How we Code Code: Leveraging GPT and Ordered Networks for Understanding Introductory Programming Education

Maciej Pankiewicz [0000-0002-6945-0523], Andres Felipe Zambrano[2][0000-0003-0692-1209], Amanda Barany[2][0000-0003-2239-2271], and Ryan S. Baker[2][0000-0002-3051-3232]

[1] Warsaw University of Life Sciences, Warsaw, Poland.
[2] University of Pennsylvania, Philadelphia, PA, United States.
maciej_pankiewicz@sggw.edu.pl

Abstract. University-level computer science (CS) courses can be particularly challenging for students with limited programming backgrounds. To support novice learners, instructors often employ automated assessment systems for programming assignments. These systems provide students with feedback on demand as they work through problems online. Outcomes remain mixed, perhaps due to differences in the strategies learners use to address different types of coding errors after receiving feedback. In this study, we used Ordered Network Analysis (ONA) to explore data from an automated assessment platform used iteratively by students in an introductory CS course. We employed a GPT-based approach to automate the process of qualitative coding on this dataset. Our analysis revealed behavioral differences in how high performing and low performing novice learners changed their code over time, particularly when disaggregated by the types of errors they faced (compiler versus non compiler errors). By understanding these patterns, instructors can create interventions that guide students through challenges, improving their chances of success in programming tasks.

Keywords: Computer Science Education, Automated Assessment, Ordered Network Analysis, Large Language Models, ChatGPT.

1 Introduction

Computer Science (CS) skills have gained attention in education research and practice, leading to global efforts to broaden and improve CS education [1, 2]. As part of this effort, automated assessment tools have been widely used in CS education to offer personalized on-demand feedback, which can be helpful when students need multiple attempts to master programming concepts [3, 4]. Although studies generally suggest that automated assessments can support learning [3], the benefits often vary, depending on how students approach and interact with these tools. However, it is not yet clear precisely how different behaviors and strategies influence learning outcomes, underscoring the need for more research into the specific activity patterns associated with success and failure in programming tasks.

We investigate this by using Ordered Network Analysis (ONA) to analyze data from an automated programming assessment system used in an introductory CS course, where students iterate on specific coding goals while intermittently using the assessment system. By investigating the emergent patterns during their work, we aim to contribute to a deeper understanding of how student actions influence their learning outcomes in university-level CS education.

Specifically, we investigate whether we can automatically detect fine-grained and meaningful changes in student behavior during programming in a scalable fashion, compose them into patterns using ONA, and then investigate which of these patterns are associated with better learning outcomes. By doing so, we seek to identify behavior patterns in debugging that could inform targeted interventions to support students in achieving programming success.

2 Related Work

2.1 Automated Assessment in Computer Science Education

Systems for automated assessment of programming code have become increasingly prevalent in computer science education assisting students in many different introductory programming courses and providing support for a wide range of programming languages [5, 6].

The availability of data originating from such systems has resulted in increased research on areas such as the impact of syntax (compiler) errors on student progress [7], automated generation of hints and feedback [8, 9], or early prediction of student performance in a programming course [10]. In contrast to some quantitative research on learning to program, qualitative research in this domain has been limited to small sample sizes [11], as the extensive volume of available data often surpasses the capacity for traditional human coding. However, the advent of Large Language Models (LLMs) now enables the automated execution of some qualitative research processes at increasingly high quality [12, 13], thereby suggesting a potential for a more thorough analysis of programming code and detecting valuable insights at larger scale.

In CS automated assessment systems, program correctness is checked using a set of predefined tests, assuming there are no syntax errors (which makes it possible for the code to be compiled and executed). If the code does not compile, compiler errors may provide insights into student struggle. For a more detailed analysis of how learners attempt to improve across attempts, programming code can be examined using an Abstract Syntax Tree (AST), which is a simplified representation of the code structure [14]. ASTs allow tracking of precise changes between different versions of the code [14]. However, ASTs are often not useful until the student is able to produce code that compiles without errors, a challenge for many novice students in introductory programming courses who struggle with code syntax. Since an AST represents the syntactic structure of code, any part of the code that doesn't adhere to the language's syntax might not be correctly represented in the tree, or the tree generation might fail entirely. This issue is often addressed by including only compiled code in the analysis [15], which offers an incomplete picture of novices' CS understanding. Regular expressions

are another alternative for examining changes in students' attempts by searching for specific sequences of characters in those lines where changes are detected [16]. However, examining such changes based on a mere identification of fixed sequences of characters can also offer a limited understanding of those students who struggle with code syntax because those types of errors might result in the specific known sequences not being present in their code. Therefore, these types of automated approaches can be insufficient for fully understanding how students improve their code across attempts.

