in Group 1 and 10 students with no drastic change in Group 2. This concludes the experiment using synthetic progressions.

Again, in the tool's efficacy experiment, for a chosen lab, we sorted the student roster in descending order by their student ID and picked 30 students who received a full score (10 points) where 10 students had drastic change, 10 students had no drastic change, and 10 students had big change (we define big change as having more than 20 lines of code difference between 2 consecutive code runs) after examining their progressions manually.

The reason for doing so is to have a balanced sample of students where 10 students had possibly drastic change, 10 students had possibly no drastic change, and 10 students with big change to see how accurately our tool can discern drastic change from big change.

For each student, we manually examined each code pair in their progression for drastic changes, and manually recorded the results. The results were 20 students with drastic change and 10 students with no drastic change. Then, we ran our drastic change detection tool to detect drastic changes with a drastic change value of 0.5 and above. The tool detected 20 students with drastic change and 10 students with no drastic change. This concludes the experiment using real progressions.

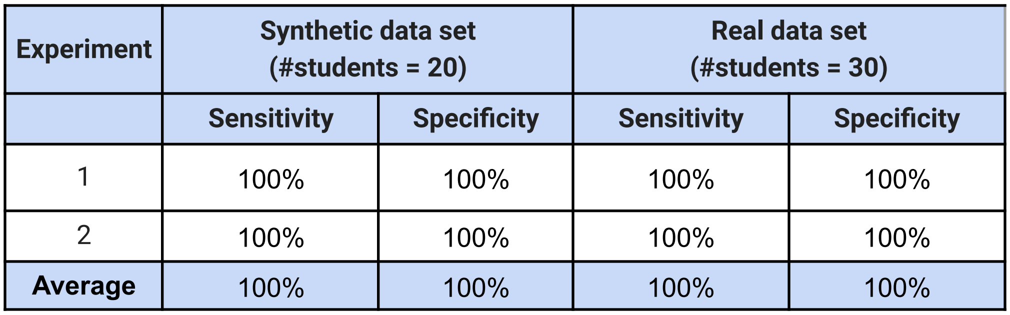
We conducted the tool's efficacy experiment twice (Experiment 1 and Experiment 2). For Experiment 1, we used 30 students from the C++ lab file. For Experiment 2, we used 30 students from the Python lab file. For Experiment 2, the results for synthetic data manual and tool analysis were 10 students with drastic change and 10 students with no drastic change. For the real data, the manual and tool analysis results were 15 students with drastic change and 15 students with no drastic change.

Tables 5.2 and 5.3 present the drastic changes (initial leaps and gave ups) accuracy using sensitivity and specificity for each of the 2 experiments. For both experiments, the tool averaged 100% sensitivity, and 100% specificity.

In the prevalence of drastic changes experiments, we used the same 2 lab files as they are without changing any data, so the experiments used real code progressions. However, in this experiment, we did not limit the experiments to only 30 students per lab, but included all students for a total of 670 students for the two labs. Table 5.4 summarizes the prevalence of drastic change experiments in real student data for Experiment 1 (using the C++ lab file) and Experiment 2 (using the Python lab file).

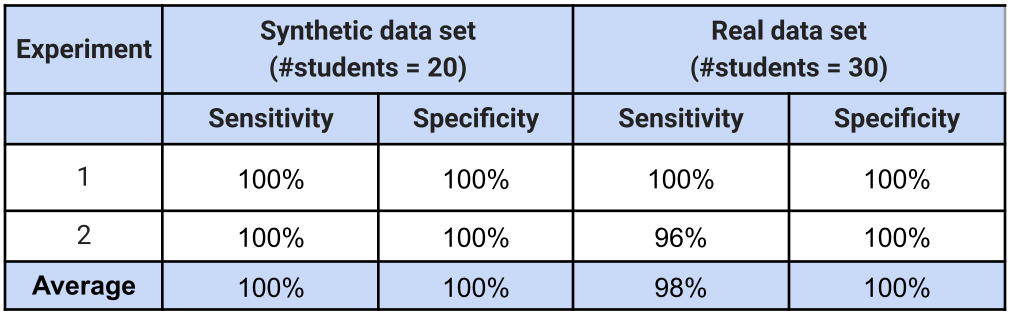
We believe the high average ratio of initial leaps of 32% might not be all related to possible cheating, and some might be rather related to students developing their code in their favorite IDEs then copying their code to the zyBook platform, hence triggering a high average ratio of initial leaps. Instructors can award students points for developing their code completely inside the cloud platform IDE to reduce the chance of such inaccuracy.

Table 5.2: The tool’s efficacy experiments for initial leaps.



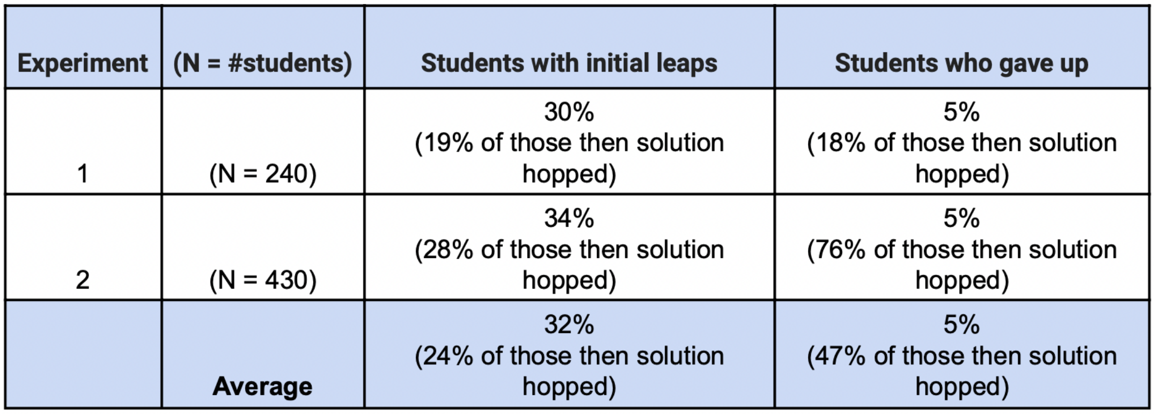
Experiment	**Synthetic data set (#students = 20)**		**Real data set (#students = 30)**	
	Sensitivity	**Specificity**	**Sensitivity**	**Specificity**
1	100%	100%	100%	100%
2	100%	100%	100%	100%
Average	100%	100%	100%	100%

Table 5.3: The tool’s efficacy experiments for gave ups.



Experiment	Synthetic data set (#students = 20)		Real data set (#students = 30)	
	Sensitivity	Specificity	Sensitivity	Specificity
1	100%	100%	100%	100%
2	100%	100%	96%	100%
Average	100%	100%	98%	100%

Table 5.4: Prevalence of drastic changes (initial leaps, gave ups, solution hoppings) in real student code.



Experiment	(N = #students)	Students with initial leaps	Students who gave up
1	(N = 240)	30% (19% of those then solution hopped)	5% (18% of those then solution hopped)
2	(N = 430)	34% (28% of those then solution hopped)	5% (76% of those then solution hopped)
	Average	32% (24% of those then solution hopped)	5% (47% of those then solution hopped)

5.3 Discussion

Recently, we considered whether the diff that comes with the version control system Git might be more appropriate for drastic change detection than the Python diff. According to Nugroho [12], Git uses Myer’s algorithm to produce its diff. We tested Google's implementation of Myer’s algorithm [13]. We noticed similar results when using Myer’s algorithm for most cases (except for code improvement and code relocating).

Switching to Myer's algorithm may be considered in future work. Also, code templates could alter our results. A code template is prewritten code (predefined variables, function layout, etc.) perhaps provided by the instructor, which students might add to their code, which could increase the code size if added yielding a high drastic change value. We used a simple method to detect templates by automatically noting the minimum common code across the top 10% of students who received full scores and had the smallest first code runs. However, other methods might be more accurate. Future work may include allowing template code to be provided as input to the tool. Moreover, our tool spends on average 15 minutes for each lab in the experiment, over 95% of which is spent just downloading code runs from the zyBooks server. In the future, we plan to investigate ways of detecting drastic change without having to download all code runs, by downloading a select subset. Also, the tool only examines two adjacent code runs at a time for simplicity, rather than examining the entire sequence of code runs, which could provide more insight into drastic change.

5.4 Conclusion

Commonly-used automated code similarity detection tools are not effective in detecting possible cheating in unique solutions. We introduced a tool to supplement similarity detectors to detect possible cheating using drastic changes in student code progression. The experiments showed that using diff is accurate for drastic change detection. On average, the tool achieved 100% sensitivity and 100% specificity. We also ran an experiment to detect drastic changes (initial leaps, gave ups, and solution hoppings) in real student code. The experiment revealed that 32% of drastic changes are initial leaps (24% of those solution hopped) and 5% are gave ups (47% of those solution hopped). We plan to make the tool available for the CS community as a free web tool.

6. Detecting Possible Cheating in Programming Courses Using Code Style Variation

6.1 Introduction

We found that a student cheating in a programming course often gets solutions from different sources for different labs, causing one student's coding style to vary unusually across labs. Previous research confirmed that coding style is a personal preference that usually persists during coding and contributes most to author attribution [1]. We developed an approach that exploits such findings by detecting variation in coding style across labs for the same student as a potential indicator of cheating. Figure 6.1 illustrates that the style variation approach is complementary to traditional code similarity: A code similarity tool can still be run for one lab across students, while a code style variation tool can be run for one student across labs.

Our approach does not label anything as “cheating”, but rather simply points instructors to suspicious cases that the instructor may wish to investigate.

There are multiple coding style choices that students make in their labs. For example, with respect to brace usage, two of the most common styles for the C++

language are Kernighan and Ritchie (K&R) style and Allman style, shown in Table 6.1. In K&R style, the opening brace is on the same line as its statement, while in Allman style, the brace is on the next line. Our approach is orthogonal to coding style standards, not requiring any particular style, but rather detects variation in style across different labs.

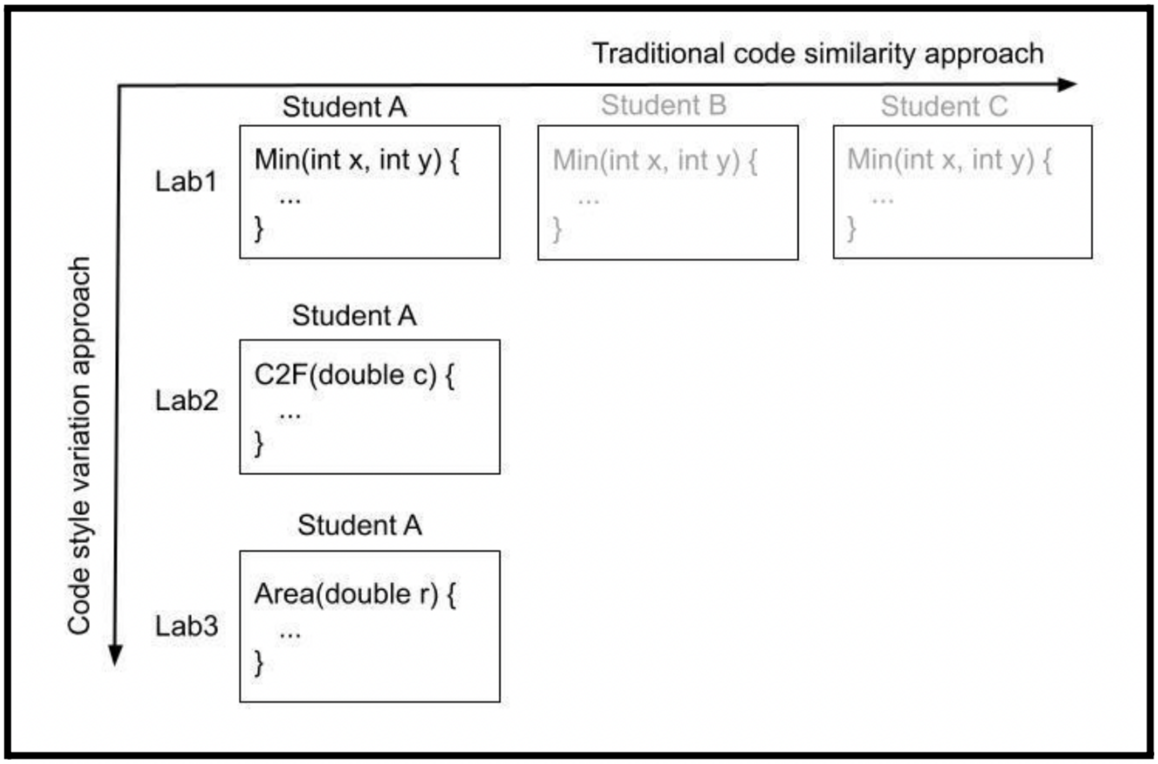
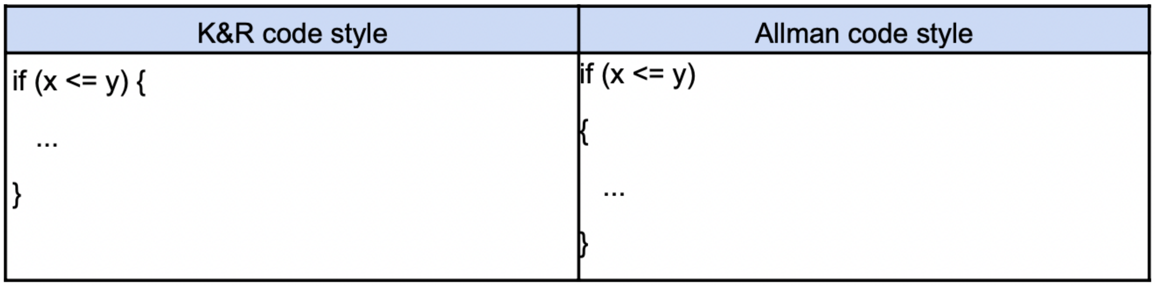


Figure 6.1: The traditional code similarity approach compares one lab across students (horizontal axis). Our code style variation approach compares one student across labs. The approaches are complementary, each pointing instructors to potential cheating.

Table 6.1: Code style feature example: Brace style variation.



K&R code style	Allman code style

if (x <= y) {
...
}
	if (x <= y)
	{
	...
	}

We developed a tool to implement a code style variation detection approach. We plan to make the tool a free web tool, akin to Moss's availability, available to programming instructors. The paper provides experiments showing the tool yields good accuracy for labs in three popular programming languages (C++, Java, and Python).

6.2 Methodology

6.2.1 Code Style Variation

To detect style variation, we first defined common style features that may vary across programming learners. Table 6.2 shows 15 style features that we used for C++ code. For Java code, we used similar features to C++ except that for the String access feature (#2 in Table 6.2) Style1, we used index.at() instead of .at(). Table 6.3 shows 15 code style features that we used for Python code. The figures show the style features with their weights (in parentheses) that we applied to each feature. The figures also present summaries for each feature, and common style variation (Style1 and Style2) that we considered, with examples for each style variation. The explanation column shows the mapping functions (unanimity, majority, single instance, or threshold) that we used to map a feature to its style variation (Style1 or Style2) for given code. We chose the code style features, their style variations, weights, mapping functions, and thresholds based on our observations and experience. The unanimity mapping function requires that all instances of a feature be in one style (Style1 or Style2), and zero instances of the feature in the other corresponding style. For example, the formatted comment feature (#4 in Table 6.2) uses the unanimity mapping function. So, If all comments are formatted, then the Style1 is used, but if at least one comment is not formatted, then the Style2 is used. The majority mapping function requires that instances of a feature in one style are the most common in code, even half of that style's instances is greater than the instances of the opposite style, otherwise we consider the feature is not applicable, and ignore it. The open brace feature (#1 in Table 6.2) has the style variation Style1 if most open braces are inline, and at the end of the line, else has Style2 if most open braces are not inline.

For example, if a student code has 10 inline open braces and 3 standalone open braces, then the majority function applies. This is because half of the inline open brace instances, which is 5, is greater than the standalone open brace instances, which is 3. Therefore, the student code conforms to Style1. We use the majority mapping function instead of just greater-than mapping function to address the ambiguity when there is no clear distinction between which style variation is the common coding style. For example, if a student code has 10 inline open braces, and 9 standalone open braces, using just greater-than would conform to Style1, which does not seem a common coding style in this example. This is because both inline open braces (10 out of 19) and non-inline open braces (9 out of 19) seem to be common coding styles in this student code, so we decided none of them is common in this case.

The single instance mapping function requires just one instance in one of the feature styles (Style1 or Style2) to occur to conform to a style. For example, the inline comment feature (#5 in Table 6.2) has the style variation Style1 if code has at least one inline comment, but Style2 if code does not use any inline comment.

The threshold mapping function requires instances in one of the feature styles (Style1 or Style2) to pass a threshold to conform to a style. For example, the code left aligned feature (#14 in Table 6.2) requires that more than 10% of lines of code for all code inside code blocks are left-aligned. If such a threshold is passed, then code conforms to Style2.

Not all style features are equally important in detecting code style variation. Thus, we also assign a weight for each style feature to indicate its importance in identifying code style variation. For example, in Table 6.2, open brace style has a weight of 0.5, while initialized variable declaration has a weight of 0.25, because we deemed some variation in the latter to be normal.

6.2.2 Tool

The input to our tool is a log file that contains all lab submissions for all students across multiple labs. Such a log file is available to instructors using the zyBooks program auto-grader zyLabs [2], which is how we obtained those log files. However, similar log files may be available from other commercial or university-built auto-grader or submission systems, as well as by scripts that download Github code in courses that use Github. The output of our tool is a roster of students, where each student has a style variation score, along with a link to their style variation matrix (or simply a student matrix). A simplified version of a roster is shown in Table 6.5.