2.2 QE Approaches for CS Education

Given the challenges with existing approaches to automated assessment for studying CS learning, some programming education researchers have instead used manual labeling of programming code [17]. This often involves researchers coding data sources for the presence of relevant qualitative constructs to understand, for example, students' thought processes when creating code comments [18], the acquisition of data science skills [11] or patterns of debugging behaviors [19].

Research in computer science education has also recently begun to employ Quantitative Ethnography techniques such as Epistemic Network Analysis (ENA) to investigate patterns in coding practices, with a particular focus on collaborative learning. In K-12 settings, Su et al. [20] and Vandenberg et al. [21] have used ENA to analyze interactions within pair programming and across course topics. At the university level, ENA has also been utilized to compare computational thinking strategies among novice learners, distinguishing between low- and high-performing students during collaborative problem-solving sessions in CS courses [22]. Data sources from these examples consisted of transcripts of discourse or student written reflections, which offer insights on student mindset, but not direct assessment of programming skill development.

To address this limitation, Pinto and colleagues [23] qualitatively coded programming actions directly, building epistemic networks to explore debugging patterns in code submissions and revealing distinct approaches between experienced and novice programmers. The codebook contained constructs describing changes between two subsequent code submissions: a) code changes (e.g. changing variable name, massive deletion), b) results of the compilation event (new error detected in the compilation process), and c) combination of both: code changes and result of the compilation event (line with an error deleted). Although their results from the use of these codes highlighted differences between experienced and novice students' debugging behaviors, coding novices' data for more nuanced constructs as those constructs emerge across repeated programming attempts might reveal further insights into complex CS learning processes.

Therefore, we develop and apply a set of qualitative codes regarding students' fine-grained modifications to programming code, processed efficiently by an LLM. We then employ Epistemic Network Analysis (ENA) and Ordered Network Analysis (ONA) techniques to delve into the debugging behaviors of novice programmers. By doing so, we aim to discern the strategies that correlate with high and low outcomes on final examinations, providing deeper insights into the learning processes of students engaged with these automated systems.

3 Methods

3.1 Data Collection and Research Context

The data for this study was gathered during the fall 2022/2023 semester as part of a CS1 course, a required first-semester class for computer science students at a large European university. The dataset includes 1) responses to a questionnaire completed at the start of the semester, 2) results from a pre-test taken immediately after the questionnaire, 3) logs of activity on an online platform for automated programming assessment throughout the semester, and 4) scores from the final test, which was a graded assignment at the semester's end. All participants (N=198) provided consent before their data was included in the study.

An automated assessment platform for programming assignments was provided as a learning tool within the course. Use of the tool was optional and did not count towards the final grade. During the study, 169 students submitted code 44,448 times through the platform. They uploaded their C# code for evaluation across 146 tasks, which spanned basic programming topics introduced during the semester including types and variables (33 tasks), conditional statements (25 tasks), recursion (28 tasks), and arrays and loops (60 tasks).

Directly after each submission, the code was automatically compiled. If compiler errors were detected at this stage (which means the code cannot be executed and tested), students were presented with a feedback message that listed the detected compiler errors. If compilation succeeded, the code was executed against a set of unit tests that verified if the students' code met the assignment requirements. In this case, students were presented with a list of all tests executed on their code. Each test was marked with fail or success. To retrieve details about the test execution, students had to click on a selected test. As correcting the submitted code requires understanding which test requirements were not met, we only included clicks checking the details of tests that failed.

3.2 Knowledge Self-Reports and Tests

In this work, we examine the debugging behaviors of learners who self-reported having little to no programming experience prior to joining the course. The level of prior programming experience was determined based on student responses to a survey item deployed during the first class sessions: "On the scale 1-5, where 1 is zero experience, and 5 is a lot of experience, please rate your basic programming knowledge (types, variables, conditional statement, recursion, loops, arrays)." We considered students who responded with a 1 or 2 (N=110 out of total 198) in the questionnaire to be novices, as in past use of this instrument [24], and included them in our analyses. We also excluded students who did not use the online platform to solve conditional statements tasks (69 experienced and 100 novices solved at least one conditional statement task in the platform).