Table 6.2: C++ code style features with their weight (in parentheses), explanation (including the mapping function such as unanimity, majority, single instance or threshold to map a feature to its style variation), and style variation (Style1 and Style2) that we used in our approach to detect code style variation in C++ labs.

No.	Feature	Explanation / Mapping (M)	Style1	Style2

1	Open brace (0.5)	Using inline open braces vs not.
		M: Majority.
			Most open braces are inline and at the end of the line.
			Example:
			if (...) {
				Most open braces are not inline.
				Example:
				if (...)
				{

2	String access (0.5)	Using at() in string indexing vs [ ]. M: Majority.	Most string indexing uses at(). Example: word.at(0);	Most string indexing uses [ ]. Example: word[0];
3	Comment (0.5)	Using comments vs not. M: Single instance.	At least one comment is used. Example: /* Compute max*/	No comment is used. Example: N/A
4	Formatted comment (0.5)	A formatted comment starts with a capital letter and ends with a period. Using formatted comments vs not. M: Unanimity.	All comments are formatted. Example: /*Compute max.*/	At least one comment is not formatted. Example: /*compute max*/
5	Inline comment (0.5)	An inline comment is a comment located on the same line with a line of code. Using inline comments vs not. M: Single instance.	At least one inline comment is used. Example: if (...) …//Compute max	No inline comment is used. Example: if (...) …

6	Commented functions/methods (0.5)	Using comments (before or after) all function/method headers vs not.
		M: Unanimity.
			All functions/methods have comments.
			Example:
			// This function gets min
			int min(...) {
				At least one function/method lacks comments.
				Example:
				int min(...) {

7	Initialized variable declaration (0.25)	Not using initialized variable declaration vs using at least one. M: Single instance.	No initialized variable declaration is used. Example: int x;	At least one initialized variable declaration is used. Example: int x = 0;

14	Code left aligned (0.25)	A code block is code located between two curly braces { and }.
		Using code with less than 10% of its code blocks left-aligned vs more than 10% of its code blocks left-aligned.
		M: Threshold of 10%.
			Less than 10% of code is left-aligned above.
			Example:
			…
			if (...) {
			  x = 1;
			  y = 2;
				10% or more of code is left-aligned above.
				Example:
				if (...) {
				x = 1;
				y = 2;

15	Empty lines (0.5)	Using empty lines in code blocks vs not.
		M: Single instance.
			At least one empty line is used in a code block.
			Example:
			{
			  int x;
			
			  int y;
			}
				No empty line is used in a code block.
				Example:
				{
				  int x;
				  int y;
				}

Table 6.4 shows a student matrix for a student with 3 C++ labs: Lab1, Lab2, and Lab3 in the first column. The second column shows the student code for the 3 labs. The next 3 columns show 3 C++ code style features (with their weights in parentheses): open brace, multi-statement per line, and initialized variable declaration. Only 3 out of the 15 C++ code style features are shown for brevity.

For the open brace feature, a lab can have Style1 (S1) if the majority of open braces are inline and at the end of the line, or Style2 (S2) if the majority of open braces are not inline and not at the end of the line. For the multi-statement per line feature, a lab can have S1 style for no multi-statement per line is used, or S2 for at least one multi-statement per line is used. For the initialized variable declaration, a lab can have S1 for no initialized variable declaration is used, or S2 for at least one initialized variable declaration is used.

The first lab has style S2 for the open brace feature cell because the majority of open braces are not inline and at the end of the line. The lab has S2 style in the multi-statement per line feature cell because code has 2 statements (return x; and return y;) on the same line. The lab has an empty cell for the feature initialized variable declaration because code has no variable declaration, so this feature does not apply.

Table 6.3: 15 Python code style features with their weight (in parentheses), explanation (including the mapping function such as unanimity, majority, single instance or threshold to map a feature to its style variation), and style variation (Style1 and Style2) that we used in our approach to detect code style variation in Python labs.

No.	Feature	Explanation / Mapping (M)	Style1	Style2
1	Quotes in print() (0.5)	Using single quotes vs double quotes in print(). M: Majority.	Most prints use single quotes. Example: print('Hello')	Most prints use double quotes. Example: print("Hello")
2	Space after operator (0.5) (this includes any operator that ends with = such as !=, ==, <=, >=, etc.	Using space after operator vs not. M: Majority.	Most operators use space after them. Example: x = 1	Most operators do not use space after them. Example: x =1
3	Comment (0.5)	Using comments vs not. M: Single instance.	At least one comment is used. Example: # Compute max	No comment is used. Example: N/A
4	Formatted comment (0.5)	A formatted comment is a comment that starts with a capital letter and ends with a period. Using formatted comments vs not.	All comments are formatted. Example: #Compute max.	At least one comment is not formatted. Example: #compute max

		M: Unanimity.		
5	Inline comment (0.5)	An inline comment is a comment located on the same line with a line of code. Using inline comments vs not. M: Single instance.	At least one inline comment is used. Example: if (...) …#Compute max	No inline comment is used. Example: if (...) …
6	Commented functions/methods (0.5)	Using comments with all functions/methods vs not. M: Unanimity.	All functions/methods have comments. Example: # This function gets min int min(...) {	At least one function/method has no comment. Example: int min(...) {

7	Printing newline (0.5)	Using empty print() vs print() with ‘\n’ to print a new line.
		M: Majority.
			Most printing newlines use empty print().
			Example:
			print(‘Hello’)
			print()
				Most printing newlines use print() with ‘\n’.
				Example:
				print(‘Hello \n’)

7	Variable placeholder in print (0.5)	Not using variable placeholders (such as % or {}) vs using variable placeholders. M: Majority.	Most prints do not use variable placeholders. Example: print(‘Hello’, x)	Most prints use variable placeholders compared. Example: print(‘Hello {x}’)
9	Variable increment style (0.5)	Variable increment by one: Simple increment expression + 1 vs compound increment expression += 1 (this includes - 1 vs -= 1) M: Majority.	Most variable value increments by one use simple increment expression + 1. Example: x = x + 1	Most variable value increments by one use compound increment expression += 1. Example: x += 1
10	Space after comma (0.5)	Using space after comma vs not. M: Majority.	Most commas have space after them. Example: print(x, y)	Most commas have no spaces after them. Example: print(x,y)
11	Underscore in identifiers (0.5)	Using underscores (_) in identifiers vs not. M: Majority.	Most identifiers have underscores. Example: first_letter = “a”	Most identifiers do not have underscores. Example: firstletter = “a”

12	Using __name__ (0.5)	Using built-in variable__name__ to identify the main module vs not.
		M: Single instance.
			Using __name__ to identify the main module.
			Example:
			if __name__ == ‘main’:
			  x = 1
				Not using __name__ to identify the main module.
				Example:
				x = 1

13	Keyword with space (0.5)	Control structure keywords for branching and iteration can have an open parenthesis after them such as if (, while (, etc. Control keywords with no space before the opening parenthesis vs not. M: Majority.	Most control keywords have no space between them and the open parenthesis. Example: if(...)	Most control keywords have space between them and the open parenthesis. Example: if (...)

14	Left indentation (0.5)	Using 4 spaces for code left indentation vs not.
		M: Threshold of 4.
			Using 4 spaces for left indentation.
			Example:
			for …
			    if …
				Not using 4 spaces for left indentation.
				Example:
				for …
				  if …

15	Empty lines (0.5)	Using empty lines in code vs not.
		M: Single instance.
			At least one empty line is used.
			Example:
			for …
			
			    if …
				No empty line is used.
				Example:
				for …
				    if …

The second lab has S2 for the open brace feature cell because the majority of open braces are not inline and not at the end of the line. The lab has S1 style in the multi-statement per line feature cell because code has no multi-statement per line. The lab has an empty cell for the feature initialized variable declaration because code does not use any variable declaration, so this feature does not apply.

The third lab has S1 for the open brace feature cell because the majority of open braces are inline and at the end of the line. The lab has S1 in the multi-statement per line feature cell because code is not using multi-statement per line. The lab has S2 for the feature initialized variable declaration cell because the student code has at least one initialized variable declaration (double pi = 3.14;).

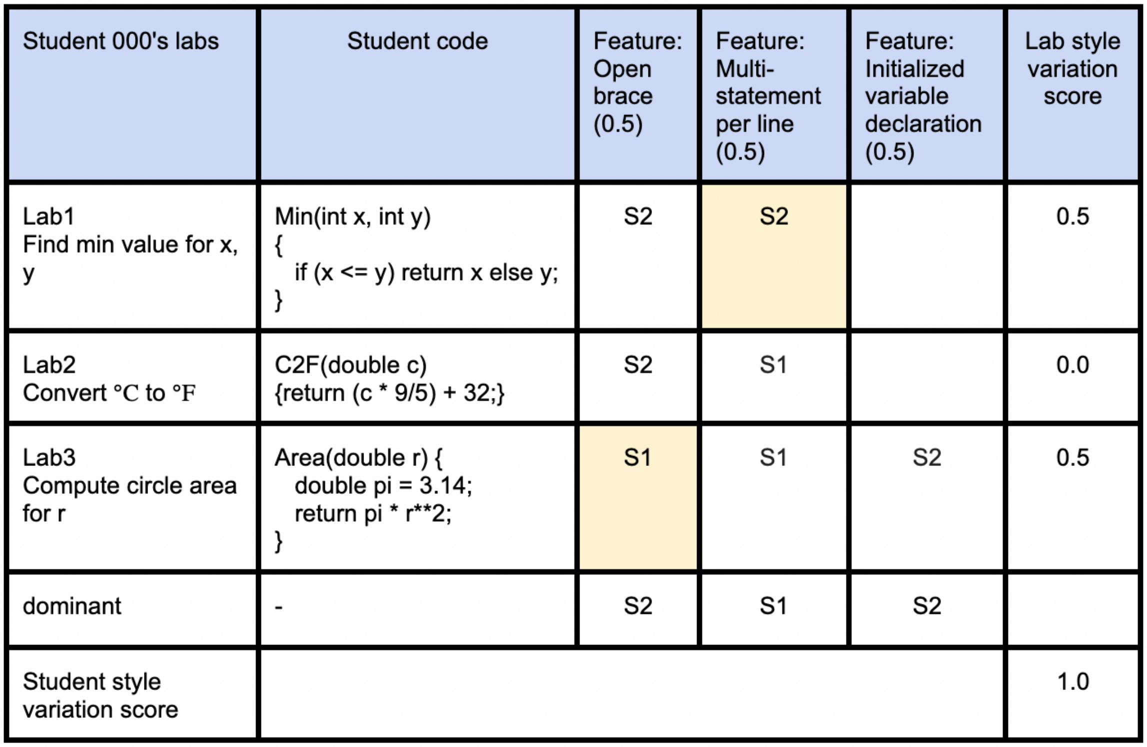
To find student style variation score, the last row in Table 6.4, we (1) Find the majority style for each feature, (2) Find lab style variation score, last column, by adding the weight of minority styles (styles that departs from the majority styles), and (3) Add lab style variation scores to get student style variation score. The majority style for the open brace feature is S2, and the minority style is S1 (highlighted in yellow). For the multi-statement per line feature, the majority style is S1, and the minority style is S2 (highlighted in yellow). For the initialized variable declaration feature, the majority style is S2, and the minority style is empty since there is no style detected for this feature in the first 2 labs (Lab1 and Lab2).

The row before the last row labeled Majority summarizes the majority style for each feature. The minority style (yellow cells) represents departure from a student's own majority style, suggesting that maybe that code was not written by the student. If the number of minority is the same as the number of majority, then the Majority column will show a value of “No majority”, which also signals a style variation.

To find a lab style variation score for a given lab, we add the weight of each minority style (yellow cell) for a lab across the 3 features. For Lab1, there is only one minority feature (yellow cell), the multi-statement per line feature with its weight of 0.5. So, Lab1 has a lab style variation score of 0.5. For Lab2, there is no minority feature (no yellow cells). So, Lab2 has a lab style variation score of 0.0. For Lab3, there is only one minority feature (yellow cell), the open brace feature with its weight of 0.5. So, the Lab3 has a lab style variation score of 0.5. The last column titled “Lab style variation score” summarizes the style variation score for each lab. To find the student style variation score, we add the lab style variation scores (the last column). The last row labeled “Student style variation score” shows the score, which is 1.0 in this case (0.5 + 0.5).

Our tool is written in Python 3.7, using Common Gateway Interface web technology, in about 4.5K lines of code [3]. The tool uses string matching such as regular expressions and built-in Python string comparison functions to detect code styles in student code. The tool uses nested dictionary data structures to build a student code style profile and compare student lab styles to detect style variation.

Table 6.4: A student matrix showing three C++ labs for the same student 000, showing three code style features out of 15 for brevity. The feature cells indicate whether that lab used Style1 (S1) or Style2 (S2) for that feature (blank means the feature didn't exist in code). Then: (1) a majority style (before the last row) is determined for each feature, (2) cells that differ from the majority are highlighted in yellow, (3) a "Lab style variation score" (right) is calculated for each lab as the sum of 1*weight for each highlighted cell, and (4) a “Student style variation score” (bottom) is calculated by adding each lab “Lab style variation score”.



Student 000's labs	Student code	Feature: Open brace (0.5)	Feature: Multi-statement per line (0.5)	Feature: Initialized variable declaration (0.5)	Lab style variation score

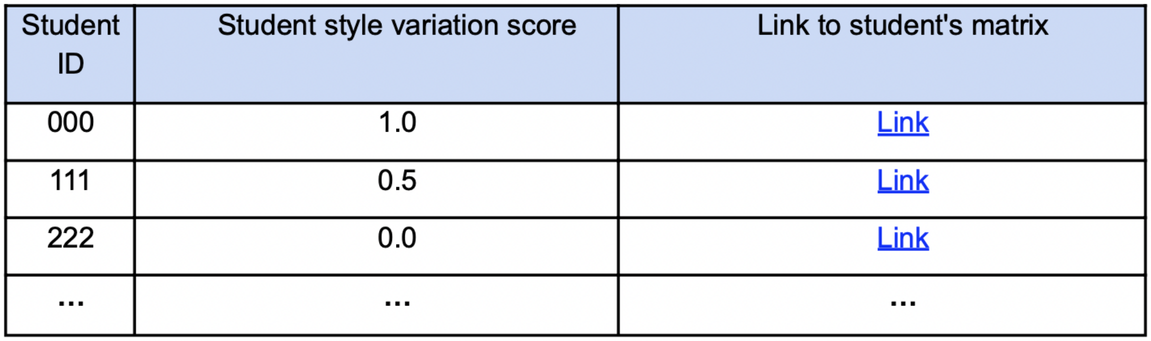
Lab1
Find min value for x, y
	Min(int x, int y)
	{
	if (x <= y) return x else y;
	}
		S2	S2		0.5

Lab2
Convert °C to °F
	C2F(double c)
	{return (c * 9/5) + 32;}
		S2	S1		0.0

Lab3
Compute circle area for r
	Area(double r) {
	double pi = 3.14;
	return pi * r**2;
	}
		S1	S1	S2	0.5

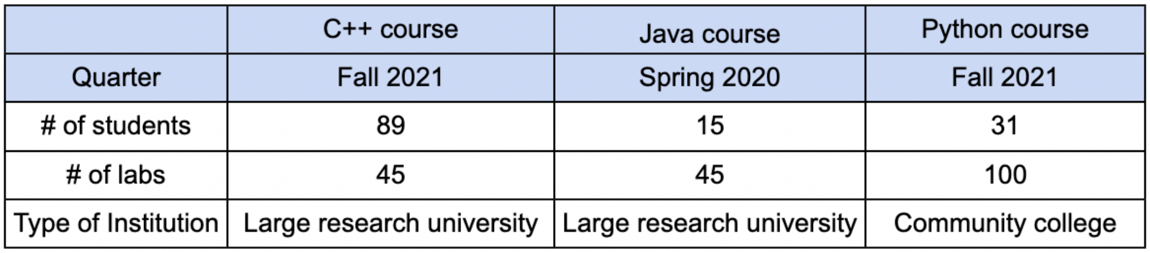
dominant	-	S2	S1	S2	
Student style variation score					1.0

Table 6.5: Student roster sorted by student style variation score. Instructors can click on the link to view a student matrix shown in Table 6.4.



Student ID	Student style variation score	Link to student's matrix
000	1.0	Link
111	0.5	Link
222	0.0	Link
...	...	...

Table 6.6: The three courses used in the experiment are presented by language.



	C++ course	Java course	Python course
Quarter	Fall 2021	Spring 2020	Fall 2021
# of students	89	15	31
# of labs	45	45	100
Type of Institution	Large research university	Large research university	Community college

6.2.3 Experiments

To test the efficacy of our tool to detect style variation, we conducted an experiment using real data. The experiment used zyBooks lab log files for three courses in three programming languages (C++, Java, and Python) as shown in Table 6.6. To simplify the experiment, we conducted the experiment using students with no-zero score in the first three labs, and using three common code style features for each language. We picked the three features, based on our judgment, that were most commonly the source of code style variation.

For the C++ course, we used the following features: control keyword space, code left aligned, and empty lines. For the Java course, we used: late variable declaration, control keyword space, and for-loop variable declaration. For the Python course, we used: quotes in print(), space after comma, and using __name__.

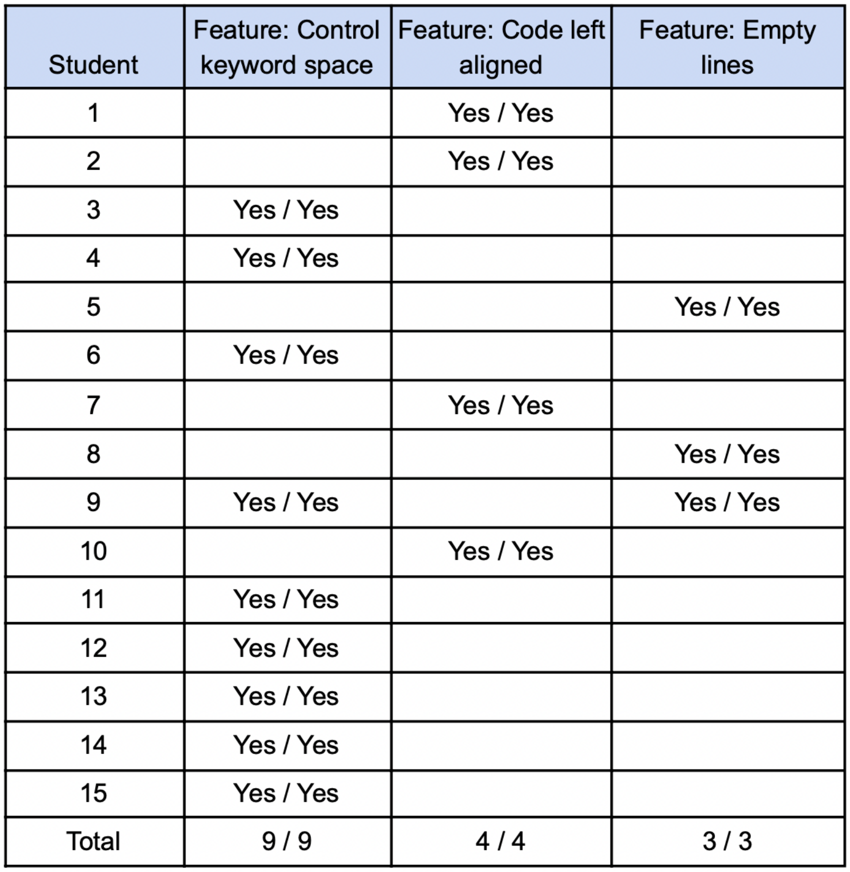
For each course, we picked 30 students, 15 students with style variation, and 15 students with no style variation. For the 15 students with style variation, for each student, we made sure to have at least three instances of style variation in each of the three features. We manually examined each student's three labs for style variation in the three features, and manually recorded Yes for style variation, or No for no style variation based on visual determination. Then, we ran our tool on each student and recorded the tool results. If the tool showed that a student used a consistent style (S1 or S2) for each of the three features in the three labs, then we recorded No for no style variation, otherwise, we recorded Yes for style variation. Tables 6.7, 6.8, and 6.9 summarize the manual and the tool results for the 15 students with style variation in the three courses (the 15 students with no style variation are not shown for brevity). The tool scored 100% accuracy using sensitivity and specificity in the three courses as shown in Table 6.10.

6.3 Discussion

We have found the tool useful in detecting possible cheating in real classes. The following three real examples show compelling cheating cases detected by the tool in the three courses (C++, Java, and Python) that we used in the experiment.

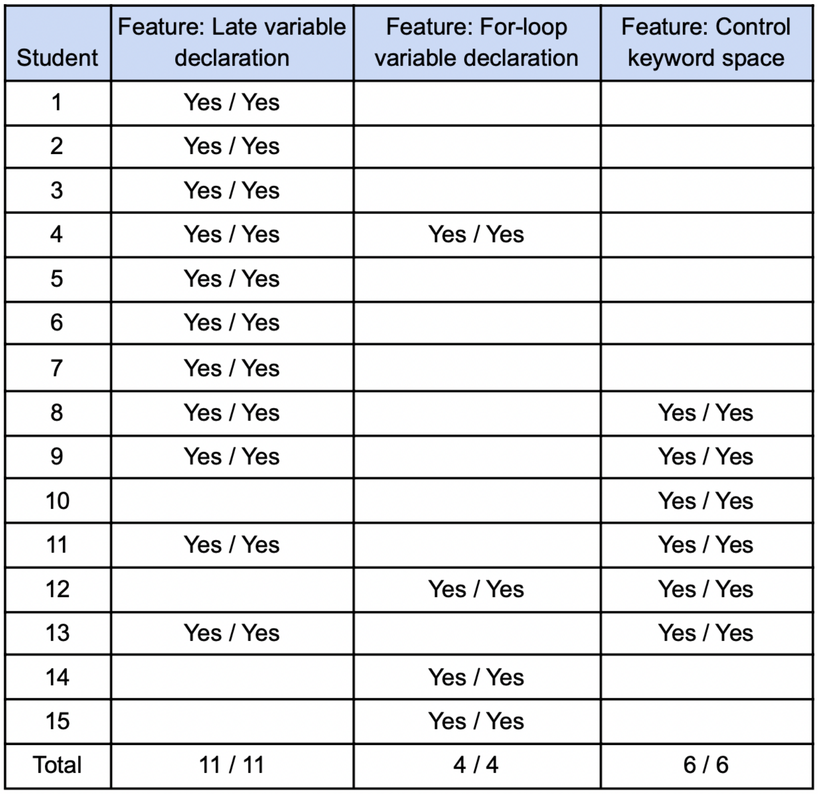
Table 6.11 shows a real code style variation example for a single student in three C++ labs. The student used different style variations for the following features: open brace, control keyword space, code left aligned, and empty lines.

Table 6.7: 15 Students in the C++ course with style variation in the three features (control keyword space, code left aligned, and empty lines), the columns' labels. In X / Y, X stands for the manual result, and Y stands for tool result. Yes / Yes means both manual and tool results showed a student having a style variation for that feature. The last row titled Total showed that both the manual and tool results found 9 instances of style variation for control keyword space, 4 instances of code left aligned, and 3 instances of empty lines.



Student	Feature: Control keyword space	Feature: Code left aligned	Feature: Empty lines
1		Yes / Yes	
2		Yes / Yes	
3	Yes / Yes		
4	Yes / Yes		
5			Yes / Yes
6	Yes / Yes		
7		Yes / Yes	
8			Yes / Yes
9	Yes / Yes		Yes / Yes
10		Yes / Yes	
11	Yes / Yes		
12	Yes / Yes		
13	Yes / Yes		
14	Yes / Yes		
15	Yes / Yes		
Total	9 / 9	4 / 4	3 / 3

Table 6.8: 15 students in the Java course with style variation in the three features (late variable declaration, for-loop variable declaration, and control keyword space), the columns' labels. In X / Y, X stands for the manual result, and Y stands for the tool's result. Yes / Yes means both manual and tool results showed a student having a style variation for that feature. The last row titled Total showed that both the manual and tool results showed a total of 11 instances of style variation for late variable declaration, 4 instances of for-loop variable declaration, and 6 instances of control keyword space.



Student	Feature: Late variable declaration	Feature: For-loop variable declaration	Feature: Control keyword space
1	Yes / Yes		
2	Yes / Yes		
3	Yes / Yes		
4	Yes / Yes	Yes / Yes	
5	Yes / Yes		
6	Yes / Yes		
7	Yes / Yes		
8	Yes / Yes		Yes / Yes
9	Yes / Yes		Yes / Yes
10			Yes / Yes
11	Yes / Yes		Yes / Yes
12		Yes / Yes	Yes / Yes
13	Yes / Yes		Yes / Yes
14		Yes / Yes	
15		Yes / Yes	
Total	11 / 11	4 / 4	6 / 6

Table 6.9: 15 students in the Python course with style variation in the three features (quotes in print(), space after comma, and using __name__), the columns' labels. In X / Y, X stands for the manual result, and Y stands for tool's result. Yes / Yes means both manual and tool results showed a student having a style variation for that feature. The last row titled Total showed that both the manual and tool results showed a total of 9 instances of style variation for quotes in print(), 7 instances of space after comma, and 5 instances of using __name__.

Student	Feature: Quotes in print()	Feature: Space after comma	Feature: Using __name__
1	Yes / Yes		
2	Yes / Yes		
3		Yes / Yes	
4	Yes / Yes		
5	Yes / Yes		
6	Yes / Yes		Yes / Yes
7	Yes / Yes	Yes / Yes	Yes / Yes
8			Yes / Yes
9	Yes / Yes	Yes / Yes	
10	Yes / Yes		Yes / Yes
11		Yes / Yes	
12		Yes / Yes	
13		Yes / Yes	
14		Yes / Yes	
15	Yes / Yes		Yes / Yes
Total	9 / 9	7 / 7	5 / 5

Table 6.10: Tool accuracy presented by language.

Language	Sensitivity	Specificity
C++	100%	100%
Java	100%	100%
Python	100%	100%
Average	100%	100%

Table 6.12 shows another real code style variation example for a student in three Java labs. The student used different style variations for the open brace, initialized variable declaration, for-loop variable increment style, for-loop variable declaration, code left aligned, and empty lines features.

Table 6.13 shows a third real code style variation example for a single student in three Python labs. The student used different code style variations in the labs. For example, the student used different styles of quotes in print(), space after operator, variable placeholder in print, space after comma, using __name__, and empty lines features.

The three examples show that the labs in each course are unlikely to be written by the same author.

The roster makes it easy for instructors to focus on students having the most style variation across labs, since the roster can be sorted by descending student style variation scores. Also, the student matrix helps to quickly confirm suspicious submissions instead of examining each student's solution manually for code style variation. A manual inspection is tedious and time-consuming to perform on a single student with many labs (such as 30 labs or more per student), let alone a whole class of many students (such as 50 students or more). Instead, an instructor can view a student’s matrix to glance through all labs for a single student in a short period of time (less than a minute), and then in some cases may decide to look more carefully at each code submission.

The student matrix is sortable by any column. For example, an instructor may sort a student matrix by lab style variation score (lab score), which helps an instructor to first examine labs with most style variation. Table 6.15 shows a real matrix sorted by lab score for a single student in a C++ course with 33 labs (the matrix legend is shown in Table 6.14). At the bottom of the matrix, the tool also provides access to labs' data as shown in Table 6.16. A lab data contains valuable information such as lab caption, lab number, lab max score, student code, student score, student's submission timestamp, etc. An instructor can easily jump from a lab style variation feature row in a student matrix to the lab corresponding data by clicking on the lab number in the matrix, which is a hyperlink to the lab's data (the first column labeled Lab in Table 6.15). Also, from the lab data at the bottom of the matrix, an instructor can jump back up to the lab style variation feature row in the matrix by clicking on the lab number in the lab data table, which is a hyperlink (the second column labeled "Content section" in Table 6.16).

Some instructors might happen to have access to first-highest code runs in addition to the usual last-highest code runs, or even the last code runs. Our tool is orthogonal to the code run types (first-highest, last-highest, or last code runs) and works on any of them. Some instructors might discover more student attempts to cheat using first-highest code runs, where students might copy code in the first-highest code runs and change the code style to reduce code style variation in the last-highest code runs to trick our tool. However, using first-highest code runs might not give such students a second chance to backtrack from submitting copied code when they change their mind and decide not to cheat.

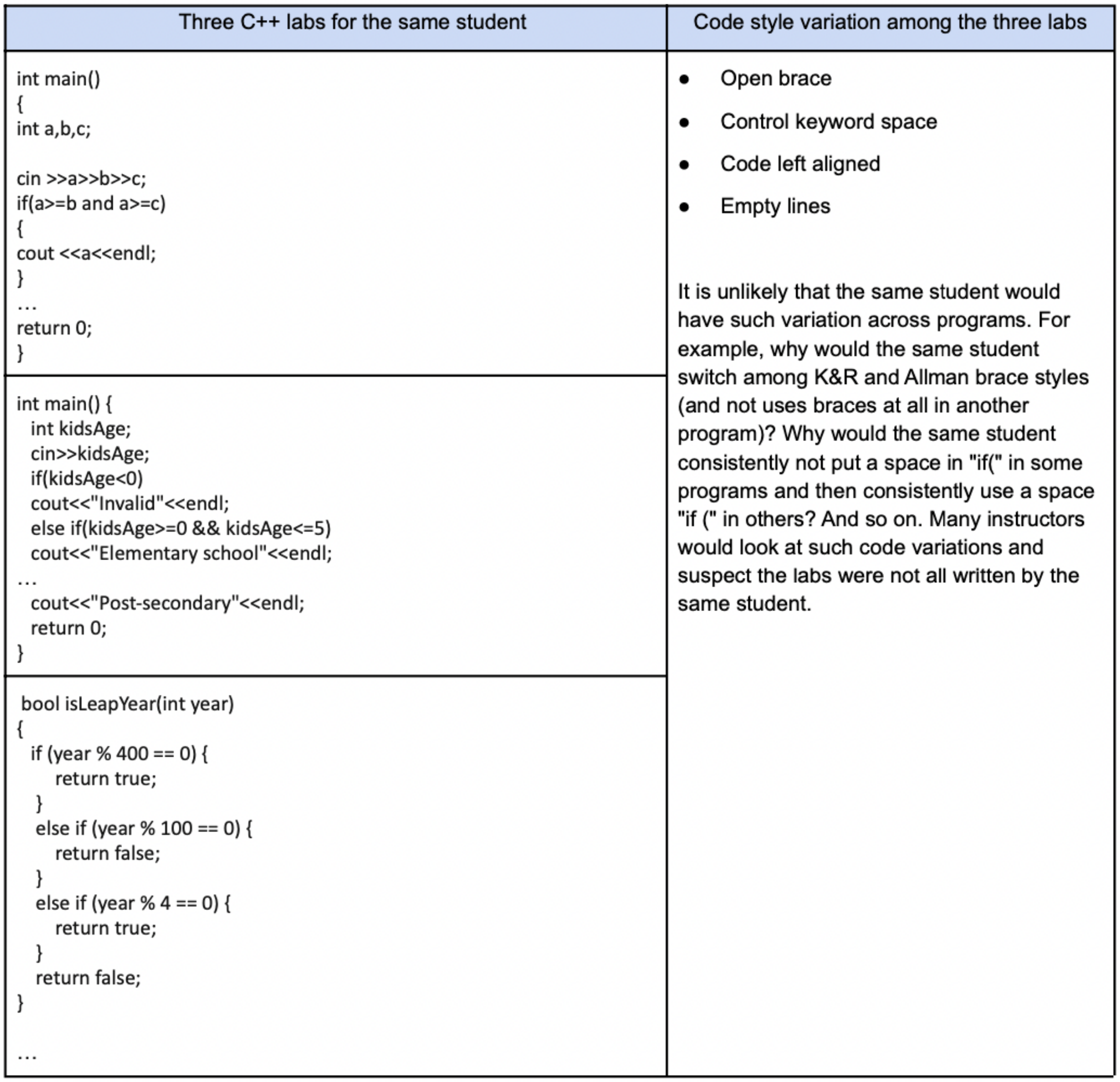
It is possible for a single lab solution by itself to have the same number of instances (or greater than “very similar”, but not majority as we explained before in the section “code style variation”, when we discussed the majority mapping function) in both styles for the same feature. For example, If the number of instances for both inline and non-inline open braces is the same or very similar for the open brace feature. This could be due to cheating by copying a portion of a solution, like a function. We call such a problem “code style variation within a lab” and we believe it differs slightly from the problem that we are addressing in this paper, which is “code style variation across labs for the same student”. Our tool currently does not address the “code style variation within a lab” problem, but we plan to address it in future work.

We chose the code style features, weights, mapping functions, and thresholds in our tool based on our observations and experience of having taught CS1 to hundreds of students per term for many years at our institution, using about 40-50 programming assignments per term (some shorter, some longer), catching about 10-20 students per term flagrantly cheating. Different settings could be considered. We plan to make the tool configurable, where instructors can adjust these settings to enable the tool to adapt to their preferences. Our code style variation approach will not detect if a student contracts with one other person to write all the student's programs. Code similarity also won't detect that case. Fortunately, based on our observations, such cases are much rarer than students who have different sources of cheating for different labs (e.g., on the popular low-cost online cheating websites, different "tutors" post solutions to requests even from the same student). However, for such contract cases, other techniques can help, such as detecting unusually little time spent, detecting IP addresses from other locations/countries, or detecting style that is consistent across labs but that departs from the class style. We have caught students cheating via contract programmers using such other means, but details are beyond this paper's scope.

6.4 Conclusion

We presented a complementary approach to the traditional code similarity approach (such as Moss) to detect possible cheating in programming lab solutions. The traditional approach checks the same lab across different students for similar code. However, the traditional approach is not effective if a student's solutions are unique, which can happen if they find a unique solution online, get solutions from “tutors” on today’s popular help websites, or use tricks to modify copied code. Instead, our approach checks different labs for code style variation for the same student. We introduced a tool to detect code style variation, which can supplement code similarity detectors. The experiments show that our tool is effective and neared 100% sensitivity and specificity in detecting code style variation for several key features, and hence for detecting possible cheating. We plan to make our tool available to the CS community as a free web tool.

Table 6.11: A real code style variation example for the same student in three C++ labs.



Three C++ labs for the same student	Code style variation among the three labs

int main()
{
int a,b,c;
 	
cin >>a>>b>>c;
if(a>=b and a>=c)
{
cout <<a<<endl;
}
…
return 0;
}
	● Open brace
	● Control keyword space
	● Code left aligned
	● Empty lines
	
	It is unlikely that the same student would have such variation across programs. For example, why would the same student switch among K&R and Allman brace styles (and not uses braces at all in another program)? Why would the same student consistently not put a space in "if(" in some programs and then consistently use a space "if (" in others? And so on. Many instructors would look at such code variations and suspect the labs were not all written by the same student.

int main() {
  int kidsAge;
  cin>>kidsAge;
  if(kidsAge<0)
  cout<<"Invalid"<<endl;
  else if(kidsAge>=0 && kidsAge<=5)
  cout<<"Elementary school"<<endl;
…
  cout<<"Post-secondary"<<endl;
  return 0;
}	

bool isLeapYear(int year)
{
  if (year % 400 == 0) {
      return true;
   }
   else if (year % 100 == 0) {
      return false;
   }
   else if (year % 4 == 0) {
      return true;
   }
   return false;
}
 	
…	

Table 6.12: A real code style variation example for the same student in three Java labs.

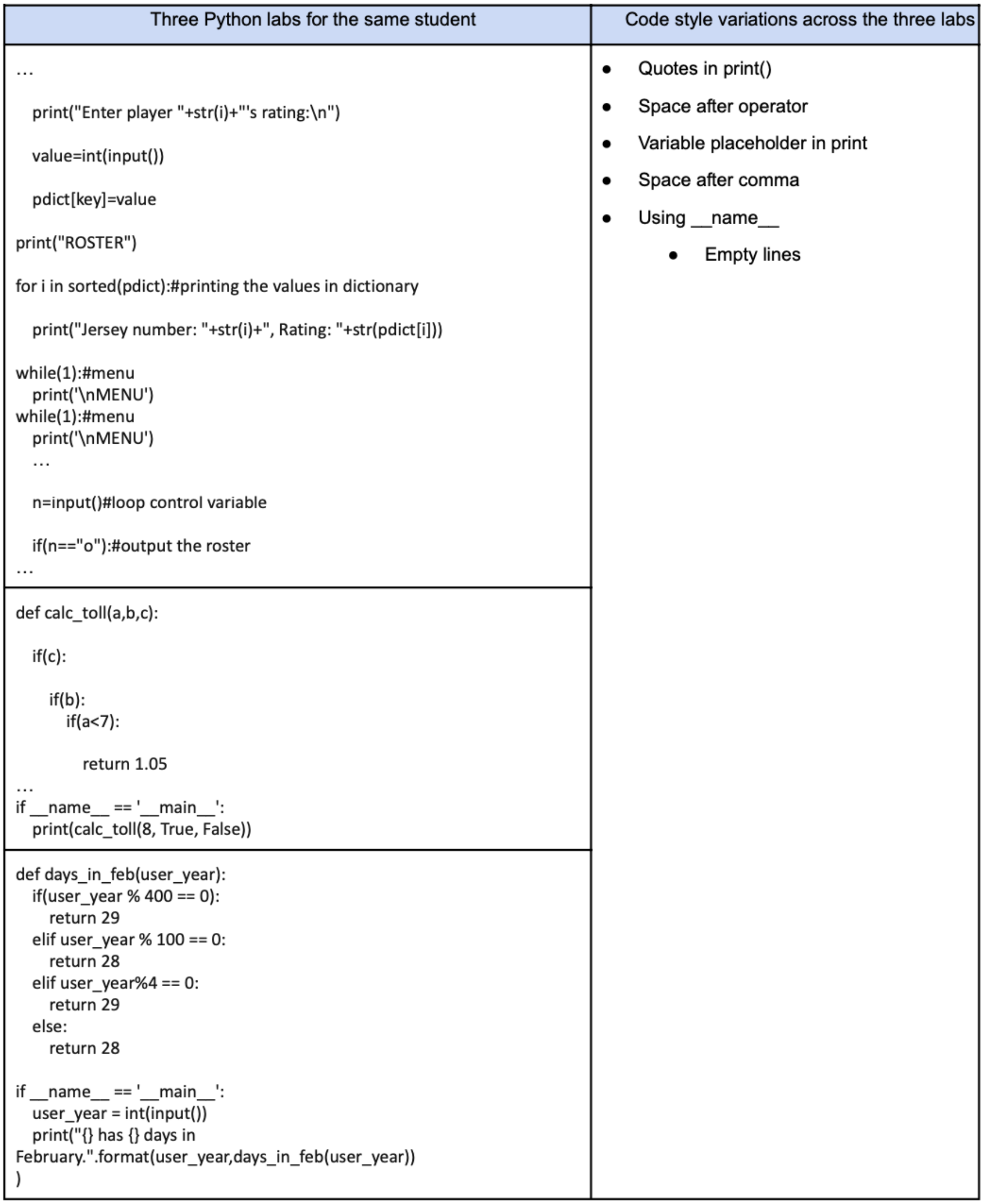
3 Java labs for the same student	Code style variation among the 3 labs

public class PeopleWeights {
  public static void main(String args[])
{
Scanner sc=new Scanner(System.in);
double m[]=new double[5];
double weight = 0, avg, max = 0;
int i;
for (i = 0; i < 5; i++)
{
System.out.println("Enter weight " + (i + 1) + ":");
m[i] = sc.nextDouble();
}
…
	● Open brace
	● Initialized variable declaration
	● For-loop variable increment style
	● For-loop variable declaration
	● Code left aligned
	● Empty lines
	
	It is unlikely that the same student would have such variation across programs. The first lab is left aligned, lacks any empty lines, and uses Allman brace style, while the latter 2 are more typical by using indents, blank lines, and K&R brace style. The first lab declares "int i;" outside the for loop while the third declares it inside the for loop. It is likely that the first lab was written by someone different from the latter 2.

public class DrawRightTriangle {
  public static void main(String[] args) {
    Scanner scnr = new Scanner(System.in);
    char triangleChar;
    int triangleHeight;
 	
    System.out.println("Enter a character:");
    triangleChar = scnr.next().charAt(0);
 	
    System.out.println("Enter triangle height:");
    triangleHeight = scnr.nextInt();
    System.out.println("");
 	
    for(int row = 0; row < triangleHeight; row++) //Outer Loop = Rows: /n
  {
      for (int col = 0; col < row +1; col++) //Inner Loop = Cols: '*'
      {
        System.out.print(triangleChar + " ");
      }
      System.out.println();
…	

public class LabProgram {
 	
   public static String removeSpaces(String userString) {
      StringBuilder result = new StringBuilder();
      for (int i = 0; i < userString.length(); ++i) {
         if (userString.charAt(i) != ' ') {
            result.append(userString.charAt(i));
         }
      }
      return result.toString();
   }
:::
   }
}
…	

Table 6.13: A real code style variation example for the same student in three Python labs.



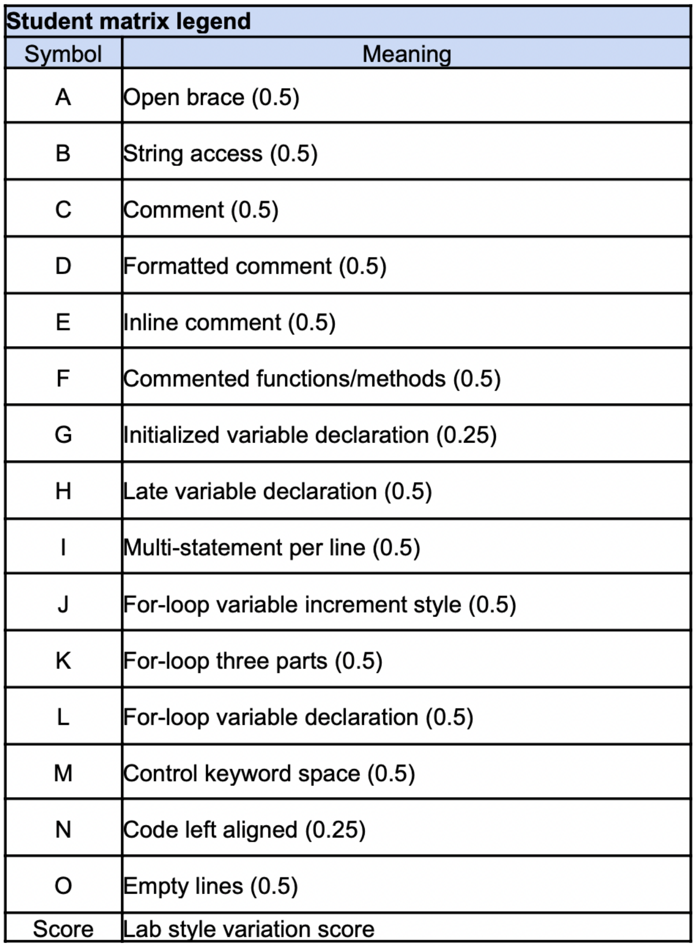
Three Python labs for the same student	Code style variations across the three labs

…
print("Enter player "+str(i)+"'s rating:\n")
value=int(input())
pdict[key]=value
print("ROSTER")
for i in sorted(pdict):#printing the values in dictionary
print("Jersey number: "+str(i)+", Rating: "+str(pdict[i]))
while(1):#menu
print('\nMENU')
while(1):#menu
print('\nMENU')
…
n=input()#loop control variable
if(n=="o"):#output the roster
…
	● Quotes in print()
	● Space after operator
	● Variable placeholder in print
	● Space after comma
	● Using __name__
	● Empty lines

def calc_toll(a,b,c):
if(c):
if(b):
if(a<7):
return 1.05
…
if __name__ == '__main__':
print(calc_toll(8, True, False))	

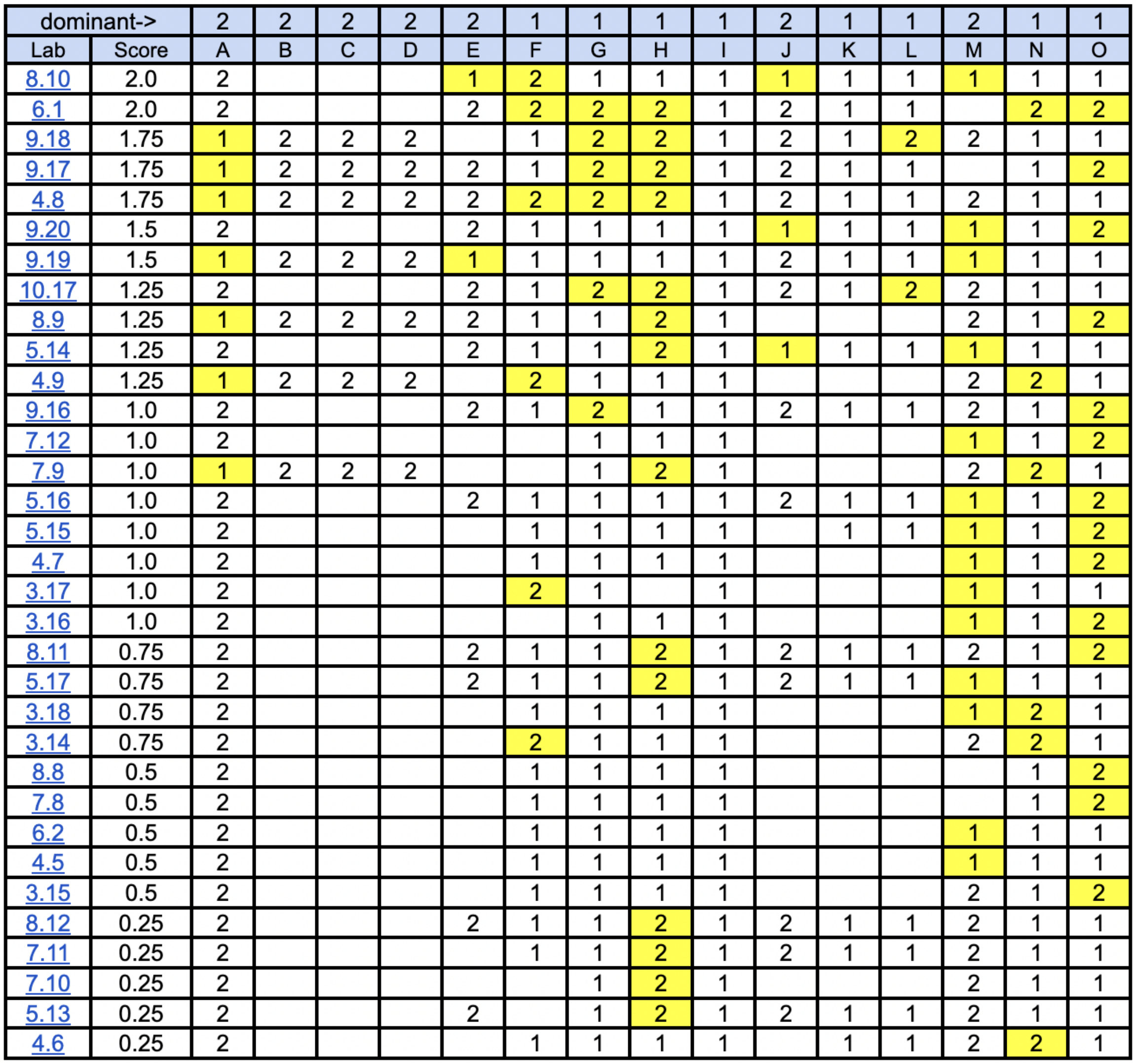
def days_in_feb(user_year):
if(user_year % 400 == 0):
return 29
elif user_year % 100 == 0:
return 28
elif user_year%4 == 0:
return 29
else:
return 28
if __name__ == '__main__':
user_year = int(input())
print("{} has {} days in February.".format(user_year,days_in_feb(user_year))
)	

Table 6.14: Student matrix legend.



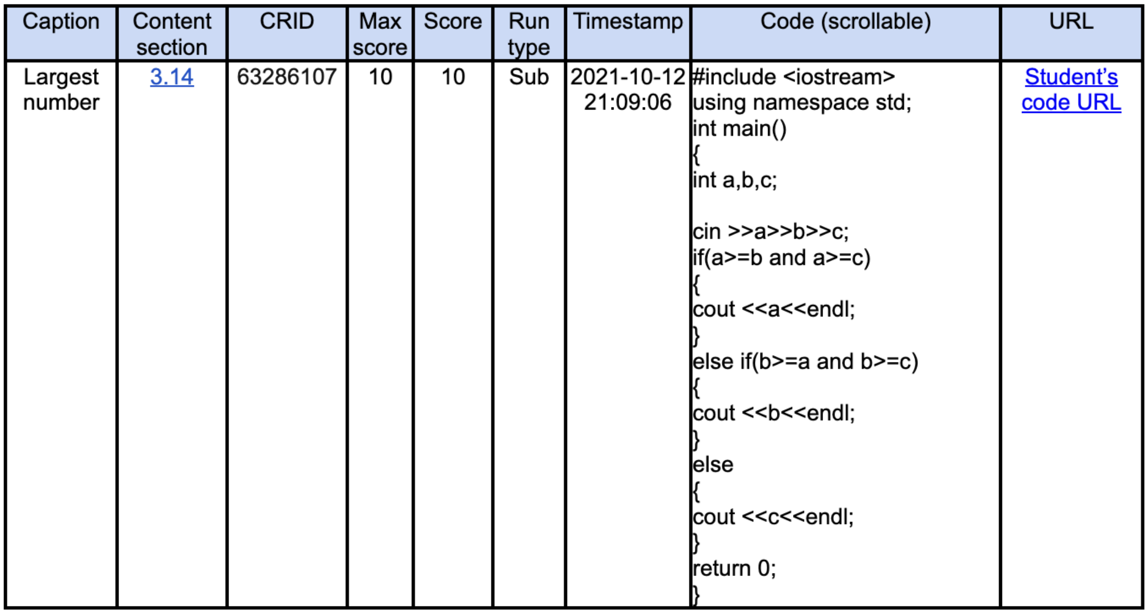
Student matrix legend	
Symbol	Meaning
A	Open brace (0.5)
B	String access (0.5)
C	Comment (0.5)
D	Formatted comment (0.5)
E	Inline comment (0.5)
F	Commented functions/methods (0.5)
G	Initialized variable declaration (0.25)
H	Late variable declaration (0.5)
I	Multi-statement per line (0.5)
J	For-loop variable increment style (0.5)
K	For-loop three parts (0.5)
L	For-loop variable declaration (0.5)
M	Control keyword space (0.5)
N	Code left aligned (0.25)
O	Empty lines (0.5)
Score	Lab style variation score

Table 6.15: A real student matrix for a single student in a C++ class with 33 labs. Intuitively, a lab (row) with many highlighted cells means that the lab departs from the student's "dominant" code style, and thus may have been written by someone else. A student with much yellow appearing across many rows may be cheating on many labs. An instructor may click on a lab number (the hyperlink) in the matrix column labeled “Lab” to jump to the lab data at the bottom of the matrix, which contains the lab details as shown in Table 6.16.



dominant->		2	2	2	2	2	1	1	1	1	2	1	1	2	1	1
Lab	Score	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8.10	2.0	2				**1**	**2**	1	1	1	**1**	1	1	**1**	1	1
6.1	2.0	2				2	**2**	**2**	**2**	1	2	1	1		**2**	**2**
9.18	1.75	**1**	2	2	2		1	**2**	**2**	1	2	1	**2**	2	1	1
9.17	1.75	**1**	2	2	2	2	1	**2**	**2**	1	2	1	1		1	**2**
4.8	1.75	**1**	2	2	2	2	**2**	**2**	**2**	1	2	1	1	2	1	1
9.20	1.5	2				2	1	1	1	1	**1**	1	1	**1**	1	**2**
9.19	1.5	**1**	2	2	2	**1**	1	1	1	1	2	1	1	**1**	1	1
10.17	1.25	2				2	1	**2**	**2**	1	2	1	**2**	2	1	1
8.9	1.25	**1**	2	2	2	2	1	1	**2**	1				2	1	**2**
5.14	1.25	2				2	1	1	**2**	1	**1**	1	1	**1**	1	1
4.9	1.25	**1**	2	2	2		**2**	1	1	1				2	**2**	1
9.16	1.0	2				2	1	**2**	1	1	2	1	1	2	1	**2**
7.12	1.0	2						1	1	1				**1**	1	**2**
7.9	1.0	**1**	2	2	2			1	**2**	1				2	**2**	1
5.16	1.0	2				2	1	1	1	1	2	1	1	**1**	1	**2**
5.15	1.0	2					1	1	1	1		1	1	**1**	1	**2**
4.7	1.0	2					1	1	1	1				**1**	1	**2**
3.17	1.0	2					**2**	1		1				**1**	1	1
3.16	1.0	2						1	1	1				**1**	1	**2**
8.11	0.75	2				2	1	1	**2**	1	2	1	1	2	1	**2**
5.17	0.75	2				2	1	1	**2**	1	2	1	1	**1**	1	1
3.18	0.75	2					1	1	1	1				**1**	**2**	1
3.14	0.75	2					**2**	1	1	1				2	**2**	1
8.8	0.5	2					1	1	1	1					1	**2**
7.8	0.5	2					1	1	1	1					1	**2**
6.2	0.5	2					1	1	1	1				**1**	1	1
4.5	0.5	2					1	1	1	1				**1**	1	1
3.15	0.5	2					1	1	1	1				2	1	**2**
8.12	0.25	2				2	1	1	**2**	1	2	1	1	2	1	1
7.11	0.25	2					1	1	**2**	1	2	1	1	2	1	1
7.10	0.25	2						1	**2**	1				2	1	1
5.13	0.25	2				2		1	**2**	1	2	1	1	2	1	1
4.6	0.25	2					1	1	1	1		1	1	2	**2**	1

Table 6.16: The lab data for lab 3.14 is at the bottom of the student matrix in the previous table (Table 6.15). The instructor may click on the lab number (the hyperlink) in the lab data columns labeled “Content section” to jump back to the student matrix to view the student style variation for this lab.



Caption	Content section	CRID	Max score	Score	Run type	Timestamp	Code (scrollable)	URL

Largest number	3.14	63286107	10	10	Sub	2021-10-12 21:09:06	#include <iostream>
							using namespace std;
							int main()
							{
							int a,b,c;
							
							cin >>a>>b>>c;
							if(a>=b and a>=c)
							{
							cout <<a<<endl;
							}
							else if(b>=a and b>=c)
							{
							cout <<b<<endl;
							}
							else
							{
							cout <<c<<endl;
							}
							return 0;
							}
								Student’s code URL

7. Detecting Possible Cheating in Programming Courses Using Class Code Style Anomaly

7.1 Introduction

To counter cheating in programming courses, instructors use traditional code similarity detection tools to detect similar code among students in class for the same lab, such as Moss [1], which is widely used for automated code similarity detection. However, the code similarity detection has limitations when it comes to unique code. Also, code similarity detection can be tricked by multiple techniques that are available online, which students can learn and use [2, 3, 4]. Therefore, complementary approaches are needed to supplement code similarity detection approaches.

We noticed that when students submit copied code, they focus on submitting working code, rather than working code that is compatible with the current class code style. For example, copied code often uses constructs that have not yet been taught in a class, such as using user-defined functions or arrays/vectors, before those constructs had been introduced. Some students use constructs that are not even covered in the class, such as continue/break statements or conditional operators. Some use styles that depart from the teacher/book style, like brace style, variable naming, or spacing conventions. For a given feature, like user-defined functions or brace style, we call code departing from the class' normal use of that feature a class code style "anomaly".

There are multiple ways to determine the class code style. One approach is to allow the instructor to specify the allowable styles, but that might be cumbersome since the instructor needs to specify the styles. A second approach is to use the instructor’s solution to estimate the allowable styles. However, the instructor's code might not exhibit all allowable styles. A third approach is to examine the entire class' code to determine if there is a majority style. Although this approach eliminates the need for the instructor's solution, it has limitations in small classes with high cases of cheating. Each approach has limitations, but a hybrid approach might help to overcome some of these limitations, which is what we used in our approach.

We developed an approach to detect possible cheating based on class code style anomalies. Our approach flags students' code that deviates from the class code style as possible cheating, due to possibly being written by someone outside the class who isn't familiar with the class' style. We use class code style (which is a combination of the styles used in the instructor's solution, and a list of allowable styles) to help determine what the current class code style is, and what it is not, for each lab. Table 7.1 summarizes the differences between the traditional code similarity approach and our class code style anomaly approach.

A limitation of our approach is that some instructors may not enforce any particular coding style, allowing students for example to use any brace style. However, many instructors do enforce a particular style, often matching the course textbook's style. Also, our approach still detects some anomalies even if no particular style is enforced, such as using user-defined functions before they were taught.

Our approach is orthogonal and complementary to the traditional code similarity approach. Both approaches direct instructors to possible cheating. Similar to the traditional code similarity approach, our approach does not label anything as “cheating”, but rather points instructors to suspicious cases that might require further investigation.

Table 7.1: The traditional code similarity approach compares each student’s code against other students’ code to find code similarity. Our approach compares the class code style with each student’s code style to find class code style anomalies. The approaches are complementary, pointing instructors to possible cheating.

	Traditional code similarity approach	**Class code style anomaly approach**
Input	Student’s code	Student’s code, and class code style
Compare	Statements	Style
Objective	Find similarity	Find anomalies
Output	Code similarity	Style anomalies

There are multiple coding styles. For example, with respect to brace usage, students might use Kernighan & Ritchie (K&R) or Allman style, as shown in Table 7.2. Notice the open brace is inline in K&R style, but it is standalone in Allman style. Our approach is orthogonal to code style standards and does not require any particular code style, but rather detects code style anomalies between instructor’s code and student’s code.

We developed a tool to detect possible cheating based on class code style anomalies. We plan to make our tool available for the CS community as a free web tool, akin to the Moss tool.

Table 7.2: K&R versus Allman brace code style.

K&R brace code style	Allman brace code style

if (x <= y) {
…
}
	if (x <= y)
	{
	…
	}

In this chapter, we explain the code features that we used in our tool to profile instructor code and student code; describe how the tool works to detect class code style anomalies; perform experiments to evaluate our tool's accuracy based on sensitivity and specificity metrics; and explain how the tool helped to detect multiple real cheating cases in our 83-student class.

7.2 Methodology

7.2.1 Class Code Style Anomalies

In this section, we define terms to use in the next section to explain how the class code style anomalies detection approach works.

A code ***feature*** is a language construct or a layout property of code. For example, constructs include user-defined functions, arrays/vectors, or the increment operator. Layout properties include how code braces are used or how variables are named.

In this chapter, we consider any feature to have only 2 possible styles, although more than 2 styles might be possible for some features.

A code feature style is a value for a feature. We classify a feature based on its styles into the following 2 types:

- Use-version style feature type. We consider this type of feature to have 2 use-version styles. For example, the brace feature has the 2 styles (inline or standalone); the increment operator has the 2 styles (prefix or postfix,), etc.
- Use-only style feature type. We consider this type of feature to have 2 use-only styles. For example, the user-defined function has 2 styles (used or not used in code), the array has 2 styles (used or not used), the printf (used or not used), etc.

An ***instructor's style*** is the style used for a feature in an instructor's solution. For example, an instructor's solution might use the inline style for the brace feature, prefix style for the increment operator feature, etc.

A ***student’s style*** is the style used for a feature in a student's code. For example, a student’s code might use the standalone style for the brace, postfix style for the increment operator, etc.

An ***allowed/disallowed list*** is a list of features with their styles that an instructor can set to disallow or unset to allow in student code (more details in the next section). An instructor can manually set/unset each feature with their available styles in the list, but to save time we automatically determine some of the allowed features with their styles from the instructor solution. For example, an instructor can disallow the style of using a feature, like not using printf, etc. However, If no feature is set in the list, any feature (and hence any style) will be allowed in student code regardless of the instructor's style.

A ***style anomaly*** is a difference between the instructor’s style and the student’s style for the same feature, or a student's use of a disallowed style. For example, an instructor's solution might use inline braces, while a student uses standalone braces, yielding an anomaly. Or, early in a term, an instructor might select the style of using the compound assignment operator += as a disallowed style, so student code using += yields an anomaly.

A ***class style*** is a style that is allowed in class by the instructor. An instructor can allow a style in class by using the style in the instructor's code or by not disallowing a style in the allowed/disallowed list.

Our class style anomalies detection approach is based on 2 assumptions:

- For a use-only style feature, if the instructor’s code uses a feature (which inherently sets its use style), then it is allowed in students’ code. However, if the feature is not used in the instructor's code and the feature use style is set as a disallowed style, then it is an anomaly. For example, a student can use array, printf, scanf, etc. as long as it is a class style (used in the instructor’s code).
- For a use-version style feature, our approach automatically makes the opposite corresponding use-version style of a feature (the style used in the instructor’s code) an anomaly (more details in the next section). For example, if the instructor’s code uses the prefix style for the increment operator and a student’s code uses the postfix style, our approach automatically flags the postfix style in students’ code as an anomaly.

To detect style anomalies, we first defined common features that we found commonly represent different styles between an instructor's code and a student's code. Table 7.3 presents 40 such features for C++ code.

We assign weights between 0 and 1 to features to signify each feature's importance in detecting style anomalies. Weights are used because not all features have the same importance in detecting style anomalies. In the table, many features (like Array) have a weight of 0.5, but others like IniVarDec have a weight of 0.3 because in our experience such initialization is not as indicative of coding by a person outside the class.

The mapping function maps a feature to one of its styles and adjusts the weight for some features. We define three types of mapping: Single instance, dominant instance, and threshold.

- A single instance mapping applies only to use-only style features. A single instance mapping means a single instance of the *used* style is enough to consider the existence of the style in code. For example, for the Array feature, a single instance of an array bracket [ ] in a student's code is enough to conclude that the student's code uses a C-style array.
- A dominant mapping applies only to use-version style features. If the number of instances of one of the feature 2 styles is zero, then the default weight of 0.5 applies. For example, for the OpenBrace feature, if the number of instances of standalone braces (or inline braces) is zero, then the weight is 0.5. However, if both of the 2 styles have a non-zero number of instances, then the mapping

function adjusts the weight. For example, for the OpenBrace feature, if a student's code uses 2 inline open braces, and 5 standalone open braces, the dominant mapping function picks the dominant style. The dominant style is the style with a larger number of instances, which is the standalone style in this case because it has 5 instances. The mapping function then adjusts the default OpenBrace feature's weight as follows: (1) Calculate the fraction of the dominant style (the number of instances of the dominant style, which is 5, divided by the total number of instances of both styles, which is 7), (2) multiply the dominant style fraction (which is 5/7) by the default feature's weight for OpenBrace (which is 0.5). This gives the adjusted weight of 5/7 * 0.5 or 0.4.

- A threshold mapping applies only to use-only style features. A threshold mapping first determines the frequency of a style before deciding whether the style represents an anomaly. For example, the LeftAligned feature has a threshold of 10%, meaning that more than 10% of lines of code must be left-aligned for the *LeftAligned: Yes* style to be detected, otherwise *LeftAligned: No* style is detected. If *LeftAligned: Yes* style is detected, then the default feature's weight applies (which is 0.3 for this feature).

Notice that the use-only style features (# 1 to # 34, except # 24 in Table 7.3) are the features with the single instance or threshold mapping functions, while the use-version style features (# 35 to # 40, in addition to # 24 in Table 7.3) are the features with the dominant instance mapping function.

Table 7.3: C++ features considered. Each feature has a mapping function: SI (single instance), DI (dominant instance), or a threshold. Each feature's mapping function is SI, and weight is 0.5 unless otherwise specified. Style examples are shown when relevant.

#	Feature (weight) (mapping)	Description	Style 1	Style 2
1	Array	Uses any array bracket	No	Yes item[0];
2	Pointer	Uses a pointer	No	Yes int *a;
3,4	Printf, Scanf	Using any printf or scanf	No	Yes printf(a), scanf(a)
5	Null	Using a null pointer	No	Yes a = '\0';
6	NewLine	Using any \n	No	Yes cout << "\n";
7	HHeaderFile	Using any C-style .h header file (in the #include directive)	No	Yes #include "string.h"
8 9 10 11 12 13 14	Break Continue Do Const Switch Float Cast	Using any break, continue, do, const, switch, float, or cast keyword	No	Yes
15	WhileTrue	Using any unconditional while (such as while(true) or while(1))	No	Yes while(1) {...
16	Function	Using any user-defined function	No	Yes int add(int x, int y)
17	Return	Using any return keyword	No	Yes return;
18	CamelCase (DI)	Using the camelcase naming convention with identifiers (including function, and variable names that have more than 1 character) We define a camelCase identifier as an identifier that has more than 1 character, and has both uppercase, and lowercase letters only.	Yes double avgSum;	No double avgsum;
19	Underscore	Using any underscore (_) in identifier	No	Yes first_num = 1;
20	IniVarDec (0.3)	Using any initialized variable declaration We define an initialized variable declaration as assigning a value when declaring a variable.	No	Yes int a = 0;

21	LateVarDec	Using any late variable declaration
		We define a late variable declaration as a variable declaration after a control structure keyword (such as while, for, if, else).
			No	Yes
				if … {
				int a;

22	MultiVarDec	Using any multi-variable declaration per line	No	Yes int a, b;
23	MultiVarRead	Using any multi-variable read per line	No	Yes cin >> a, b;
24	IncrementOp (DI)	Using a prefix or postfix (an increment or a decrement) operator	Prefix ++i;	Postfix i++;
25	CompAssign	Using any compound assignment operator (such as +=, -=, *=, /=, etc.)	No	Yes i += 1;
26	ScopeOp	Using any scope (::) operator (except for string::npos)	No	Yes std::cout ...
27	TernaryOp	Using any ternary (?) operator	No	Yes (a > b) ? a : b;
28	BoolAssig	Using any boolean assignment	No	Yes

		A boolean assignment is an assignment statement that has a boolean expression on the right-hand side.		isBigger = (a > b);

29	LeftAligned
	(0.3)
	(Threshold: 10%)
		Code is mostly left-aligned code
		More than 10% of code blocks are left aligned. A code block is a code enclosed by braces.
			No
			int main() {
			int a;
			cout << a;
			...
				Yes
				int main() {
				int a;
				cout << a;
				...

30	EmptyLines
	(0.3)
		Using empty lines in code
		If empty lines are used in code blocks.
			Yes
			int main() {
			int a; …
				No
				int main() {
				int a;
				cout <<

31	MultiEmptyLines
	(0.3)
	(Threshold: 30%)
		Using multiple consecutive empty lines in code.
		More than 30% of code blocks are empty (consisting of a group of at least 2 consecutive empty lines).
			No
			int main() {
			int a;
			cout << a;
			...
				Yes
				int main() {
				int a; …

32	InlineTemplateComment	Template code comment is inline with student's code A template comment like /* Type your code here */ usually stays in its own line at the top of the student's code if a student types their code. If the comment is inline with the student's code that might indicate a student copied and pasted code.	No	Yes int main() { int a; /* Type your code here */ cout << a; ...
33	AtEndTemplateComment	Using a template comment toward the end of a student's code A template comment like /* Type your code here */ usually starts at the top of code if a student types their code, otherwise it might indicate a student copied and pasted code.	No	Yes int main() { … Return 0; /* Type your code here */
34	MultiStatement	Using a multi-statement per line.	No	Yes a = 1; b = 2;
35	OpenBrace (DI)	Using an open brace.	Inline If (...) { …	Standalone if (...) { …
36	ForLoop3parts (DI)	Using a for-loop with 3 parts.	Yes for (…; …; …)	No for (…;...)

37	ForLoopCounterDec
	(DI)
		Using outside or inside for-loop counter declaration	Outside
			int i;
			for (i = 1; …)
				Inside
				for (int i = 1;...;...)

38	KeywordSpace (DI)	Using a space after a control keyword	Yes for (...	No for(...
39	OperatorSpace (DI)	Using a space after an operator (such as =, +, -, etc.)	Yes a = 5;	No a =5;
40	BraceAfterKeyword (DI)	Using an open brace on the same line after control keywords	Yes if (...) {	No if (...)

7.2.2 Tool

The input to our tool is a lab log file of student submissions and the instructor’s solution. We obtained our logfile from the zyBook autograder zyLab [5]. A similar log file can be obtained from auto-graders, submission systems, or code repositories (such as GitHub, etc.). The output of the tool is a student roster, where each student has a style anomalies score, style anomalies list, and the student's code. The roster is sorted by style anomalies scores in descending order, so likely cheating cases appear at the top of the roster. Table 7.4 shows a sample output student roster.

Table 7.5 shows a C++ code example to explain how the tool detects style anomalies and calculates a style anomalies score for the first student in Table 7.4. First, the instructor provides the tool with a lab log file of students’ submissions, and the instructor's code (the lab solution).

The first row titled *Code* shows the instructor’s code, and student xyz’s code for a C++ lab example.

First, the tool analyzes the instructor's code for the used features from the available 40 features (defined in Table 7.3). If a feature is not used in the instructor's code, the tool ignores the feature and its styles. For example, if an array is not used in the instructor’s code, the tool ignores the Array feature (and its styles). By the same token, if the instructor’s code does not use any variable, the tool ignores all variable-related features (and their styles) such as IniVarDec, LateVarDec, MultiVarDec, etc.

Table 7.4: A student roster example sorted in descending order by style anomalies scores, so higher possible cheating cases are shown at the top of the roster.

Student ID	Style anomalies score	Style anomalies list	Student’s code
xyz	1.8	OpenBrace: Standalone BraceAfterKeyword: No Scanf: Yes	…
abc	1.0	OpenBrace: Standalone BraceAfterKeyword: No	…
…	…	…	…

For the used features in the instructor’s code only, the tool detects the common features used in both the instructor’s code and the student’s code. The second row titled *Instructor and student styles* shows the instructor’s styles and the student’s styles for such features. For brevity, only 4 common features (with their used styles) are shown.

The third row, titled *Style anomalies due to differing from instructor*, shows the 2 style anomalies between the instructor’s styles (which are *Inline*, and *Yes*), and the student’s styles (which are *Standalone*, and *No*) for the 2 features *OpenBrace*, and *BraceAfterKeyword*, respectively.

Table 7.5: Example of C++ instructor's code and student's code with different styles to show how the tool detects style anomalies and calculates a style anomalies score.

	Instructor	**Student xyz**

Code	int main () {
	  int x;
	
	  cin >> x;
	  if (x == 0) {
	    cout << "zero";
	  }
	  else {
	    cout << "non-zero";
	  }
	}
		int main ()
		{
		  int x;
		
		  scanf(x);
		  if (x == 0)
		    cout << "zero";
		  else
		    cout << "non-zero";
		}

Instructor and student styles	● OpenBrace: Inline ● BraceAfterKeyword: Yes ● KeywordSpace: Yes ● OperatorSpace: Yes	● OpenBrace: Standalone ● BraceAfterKeyword: No ● KeywordSpace: Yes ● OperatorSpace: Yes
Style anomalies due to differing from instructor		● OpenBrace: Standalone ● BraceAfterKeyword: No
Styles disallowed by instructor	● Scanf: No ● Printf: No	
Style anomalies due to disallowed styles		● Scanf: Yes
Total style anomalies (weight)		● OpenBrace: Standalone (0.5) ● BraceAfterKeyword: No (0.5) ● Scanf: Yes (0.8)
Style anomalies score		0.5 + 0.5 + 0.8 = 1.8

The tool provides the instructor with the allowed/disallowed list as shown in Table 7.6 (only 4 out of 40 features with their styles for brevity). By default, all the features with their styles are selected in the allowed/disallowed list, and the instructor needs to deselect the styles they want to allow in students' code. The allowed/disallowed list provides instructors with the following advantages:

- For use-only style features: Automatically flags the use style of a feature in students' code that is not used in the instructor's code. For example, if the instructor's code does not use an array, the tool will flag the use of the array in students' code. If the instructor wants to allow students to use arrays, the instructor needs to de-select the array use style from the allowed/disallowed list.

- For use-version style features: Automatically flags the opposite corresponding use-version style, of the style used in the instructor's code, for a feature**.** For example, if the instructor's code uses the prefix style for the increment operator, the tool automatically flags the postfix usage version style without any intervention from the instructor. If the instructor wants to allow students to use both prefix and postfix styles, the instructor needs to deselect the increment operator feature from the allowed/disallowed list.
- Allows instructors to customize the weights for the styles in the allowed/disallowed list if the instructors wish to highlight indicative anomalies. For example, an instructor might want to have a weight of 1.0 (instead of the default of 0.5) for using an array in students' code before array has been introduced in the class.

Under the hood, the allowed/disallowed list is the list of the 40 features in Table 7.3, but their styles are the styles in *Style 2* column in Table 7.3. Notice that for a use-version style feature, our approach automatically flags only a student style that: (1) Has a style in *Style 2* column in Table 7.3, and (2) has the opposite corresponding style, of the style used in the instructor's code, for the same feature.

Table 7.6: The allowed/disallowed list as it appears in the tool (only 4 out of 40 are shown for brevity). Notice that scanf and printf disallowed use styles are selected (because all use styles in the allowed/disallowed list are selected by default), and the instructor changed the weights of their corresponding features to 0.8 instead of their default weights of 0.5 as shown in Table 7.3.

Weight	Selected	Disallowed use style
0.5	☑	Array
0.5	☑	Pointer
0.8	☑	scanf
0.8	☑	printf
…	…	…

The 40 styles with their 40 features are formatted and presented by the tool as choices (checkboxes) in a user-friendly way. The allowed/disallowed list is based on our experience and observations of what an instructor might wish to flag in their C++ code since Table 7.3 has 80 styles, but we only picked 40 from them.

In Table 7.5, the instructor's solution did not use the scanf, and printf. However, the allowed/disallowed list disallows the use of scanf, and printf in the student's code as shown in Table 7.6 (notice that the instructor also changed the weight of scanf and printf from the default weight of 0.5 in Table 7.3 to 0.8 for the 2 features). The fourth row titled *Styles disallowed by instructor* shows the disallowed styles *(Scanf: No*, and *Printf: No)* are selected.

The tool analyzes the student's code for the disallowed styles and detects *Scanf: Yes* style in the student's code (because the student's code contains scanf, but not printf) as shown in the fifth row labeled *Style anomalies due to disallowed styles.* The tool adds the 1 detected style anomaly (Scanf: Yes) to the 2 previously identified style anomalies (OpenBrace: Standalone BraceAfterKeyword: No) to form a total of 4 style anomalies as shown in the sixth row titled *Total style anomalies (weight).*

To calculate the student style anomalies score, the tool adds the weights of each feature in the student's code (*OpenBrace, BraceAfterKeyword, and Scanf*) in the row titled *Total style anomalies (weight)*. The row titled *Style anomalies score* shows the result of 0.5 + 0.5 + 0.8, which is 1.8 since the first 2 features (*OpenBrace, and BraceAfterKeyword*) have a default weight of 0.5, but the instructor changed the default weight of *Scanf* (via the allowed/disallowed list) to 0.8.

We developed our tool as a web tool using Python 3.7, and the Python library Common Gateway Interface (CGI) [6]. The tool code size is about 1.5 K lines of code. The tool uses string matching, and regular expression to detect used features and styles. The tool uses multiple Python dictionary data structures to profile the instructor's style (including the allowed/disallowed list), the student's style, the 40 features with their styles, and weights. The tool runs the dictionaries against each other to discover style anomalies and calculate their score accordingly. The tool took less than 1 minute to produce the student roster for one of our classes of 83 students, which is the same class that we used in the experiments section below.

7.2.3 Experiments

We ran experiments to answer the following research question: What is the accuracy (in terms of sensitivity, and specificity) of our tool in detecting style anomalies?

We used a quantitative approach; experimental design, and method; probability sampling; statistical analysis (using sensitivity, and specificity); and primary data to conduct the experiments. Our experiments used a zyBooks lab logfile for a 83-student C++ CS1 class in Fall 2021 at a large research university.

We manually examined the instructor's code and recorded the used styles for 5 features that we found based on your judgment were most commonly the source of style anomalies. The following list shows the instructor's style in parentheses for the 5 features:

- Array (No)
- IncrementOp (Prefix)
- OpenBrace (Inline)
- KeywordSpace (Yes)
- OperatorSpace (Yes)

Then we manually examined the 83 students and recorded their styles for the 5 features. We picked 30 students, where 15 students have the same styles as the instructor's styles in the 5 features (which we call the normal style group), and 15 students that have different styles compared with the instructor's styles (which we call the abnormal style group) in the 5 code features. The abnormal style group has at least 5 instances of style anomalies in each feature. The following list shows the student style anomalies in parentheses for the 5 features of the abnormal style group:

- Array (Yes)

- IncrementOp (Postfix)
- OpenBrace (Standalone)
- KeywordSpace (No)
- OperatorSpace (No)

We chose 30 students in the sample, 15 students for the normal style group, and 15 students for the abnormal style group, to have a representative balanced sample in our test data.

For each student in both style groups (normal, and abnormal), we manually examined and recorded the results of comparing the instructor's style with the student's style for each of the 5 features. We recorded 'Yes' for style anomaly, and 'No' for no style anomaly based on our visual determination. Then, we ran the tool on both style groups and recorded the results for each student for each of the 5 features. If the tool showed a style anomaly for the student in the student roster, we recorded 'Yes' for style anomaly, otherwise we recorded 'No' for no style anomaly. Table 7.7 showed the manual, and tool results for the abnormal style group for the 5 features. The result for the normal style group is not shown for brevity.

The results show that our tool correctly detected the occurrence of style anomalies in the abnormal style group, and the non-occurrence of style anomalies in the normal style group for the 5 features. We built the confusion matrix and calculated the sensitivity and specificity measurements. Our tool neared 100% accuracy using the sensitivity, and specificity metrics.

Table 7.7: 15 students with style anomalies in 5 features. In X / Y, X stands for the manual results and Y for the tool results for the same feature. A Yes / Yes means both the manual and the tool results detected the same style anomalies for the same feature. The last raw titled Total shows the total number of instances detected by both manual, and tool results for each feature, which are 9 instances for Array, 10 instances for IncrementOp, 8 instances for OpenBrace, 9 instances for KeywordSpace, and 8 instances for OperatorSpace.

Student ID	**Feature: Array**	**Feature: IncrementOp**	**Feature: OpenBrace**	**Feature: KeywordSpace**	**Feature: OperatorSpace**
1	Yes / Yes	Yes / Yes		Yes / Yes	Yes / Yes
2	Yes / Yes				Yes / Yes
3	Yes / Yes				
4		Yes / Yes	Yes / Yes	Yes / Yes	
5	Yes / Yes		Yes / Yes		
6		Yes / Yes	Yes / Yes	Yes / Yes	
7	Yes / Yes	Yes / Yes		Yes / Yes	Yes / Yes
8	Yes / Yes	Yes / Yes			
9	Yes / Yes	Yes / Yes	Yes / Yes		Yes / Yes
10			Yes / Yes		Yes / Yes
11	Yes / Yes	Yes / Yes		Yes / Yes	Yes / Yes
12		Yes / Yes		Yes / Yes	Yes / Yes
13	Yes / Yes	Yes / Yes	Yes / Yes	Yes / Yes	
14			Yes / Yes	Yes / Yes	Yes / Yes
15		Yes / Yes	Yes / Yes	Yes / Yes	
Total	**9 / 9**	**10 / 10**	**8 / 8**	**9 / 9**	**8 / 8**

7.3 Discussion

We found the tool helps to detect possible cheating in large, in terms of the number of students, C++ programming courses in a short time. For example, the tool provides a student roster (shown in Table 7.4) that is sorted in descending order by the style anomalies scores. Such a roster layout makes it easy for instructors to view first likely cheating cases.

Table 7.8 shows the instructor C++ code for 1 lab titled “Adjust list by normalizing - functions” for the same class that we used in the experiments section. The instructor’s styles are shown for comparison with the students’ styles in Table 7.9. Only 13 styles that differ from students’ styles (in Table 7.9) are shown in the *Instructor's styles* column for brevity.

Table 7.9 shows 3 real students reported by the tool as having the highest style anomalies scores in our C++ class. There are more students with style anomalies in our class, but we only picked the 3 students with the highest anomalies score for brevity. The first column titled *Instructor’s styles* shows the instructor styles from the previous table to just make it visually easier to compare the student’s styles with the instructor’s styles in the same table. The second column titled Student *style anomalies (with the score at the top)* shows the style anomalies in the student’s code. The third column titled *Student’s code* shows the student’s code.

The first student in Table 7.9, has a style anomalies score of 4.2 and has 10 anomaly styles compared to the instructor's styles. The anomaly styles are: Using array bracket for indexing, not using camel case, using initialized variable declaration, using late variable declaration, using postfix increment operator, using compound assignment, using standalone brace, using inside for-loop counter variable declaration, using keyword without space, and using no open curly after keywords.

Table 7.8: A real C++ instructor's code for the lab “Adjust list by normalizing - functions”. Notice that the instructor’s styles show only 13 styles that differ from the students’ styles (in Table 7.9) for brevity.

Instructor’s styles	Instructor’s code

Array: No
CamelCase: Yes
IniVarDec: No
LateVarDec: No
MultiVarDec: No
IncrementOp: Prefix
EmptyLines: Yes
CompAssign: No
OpenBrace: Inline
ForLoopCounterDec: Outside
KeywordSpace: Yes
OperatorSpace: Yes
BraceAfterKeyword: Yes
	int GetMinimumInt(const vector<int> &listInts) {
	   unsigned int i;
	   int minInt;
	
	   minInt = listInts.at(0);
	   for (i = 0; i < listInts.size(); ++i) {
	      if (listInts.at(i) < minInt) {
	         minInt = listInts.at(i);
	      }
	   }
	
	   return minInt;
	}
	
	int main() {
	   vector<int> userValues;
	   int numValues;
	   int i;
	   int currValue;
	   int minValue;
	
	   // Get user's values
	   cin >> numValues;
	   for (i = 0; i < numValues; ++i) {
	      cin >> currValue;
	      userValues.push_back(currValue);
	   }
	
	   // Find the minimum value, subtract while outputting
	   minValue = GetMinimumInt(userValues);
	   for (i = 0; i < numValues; ++i) {
	      cout << userValues.at(i) - minValue << " ";
	   }
	   cout << endl;
	
	   return 0;
	}

Table 7.9: 3 real students with the highest style anomalies scores in one of our C++ classes.

Instructor’s styles	Student style anomalies (weight at the top)	Student’s code

Array: No
CamelCase: Yes
IniVarDec: No
LateVarDec: No
IncrementOp: Prefix
CompAssign: No
OpenBrace: Inline
ForLoopCounterDec: Outside
KeywordSpace: Yes
BraceAfterKeyword: Yes
	4.2
	Array: Yes
	CamelCase: No
	IniVarDec: Yes
	LateVarDec: Yes
	IncrementOp: Postfix
	CompAssign: Yes
	OpenBrace: Standalone
	ForLoopCounterDec: Inside
	KeywordSpace: No
	BraceAfterKeyword: No
		…
		int GetMinimumInt(const vector<int> &listInts)
		{
		int min = listInts[0];
		for(int i = 1; i < listInts.size(); i++)
		{
		if (listInts[i] < min){
		min = listInts[i];
		}
		}
		return min;
		}
		
		int main() {
		vector<int> intList;
		int elements;
		cin >> elements;
		int iteratingInput;
		for (int i = 0; i < elements; i++)
		{
		cin >> iteratingInput;
		intList.push_back(iteratingInput);
		}
		int min = GetMinimumInt(intList);
		
		for (int i = 0; i < elements; i++)
		{
		intList[i] -= min;
		}
		for(int i = 0; i < elements; i++)
		{
		cout << intList[i] << " ";
		}
		cout << endl;
		return 0;
		}

CamelCase: Yes
IniVarDec: No
LateVarDec: No
IncrementOp: Prefix
CompAssign: No
OpenBrace: Inline
ForLoopCounterDec: Outside
KeywordSpace: Yes
BraceAfterKeyword: Yes
	4.1
	
	CamelCase: No
	IniVarDec: Yes
	LateVarDec: Yes
	IncrementOp: Postfix
	CompAssign: Yes
	OpenBrace: Standalone
	ForLoopCounterDec: Inside
	KeywordSpace: No
	BraceAfterKeyword: No
		…
		int GetMinimumInt(const vector<int>& nums)
		{
		int small = nums.at(0);
		
		for(int i = 1; i < (signed)nums.size(); i++)
		{
		if(small > nums.at(i))
		{
		small = nums.at(i);
		}
		}

		return small;
		}
		
		int main() {
		
		int size;
		
		vector<int> nums;
		
		cin >> size;
		
		for(int i = 0; i < size; i++)
		{
		int num;
		
		cin >> num;
		
		nums.push_back(num);
		}
		
		int small = GetMinimumInt(nums);
		
		for(int i = 0; i < (signed)nums.size(); i++)
		{
		nums.at(i) -= small;
		cout << nums.at(i) << " ";
		}
		
		cout << endl;
		
		return 0;
		}

Array: No
IniVarDec: No
LateVarDec: No
MultiVarDec: No
IncrementOp: Prefix
EmptyLines: Yes
ForLoopCounterDec: Outside
KeywordSpace: Yes
OperatorSpace: Yes
BraceAfterKeyword: Yes
	4.0
	Array: Yes
	IniVarDec: Yes
	LateVarDec: Yes
	MultiVarDec: Yes
	IncrementOp: Postfix
	EmptyLines: No
	ForLoopCounterDec: Inside
	KeywordSpace: No
	OperatorSpace: No
	BraceAfterKeyword: No
		…
		int GetMinimumInt(vector<int> listInts) {
		int minInt = listInts[0];
		for(unsigned i=1; i<listInts.size(); i++){
		if(minInt>listInts[i])
		minInt = listInts[i];
		}
		return minInt;
		}
		
		int main() {
		/* Type your code here */
		int n, numInt;
		cin >> n;
		vector<int> listInts;
		for (unsigned i = 0; i < n; i++) {
		cin >> numInt;
		listInts.push_back(numInt);
		}
		int minInt = GetMinimumInt(listInts);
		for (unsigned i = 0; i < n; i++) {...

The second student has a style anomalies score of 4.1 and has 9 anomaly styles compared to the instructor's styles. The anomaly styles are: Non-camel case, initialized variable declaration, late variable declaration, postfix increment operator, compound assignment, standalone open brace, inside for-loop counter variable declaration, keyword without space, and no open curly after keywords.

The third student has a style anomalies score of 4.0 and has 10 anomaly styles compared to the instructor's styles. The anomaly styles are: Using array, initialized variable declaration, late variable declaration, multiple variable declaration, postfix increment operator, empty lines in code, inside for-loop counter variable declaration, keyword without space, no space around operator, and no open curly after keywords.

The 3 students' examples show obvious possible cheating cases. For example, why did a student write code that has 10 (or even 9) different styles from the instructor’s styles? The 3 cases seem to present blatant cheating cases after we confirmed them manually. The tool was helpful to detect these cases in a couple of minutes in a class of 83 students.

In the future, we plan to make the tool more flexible by allowing instructors to customize the tool’s features, weights, styles, mapping functions, and thresholds. We also plan to make the tool work when the instruction’s code is not available, by extracting the dominant styles among students in class.

7.4 Conclusion

Although the code similarity detection approach is widely used among instructors in programming courses, such an approach is not effective in detecting unique solutions that are not common among students in class. Thus, complementary approaches are needed to complement the similarity approach. We presented a class code style anomaly approach to detect possible cheating based on class code style anomalies. Our approach finds style anomalies between the class code style, and the student's code style, and reports the differences as possible cheating. We implemented our approach in a tool that automates the detection of style anomalies between the class code styles, and student code styles. We conducted experiments to assess the accuracy of our tool in detecting style anomalies. Our tool accuracy neared 100% in the sensitivity, and specificity measurements. We plan to make our tool available to the CS community as a free web tool.

8. Student Programming Behavior Tool

8.1 Techniques and Tools to Detect and Prevent Struggle and Cheating

We developed multiple techniques and tools to detect and prevent struggle and cheating. The techniques and tools are :

- Drastic change detector
- Code style variation detector
- Class code style variation detector
- Hardcoding detector
- Code similarity checker
- Code progression highlighter
- Sprint detector

Figure 8.1 shows the overall design of how the techniques/tools interact. For short, we call collectively the techniques and tools the student programming behavior tool.

The tool takes a lab log file with a simple format as described in Table 8.1. The file format is the default for zyBooks, and hence our tool is immediately usable for 2,000+ courses. However, any auto-grader log files can be auto-converted to such format so their files can be imported to our tool as well.

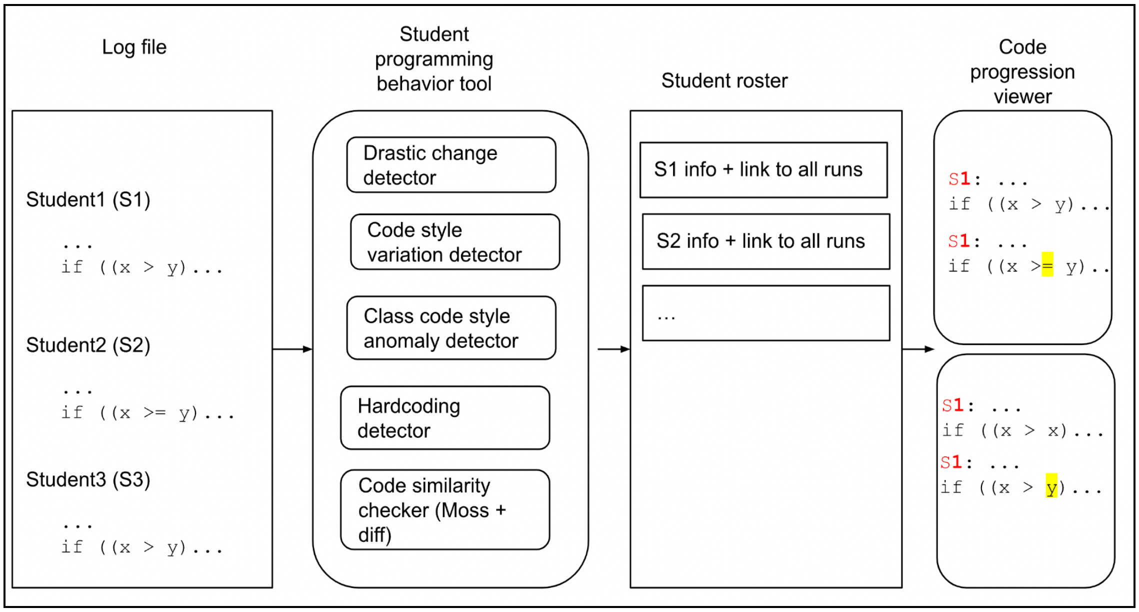


Figure 8.1: Overview of the student programing behavior tool.

Table 8.1: A sample lab log file format.

Student ID	Timestamp	Code link	Run	Score	Max score
1001	1/1/20 1:1:00	URL1	Dev		
1001	1/1/20 1:2:15	URL2	Sub	4	10
1001	1/1/20 1:9:10	URL3	Sub	10	10
1212	1/1/20 9:2:1	URL4	Dev		
1212	1/1/20 9:3:5	URL5	Dev		
…	…	…	…	…	…

An instructor starts using the tool by first uploading a lab log file. The tool then presents a class statistics table and a student roster as shown in Table 8.2 and Table 8.3. For each student (row) in the roster, the instructor can click the link in the Code progression view column to start the code progression viewer tool as shown in Table 8.4.

The code progression viewer allows the instructor to gain insight into students coding process.

Table 8.2: Class statistics.

Student	# students	Avg score	Avg time (min.)	Avg runs	Avg code size (LOC)
All students	21	8.1	10.9	14.1	16.5
Top quartile	4	10.0	7.0	12.0	16.2

Table 8.3: A sample class student roster with statistics (the numbers are arbitrary to just show what a sample roster looks like).

ID	Code progre. view	Score	Time spent (min)	# of sub.	Code size	Sprint	Hard coding	Code similarity	Style anomaly	Style variation	Drastic change
10	Link	10	11	6	15	0.1	1.0	0.3	0.6	0.7	0.6
20	Link	0	1	2	10	0.5	0.0	0.4	0.9	0.8	0.2
…	…	…	…	…	…	…	…	…	…	…	…

The code progression viewer allows the instructor to view code changes between 2 consecutive code runs in different colors. For example, added characters are shown in green, changed characters in yellow, and deleted characters in red (only the yellow color is shown in Table 8.4). The tool also provides important information for each code run such as code run type (dev or sub), score, diff (added, changed, and deleted character statistics), timestamp, estimated time for each code run and the total code run time since the first code run. If there is a time gap of more than 10 minutes between 2 consecutive code runs, the tool assumes the student stepped out. The tool also assumes a rate of 1 character per 5 seconds is used by students when coding. Thus, the tool uses that rate with the time difference between the 2 code runs to estimate the code time that was used for coding from the stepped out time.

Table 8.4: A sample student code progression view.

Run #	Code	Run type	Score	Min. since previous	Total min	Timestamp	Ins(+) Change(^) Del(-)
...	...	...	...	...	3	...	...

4	if (x > y)
	   cout << x;
		Dev		1	4	1/1/20
						9:20:03	

5	if ((x > y) && (x > z))
	   cout << x;
	else if ((y > x) && (y > z))
	   cout << y;
	else
	   cout << z;
		Sub	4	0.5	4.5	1/1/20
						9:21:15
							+55

6	if ((x >= y) && (x > z))
	   cout << x;
	else if ((y > x) && (y > z))
	   cout << y;
	else
	   cout << z;
		Sub	10	6.9	11.4	1/1/20
						9:30:10
							+1

The instructor can sort the roster by any column (by clicking the column header to sort ascending or descending), so the instructor can focus on a group of students based on a specific concern. For example, an instructor can investigate possible struggling students by sorting the roster by score and time. The instructor can also investigate the cause of a student struggle by clicking on the code progression view link for the struggling student to view their code progressions and examine what causes the struggle.

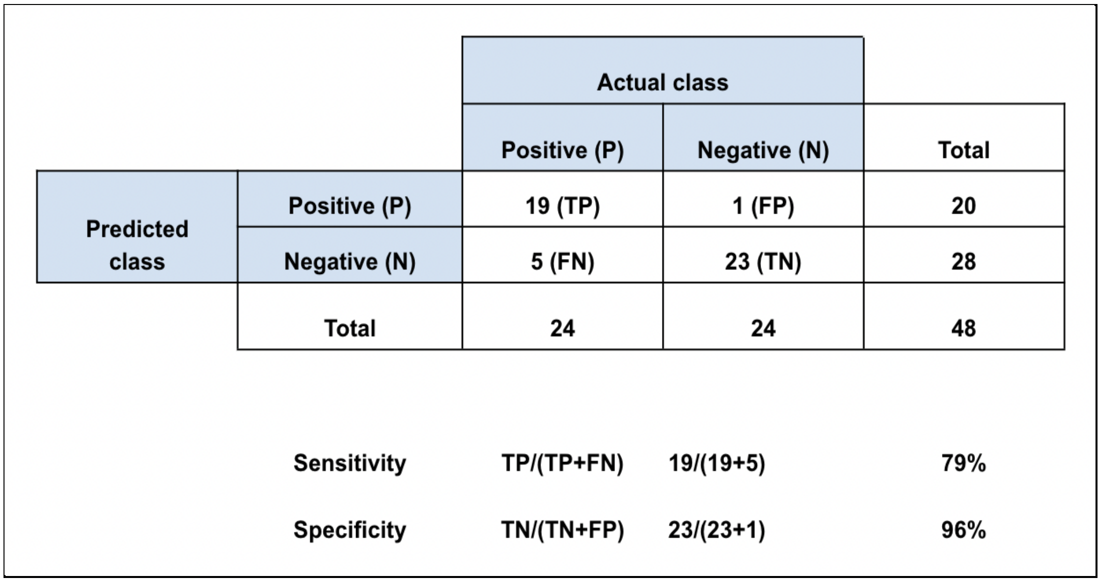
The roster also shows the results of the tool's different detectors. The Sprint column presents the sprint detector results. The sprint detector assumes a student needs more than 5 minutes to complete a lab. If a student completes a lab in less than 5 minutes and with a full score, the tool assumes it is a sprint activity. A sprint might indicate a suspicious activity by a student (such as cheating), and might be worth investigating further by the instructor. The sprint value is in the range of 0 and 1 (including 0 and 1). 0 means low likelihood of sprinting, and 1 means high likelihood of sprinting.

The hardcoding detector allows instructors to detect students who hardcode the auto-grader testcases inside their code to trick the auto-grader. The hardcoding detector examines student code for the auto-grader input and output testcases inside student code. The hardcode detection values are under the Hardcoding column. The values are in the range of 0 and 1. 0 means low likelihood of hardcoding, and 1 means high likelihood of hardcoding. Our hardcoding detector averaged 79% in sensitivity and 96% in specificity. Table 8.5 shows the confusion matrix for the hardcode detector.

The code similarity checker presents both the Moss and the Diff (the Python Diff library) similarity values for a selected student against all other students at or above a similarity ratio. The checker allows the instructor to specify a similarity ratio (a value between 0 and 1), such as 0.9. The checker then uses the similarity ratio to find all students with the similarity ratio compared to a specific student selected in the roster.

Moss ignores comments, but supports detecting moved code and renamed identifiers when detecting similarity. However, Diff examines comments, but does not support moved code and renamed identifiers when detecting similarity. Both checkers complement each other and give instructors a better picture about student code similarity.

Table 8.5: Hardcode detector accuracy.



		Actual class		
		Positive (P)	Negative (N)	Total
Predicted class	Positive (P)	19 (TP)	1 (FP)	20
	Negative (N)	5 (FN)	23 (TN)	28
	Total	24	24	48
	Sensitivity	TP/(TP+FN)	19/(19+5)	79%
	Specificity	TN/(TN+FP)	23/(23+1)	96%

The class code style anomaly detector examines the student code style against the class code style to find code style anomalies. The detector uses 40 code features with their styles to profile the instructor code and student code. The detector compares the captured 2 profiles (for instructor code and student code for the same lab) and finds discrepancies in using styles for the same feature. For example, if the instructor uses prefix style for the increment operator (such as ++i), but the student uses the postfix style for the increment operator (such as i++), the tool detects that as code style anomaly. The detector also detects unused or not introduced yet code styles in class as anomalies. For example, if the instructor has not yet introduced arrays or functions in class, but a student used arrays or functions in their code, the tool will detect that as anomalies. The detector uses a value in the range of 0 and 1 to indicate the likelihood of the anomaly in the roster. 0 means an unlikely anomaly, where 1 means a highly likely anomaly.

The style variation detector examines all labs for a single student and discovers code style variation. A style variation across labs for a student might indicate a potential concern of cheating. According to research [6.1], code style seems to persist as a personal preference, which can be used to identify the author of code. The detector uses 15 code features to profile each different lab for a single student. For example, if one lab uses the prefix style for the increment operator (such as ++i), but the same student uses the postfix style for the increment operator (such as i++) for another lab, the tool detects that as code style variation. The detector uses a value in the range of 0 and 1 to indicate the likelihood of the style variation in the roster for a student across their labs. 0 means an unlikely style variation, where 1 means a highly likely style variation.

The drastic code change detector detects unrealistic big changes in student code progressions for the same student in the same lab. For example, for one student in one lab, if the first code size (given that it has more than 15 lines of code) is bigger than 50% of the first-highest code (given that it has a full score), the detector considers a drastic code change occurred in the first code. We call that first code an initial leap. We call a drastic change anywhere in code progression, except in the first code, a gave up. The detector detects a gave up using a Python diff library. If the diff of a code pair is above 20%, the detector concludes a drastic change occurred with the diff ratio as a likelihood for the drastic change. If a gave up happened again, but with a different solution, we call that gave up a hopping solution. The roster shows the diff ratio with the code run number. For example, 0.7@ 6 means a drastic change of 0.7 occurred at code run # 6.

We built the tool using HTML, CSS, JavaScript, Python 3.7, and Python CGI web framework. The tool is split into 7 modules and has more than 12K lines of code. The tool used python dictionary data structures to store log file data, code profiles, features, etc. The tool also used regular expressions and built-in Python string matching functions to profile code and extract features. Figure 8.2 shows a block diagram of the 7 modules.

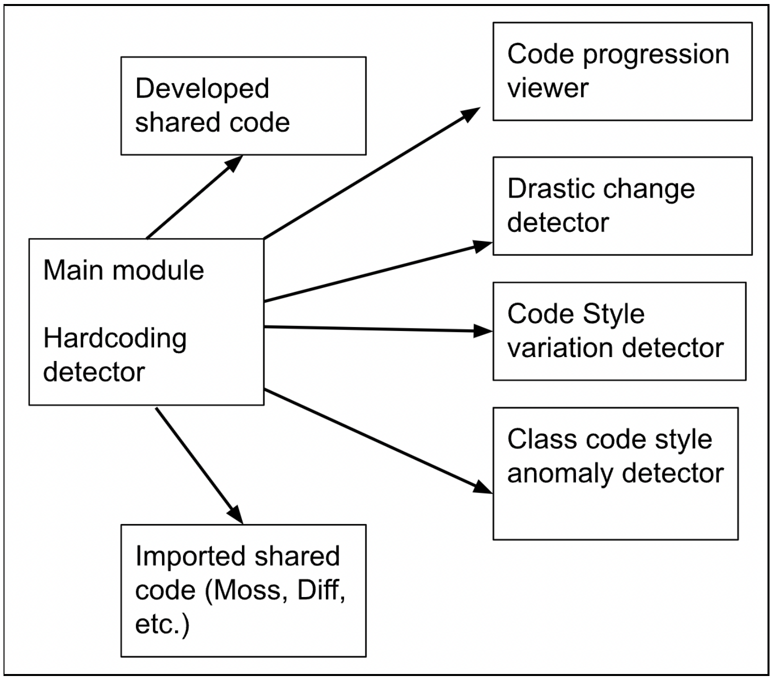

Developed shared code
Code progression viewer
Drastic change detector
Main module
Hardcoding detector
Code Style variation detector
Class code style anomaly detector
Imported shared code (Moss, Diff, etc.)

Figure 8.2 A block diagram of the 7 modules.

The tool speed depends heavily on the following factors:

- Number of students who submitted labs
- Number of code runs per student
- Code run size

- And other factors outside the scope of this work such as the network bandwidth and the available computing resources (mainly the CPU and main memory) on the instructor client computer and the tool web server.

Since the above factors differ between labs, we give the following rough estimate of the tool speed based on the following rough averages for an example lab file: The tool takes on average 3 seconds per student, with an average of 100 students, an average of 10 code runs per student, and an average of 20 lines of code size. The tool allows an instructor to upload a range of students instead of all students in the log file. We found that uploading a smaller number of students (such as 50) at a time can cut the time by more than half. Most of the time seems to be consumed by uploading the file and performing code similarity checks of all student code against each other. In the future, using threading and parallel programming may help to improve the speed.. Table 8.4 summarizes some tool statistics.

Table 8.6: Tool statistics summary.

Language	Web framework	Tool files	Tool code size	Tool speed average
Python 3.7 JavaScript HTML CSS	Python CGI	7 files	12 K	● 3 seconds per student assuming averages of 100 students, 10 runs per student and 20 LOC run size per student. ● Uploading 50 students at a time can cut the time by half

8.2 Contributions

The dissertation's contribution was in the development of techniques and tools to help instructors to detect and prevent student struggle and cheating in CS1. One aspect of the work is to automatically detect struggling students, and list common programming errors that cause struggle. Instructors can help struggling students through early intervention such as adjusting instruction, and content accordingly to reduce struggle.

Another aspect of the work is to automatically detect cheating even in unique code submissions that is not detected by code similarity detection tools. The work presented multiple automated techniques and tools that we developed to detect cheating. The techniques and tools used multiple potential indicators of cheating such as drastic change in code progression in a student's code runs, code style variation across a single student's programs, code style anomaly in a student program compared to class style, hardcode detection, code similarity checker, sprint detection, and code progression viewer. The tools averaged nearly 100% accuracy in a short time. We plan to make the tools available for the CS community as free web tools to maximize the impact of this work.

Instructors might present the tools in class in front of students to help reduce cheating. Presenting the tools in class would make students aware that their coding activities and code progressions are monitored by instructors, and instructors have access to student effort history, and not just the final code submissions. Such an approach may help to prevent cheating.

References

Chapter 1 References

[1] Stuart Zweben and Betsy Bizot. 2018. “2017 CRA Taulbee survey,” Computing Research News 30, 5 (2018), 1–47.

[2] Tracy Camp, W. Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambrusch, Ellen Walker, and Stuart Zweben. 2017. “Generation CS: the growth of computer science,” ACM Inroads 8, 2 (2017), 44–50.

[3] Christopher Watson and Frederick WB Li. 2014. “Failure rates in introductory programming revisited,” In Proceedings of the 2014 conference on Innovation & technology in computer science education, 39–44.

[4] Qusay H. Mahmoud, Wlodek Dobosiewicz, and David Swayne. 2004. Making computer programming fun and accessible. Computer 37, 2 (2004), 106–108.

[5] Theresa Beaubouef and John Mason. 2005. Why the high attrition rate for computer science students: some thoughts and observations. ACM SIGCSE Bulletin 37, 2 (2005), 103–106.

[6] James C. Spohrer, Elliot Soloway, and Edgar Pope. 1985. A goal/plan analysis of buggy Pascal programs. Human–Computer Interaction 1, 2 (1985), 163–207.

[7] Gavriel Yarmish and Danny Kopec. 2007. Revisiting novice programmer errors. ACM SIGCSE Bulletin 39, 2 (2007), 131–137.

[8] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. All syntax errors are not equal. In Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education, 75–80.

[9] Brett A. Becker. 2016. An effective approach to enhancing compiler error messages. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education, 126–131.

[10] Amjad Altadmri and Neil CC Brown. 2015. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In Proceedings of the 46th ACM technical symposium on computer science education, 522–527.

[11] Basma S. Alqadi and Jonathan I. Maletic. 2017. An empirical study of debugging patterns among novices programmers. In Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education, 15–20.

[12] Matthew Hertz and Maria Jump. 2013. Trace-based teaching in early programming courses. In Proceeding of the 44th ACM technical symposium on Computer science education, 561–566.

[13] Devon O'Dell. 2017. "The Debugging Mind-Set," Communications of the ACM 60, no. 6 (2017): 40–45. https://doi.org/10.1145/3052939.

[14] Cay S. Horstmann and Timothy Budd. 2004. Big C++. 2nd ed. Hoboken, N.J: Wiley.

[15] Steve Oualline. 2003. How not to program in C++: 111 broken programs and 3 working ones, or why does 2+ 2. No Starch Press.

[16] David Ginat and Ronit Shmalo. 2013. Constructive use of errors in teaching CS1. In Proceeding of the 44th ACM technical symposium on Computer science education, 353–358.

[17] Ibrahim Albluwi. 2019. Plagiarism in programming assessments: A systematic review. ACM Transactions on Computing Education (TOCE) 20, 1 (2019), 1–28.

[18] Jess Bidgood and Jeremy B. Merrill. 2017. As computer coding classes swell, so does cheating. New York Times 6, (2017).

[19] Breanna Devore-McDonald and Emery D. Berger. 2020. Mossad: Defeating software plagiarism detection. Proceedings of the ACM on Programming Languages 4, OOPSLA (2020), 1–28.

[20] David J. Malan, Brian Yu, and Doug Lloyd. 2020. Teaching academic honesty in CS50. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education, 282–288.

[21] Aiken, Alex, "Moss: A System for Detecting Software Similarity," https://theory.stanford.edu/~aiken/moss/ (accessed May 2, 2022).

[22] Xin Chen, Brent Francia, Ming Li, Brian Mckinnon, and Amit Seker. 2004. Shared information and program plagiarism detection. IEEE Transactions on Information Theory 50, 7 (2004), 1545–1551.

[23] Christian Collberg, Ginger Myles, and Michael Stepp. 2004. Cheating cheating detectors. TR04-05. Tucson, Department of Computer Science, University of Arizona 7, (2004).

[24] Sathiamoorthy Manoharan and Ulrich Speidel. 2020. Contract cheating in computer science: A case study. In 2020 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE), IEEE, 91–98.

Chapter 2 References

[1] zyBooks. https://www.zybooks.com/ (accessed May, 2022).

[2] Nabeel Alzahrani, Frank Vahid, Alex Daniel Edgcomb, Roman Lysecky, and Susan Lysecky. 2018. An analysis of common errors leading to excessive student struggle on homework problems in an introductory programming course. In 2018 ASEE Annual Conference & Exposition.

Chapter 3 References

[1] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. “Identifying and correcting Java programming errors for introductory computer science students,” ACM SIGCSE Bulletin 35, 1 (2003), 153–156.

[2] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. “All syntax errors are not equal” In Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education, 75–80.

[3] Andrew J. Ko and Brad A. Myers. 2005. “A framework and methodology for studying the causes of software errors in programming systems,” Journal of Visual Languages & Computing 16, 1–2 (2005), 41–84.

[4] Amjad Altadmri and Neil CC Brown. 2015. “37 million compilations: Investigating novice programming mistakes in large-scale student data,” In Proceedings of the 46th ACM Technical Symposium on Computer Science Education, 522–527.

[5] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. “Common logic errors made by novice programmers,” In Proceedings of the 20th Australasian Computing Education Conference, 83–89.

[6] Linda Grandell, Mia Peltomäki, and Tapio Salakoski. 2005. “High school programming—a beyond- syntax analysis of novice programmers’ difficulties,” In

Proceedings of the Koli Calling 2005 Conference on Computer Science Education, 17–24.

[7] Greg C. Lee and Jackie C. Wu. 1999. “Debug It: A debugging practicing system,” Computers & Education 32, 2 (1999), 165–179.

[8] Vassilios Efopoulos, Vassilios Dagdilelis, Georgios Evangelidis, and Maya Satratzemi. 2005. “WIPE: a programming environment for novices,” In Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education, 113–117.

[9] Nabeel Alzahrani, Frank Vahid, A. Edgcomb, R. Lysecky, and Susan Lysecky. 2018. “An Analysis of Common Errors Leading to Excessive Student Struggle on Homework Problems in an Introductory Programming Course,” In Proceedings of ASEE Annual Conference.

[10] Beth Simon, Sue Fitzgerald, Renée McCauley, Susan Haller, John Hamer, Brian Hanks, Michael T. Helmick, Jan Erik Moström, Judy Sheard, and Lynda Thomas. 2007. “Debugging assistance for novices: a video repository,” ACM SIGCSE Bulletin 39, 4 (2007), 137–151.

[11] Renée C. Bryce, Alison Cooley, Amy Hansen, and Nare Hayrapetyan. 2010. “A one year empirical study of student programming bugs,” In 2010 IEEE Frontiers in Education Conference (FIE), IEEE, F1G-1-F1G-7.

[12] Sandy Garner, Patricia Haden, and Anthony Robins. 2005. “My program is correct but it doesn’t run: a preliminary investigation of novice programmers’ problems,” In Proceedings of the 7th Australasian conference on Computing education-Volume 42, 173–180.

[13] J. C. Spohrer, E. Pope, M. Lipman, W. Sack, S. Freiman, D. Littman, W. L. Johnson, and E. Soloway. 1985. “Bug Catalogue: II, III, IV,” Yale University, Department of Computer Science.

[14] Ricardo Caceffo, Steve Wolfman, Kellogg S. Booth, and Rodolfo Azevedo. 2016. “Developing a computer science concept inventory for introductory programming,” In Proceedings of the 47th ACM Technical Symposium on Computing Science Education, 364–369.

[15] Brian Hanks. 2008. “Problems encountered by novice pair programmers,” Journal on Educational Resources in Computing (JERIC) 7, 4 (2008), 1–13.

[16] Yizhou Qian and James Lehman. 2017. “Students’ misconceptions and other difficulties in introductory programming: A literature review,” ACM Transactions on Computing Education (TOCE) 18, 1 (2017), 1–24.

[17] Anthony Robins, Patricia Haden, and Sandy Garner. 2006. “Problem distributions in a CS1 course,” In Proceedings of the 8th Australasian Conference on Computing Education-Volume 52, 165–173.

[18] Morgan Hall, Keri Laughter, Jessica Brown, Chelynn Day, Christopher Thatcher, and Renee Bryce. 2012. “An empirical study of programming bugs in CS1, CS2, and CS3 homework submissions,” Journal of Computing Sciences in Colleges 28, 2 (2012), 87–94.

[19] Sue Fitzgerald, Gary Lewandowski, Renee McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. “Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers,” Computer Science Education 18, 2 (2008), 93–116.

[20] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. “Debugging: the good, the bad, and the quirky--a qualitative analysis of novices’ strategies,” ACM SIGCSE Bulletin 40, 1 (2008), 163–167.

[21] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. “An analysis of patterns of debugging among novice computer science students,” In Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education, 84–88.

[22] A. Raana, M. A. Azam, M. A. Ghazanfar, A. Javed, Y. Amin, and U. Naeem. 2016. “C++ BUG CUB: Logical Bug Detection for C++ Code,” Nucleus 53, 1 (2016), 56–63.

[23] Nghi Truong, Paul Roe, and Peter Bancroft. 2004. “Static analysis of students’ Java programs,” In Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30, Citeseer, 317–325.

[24] V. Vipindeep and Pankaj Jalote. 2005. “List of common bugs and programming practices to avoid them,” Electronic, March (2005).

[25] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. “Programmers’ build errors: a case study (at google),” In Proceedings of the 36th International Conference on Software Engineering,

724–734.

[26] Alexander Hoem Rosbach. 2013. “Novice difficulties with language constructs,” The University of Bergen.

[27] Ioana Tuugalei Chan Mow. 2012. “Analyses of student programming errors in Java programming courses,” Journal of Emerging Trends in Computing and Information Sciences 3, 5 (2012), 739–749.

[28] Basma S. Alqadi and Jonathan I. Maletic. 2017. “An empirical study of debugging patterns among novices programmers,” In Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education, 15–20.

[29] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. 2014. “Identifying challenging CS1 concepts in a large problem dataset,” In Proceedings of the 45th ACM technical symposium on Computer science education, 695–700.

[30] David Pritchard. 2015. “Frequency distribution of error messages,” In Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools, 1–8.

[31] Toby J. Teorey and Ann R. Ford. 2001. “Practical DeBugging C++,” Prentice Hall Professional Technical Reference.

[32] Draylson Micael de Souza, Michael Ko̎lling, and Ellen Francine Barbosa. 2017. “Most common fixes students use to improve the correctness of their programs,” In 2017 IEEE Frontiers in Education Conference (FIE), IEEE, 1–9.

[33] R. Paul Wiegand, Anthony Bucci, Amruth N. Kumar, Jennifer L. Albert, and Alessio Gaspar. 2016. “A data-driven analysis of informatively hard concepts in introductory programming,” In Proceedings of the 47th ACM Technical Symposium on Computing Science Education, 370–375.

[34] Alireza Ebrahimi. 1994. “Novice programmer errors: Language constructs and plan composition,” International Journal of Human-Computer Studies 41, 4 (1994), 457–480.

[35] Teemu Sirkia ̈and Juha Sorva. 2012. “Exploring programming misconceptions: an analysis of student mistakes in visual program simulation exercises,” In Proceedings of the 12th Koli Calling International Conference on Computing Education Research, 19–28.

[36] James C. Spohrer and Elliot Soloway. 1986. “Novice mistakes: Are the folk wisdoms correct?,” Communications of the ACM 29, 7 (1986), 624–632.

[37] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. “Learning and teaching programming: A review and discussion,” Computer science education 13, 2 (2003), 137–172.

[38] Einari Kurvinen, Niko Hellgren, Erkki Kaila, Mikko-Jussi Laakso, and Tapio Salakoski. 2016. “Programming misconceptions in an introductory level programming course exam,” In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education, 308– 313.

[39] Michael Winikoff. 2014. “Novice programmers’ faults & failures in GOAL programs,” In Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems, 301–308.

[40] Eranki LN Kiran and Kannan M. Moudgalya. 2015. “Evaluation of programming competency using student error patterns,” In 2015 International Conference on Learning and Teaching in Computing and Engineering, IEEE, 34–41.

[41] Neil CC Brown and Amjad Altadmri. 2014. “Investigating novice programming mistakes: Educator beliefs vs. student data,” In Proceedings of the tenth annual conference on International computing education research, 43–50.

[42] Cruz Izu, Amali Weerasinghe, and Cheryl Pope. 2016. “A study of code design skills in novice programmers using the SOLO taxonomy,” In Proceedings of the 2016 ACM Conference on International Computing Education Research, 251–259.

[43] Brian Hanks and Matt Brandt. 2009. “Successful and unsuccessful problem solving approaches of novice programmers,” ACM SIGCSE Bulletin 41, 1 (2009), 24–28.

[44] Nelishia Pillay and Vikash R. Jugoo. 2006. “An analysis of the errors made by novice programmers in a first course in procedural programming in Java,” Preface of the Editors 84, (2006).

[45] Nancy Cunniff, Robert P. Taylor, and John B. Black. 1986. “Does programming language affect the type of conceptual bugs in beginners’ programs? A comparison of FPL and Pascal,” ACM SIGCHI Bulletin 17, 4.

Chapter 4 References

[1] Stephen H. Edwards and Manuel A. Perez-Quinones. 2008. Web-CAT: automatically grading programming assignments. In Proceedings of the 13th annual conference on Innovation and technology in computer science education, 328–328.

[2] zyBooks, www.zybooks.com, August 2020.

[3] Arjun Singh, Sergey Karayev, Kevin Gutowski, and Pieter Abbeel. 2017. Gradescope: a fast, flexible, and fair system for scalable assessment of handwritten work. In Proceedings of the fourth (2017) acm conference on learning@ scale, 81–88.

[4] Mimir, www.mimirhq.com, August 2020.

[5] Vocareum, www.vocareum.com/home/programming-lab, August 2020.

[6] Turing's Craft: CodeLab. https://www.turingscraft.com, August 2020.

[7] Pearson: MyProgrammingLab. www.pearsonmylabandmastering.com, August 2020.

[8] Chelsea Gordon, Roman Lysecky, and Frank Vahid, “The rise of the zyLab program auto-grader in introductory CS courses,” zyBook.com, White Paper, 2021. [Online]. Available: https://docs.google.com/document/d/e/2PACX-1vQYwxlY738_9zFFwOer1kKTNGuJx1Qe3IDW8XHf_OOYbaq9Drf_a9ljCqjcHY9Vv4ryPK423W7F mHwZ/pub

[9] Jon Loeliger and Matthew McCullough. 2012. Version Control with Git: Powerful tools and techniques for collaborative software development. O’Reilly Media, Inc.

[10] Louis Glassy. 2006. Using version control to observe student software development processes. Journal of Computing Sciences in Colleges 21, 3 (2006), 99–106.

[11] Keir Mierle, Kevin Laven, Sam Roweis, and Greg Wilson. 2005. Mining student CVS repositories for performance indicators. ACM SIGSOFT Software Engineering Notes 30, 4 (2005), 1–5.

[12] Daniel Rocco and Will Lloyd. 2011. Distributed version control in the classroom. In Proceedings of the 42nd ACM technical symposium on Computer science education, 637–642.

[13] David Robinson and Ken Coar. 2004. The common gateway interface (CGI) version 1.1. Network Working Group, RFC3875, Category: Informational (2004).

[14] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. 2003. Winnowing: local algorithms for document fingerprinting. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data, 76–85.

[15] Reneé C. Bryce, Alison Cooley, Amy Hansen, and Nare Hayrapetyan. 2010. A one year empirical study of student programming bugs. In 2010 IEEE Frontiers in Education Conference (FIE), IEEE, F1G- 1-F1G-7.

[16] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common logic errors made by novice programmers. In Proceedings of the 20th Australasian Computing Education Conference, 83–89.

Chapter 5 References

[1] David J. Malan, Brian Yu, and Doug Lloyd. 2020. Teaching academic honesty in CS50. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education, 282–288.

[2] Aiken, Alex, “Moss: A System for Detecting Software Similarity,” https://theory.stanford.edu/~aiken/moss/ (accessed May 2, 2022).

[3] Sathiamoorthy Manoharan and Ulrich Speidel. 2020. Contract cheating in computer science: A case study. In 2020 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE), IEEE, 91–98.

[4] “GitHub: Where the world builds software,” GitHub. https://github.com/ (accessed Jan. 12, 2022).

[5] “The rise of the zyLab program auto-grader in introductory CS courses” https://docs.google.com/document/d/e/2PACX-1vQYwxlY738_9zFFwOer1kKTN GuJx1Qe3IDW8X Hf_OOYbaq9Drf_a9ljCqjcHY9Vv4ryPK423W7FmHwZ/pub (accessed Jan. 12, 2022).

[6] “IDE | CodeHS.” https://codehs.com/ide (accessed Jan. 12, 2022).

[7] “zyLab Autograder, with Free Sample Labs in Java,” zyBooks. https://www.zybooks.com/catalog/zylab-autograder-with-free-sample-labs-in-java / (accessed Jan. 12, 2022).

[8] “zyBooks - Build Confidence and Save Time With Interactive Textbooks,” zyBooks. http://www.zybooks.com/home/ (accessed Jan. 12, 2022).

[9] “diff(1) - Linux manual page.” https://man7.org/linux/man-pages/man1/diff.1.html (accessed Jan. 12, 2022).

[10] “difflib — Helpers for computing deltas — Python 3.9.6 documentation.” https://docs.python.org/3/library/difflib.html (accessed Jan. 12, 2022).

[11] Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto. 2020. How different are different diff algorithms in git? *Empirical Software Engineering* 25, 1 (2020), 790–823.

[12] google/diff-match-patch. Google, 2021. (accessed: Jan. 12, 2022). [Online]. Available: https://github.com/google/diff-match-patch

Chapter 6 References

[1] Haibiao Ding and Mansur H. Samadzadeh. 2004. Extraction of Java program fingerprints for software authorship identification. *Journal of Systems and Software* 72, 1 (2004), 49–57.

[2] zyBooks. 2022. zyBooks - Build Confidence and Save Time With Interactive Textbooks,” zyBooks. Retrieved from https://www.zybooks.com/.

[3] python.org. 2022. python.org - cgi— Common Gateway Interface support. Retrieved from https://docs.python.org/3.7/library/cgi.html

Chapter 7 References

[1] Aiken, Alex, “Moss: A System for Detecting Software Similarity,” https://theory.stanford.edu/~aiken/moss/ (accessed May 2, 2022).

[2] Breanna Devore-McDonald, and Emery D. Berger. 2020. Mossad: Defeating software plagiarism detection. Proceedings of the ACM on Programming Languages 4, OOPSLA (2020), 1–28.

[3] Xin Chen, Brent Francia, Ming Li, Brian Mckinnon, and Amit Seker. 2004. Shared information, and program plagiarism detection. IEEE Transactions on Information Theory 50, 7 (2004), 1545–1551.

[4] Christian Collberg, Ginger Myles, and Michael Stepp. 2004. Cheating cheating detectors. TR04-05. Tucson, Department of Computer Science, University of Arizona 7, (2004).

[5] zyBooks. 2022. zyBooks - Build Confidence, and Save Time With Interactive Textbooks," zyBooks. Retrieved from https://www.zybooks.com/.

[6] python.org. 2022. python.org - cgi— Common Gateway Interface support. Retrieved from https://docs.python.org/3.7/library/cgi.html