We validated students' self-reports of expertise by asking students to take a pre-test assessing their general programming knowledge (see Table 2). This test was

administered immediately after students submitted the questionnaire. At the end of the semester, a final test was administered to evaluate student knowledge of the introductory programming concepts taught in the course. Within the novice group, we further categorized students based on their performance. 40 novices with final test scores within the highest quartile were assigned to the high-performing group (hp-novices). 28 novices with scores in the lowest quartile were assigned to the low-performing group (lp-novices). The two quartiles were different sizes due to students with the same score on the final test.

In this research, we focus on submissions made on conditional statements tasks, for which 10,673 code submissions (7,136 from novices) have been collected on the platform. We chose this specific set of tasks because they are the first module in the semester that requires a basic understanding of both syntax and programming logic. By examining the correctness of submissions and debugging behaviors to solve errors in this set of tasks, instructors could recognize early in the semester whether some students still need to learn some key initial concepts that are critical for future success in programming learning.

3.3 Qualitative Coding Procedures

When coding student data, our objective was to identify and characterize the changes made across all pairs of consecutive student submissions for the same task. For a task of this detail and a dataset of this size, human qualitative coding of changes across submissions is time-consuming, and the regular expression automated approach often used in QE (e.g. [25, 26]) is challenging to apply reliably because novice students' code often does not follow syntax rules. With increased availability of large language models (LLMs), however, training tools such as GPT-4 to emulate human qualitative coding procedures at scale has become more feasible for automated qualitative coding of text (e.g., [12]).

To achieve this, we first employed a difference algorithm (diff) to generate changes between all pairs of subsequent code submissions. We created a tool that presents these changes in HTML format, incorporating additional fields that allow for labeling each identified change (Fig. 1). Two human coders used this tool and initially performed qualitative coding on a set of 100 examples of consecutive submissions with differences that indicate the actions taken by students to solve the corresponding errors (Table 1).

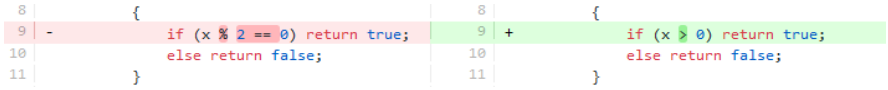

Fig. 1. A sample of two consecutive pieces of programming codes submitted by a student with changes across submissions highlighted.

Next, we used a prompt engineering approach and employed GPT-4 (gpt-4-turbo-2024-04-09) for the automated coding of the remaining submissions. Our analysis focused only on submissions for tasks involving conditional statement ("if"), as this area has been previously identified as one of the earliest where differences between high and low-performing students become apparent [24, 27].

Table 1. Codebook with definitions, accuracy and reliability metrics.

Code	Definition	Human 1 - GPT		Human 1 - Human 2	
		Accuracy	K	Accuracy	K
If Header	Student submission contains modifications to the if condition/ header	0.907	0.808	0.981	0.959
Function Return	Student submission contains modifications inside the return statement	0.928	0.814	0.916	0.748
Commenting	Student commented at least one line in a submission	0.990	0.852	1.000	1.000
Code Added	Student added new code lines in a submission	0.979	0.905	0.972	0.854
Code Removed	Student removed code lines in the submission	0.961	0.779	0.991	0.955
Testing	Student modified code inside the Main function (section of the code used for testing).	0.918	0.784	0.963	0.907
No Change	Student submission is identical to the previous submission	*	*	*	*
Other Change	Student submission contains changes other than those identified above	*	*	*	*
Compiler Error	Student submitted code containing a compiler error	*	*	*	*
Unit Test Failed	Student submitted compiling code but failing to pass at least one unit test	*	*	*	*
Feedback Clicked	Student requested feedback after submitting code that failed to pass all unit tests.	*	*	*	*

* Did not require coding procedures

During prompt engineering, GPT's performance increased when it was presented with the results of the difference algorithm (e.g., the lines affected by a change as highlighted in Figure 1) along with the code. We designed a set of prompts that achieved a Cohen's Kappa over 0.7 with human coders in the initial subset of 100 examples (See Table 1). Finally, we applied GPT-based coding to the remaining submissions for the "if" set of tasks.

Several codes included in the analysis did not require qualitative coding techniques. Instances of no change across submissions were identified using the diff algorithm; changes identified by the diff algorithm that did not fall under the human-identified coding categories were grouped into "Other change". Student responses marked as having compiler errors or unit tests that failed were pulled directly from the log data from student use of the automated platform, as were instances in which students chose to access the platform's feedback.

3.4 Data Analysis Procedures

To compare the debugging behaviors between the two groups (hp-novices vs. lp-novices), we used Ordered Network Analysis (ONA), which accounts for the order of appearance of constructs. This technique helped us identify which actions followed an unsuccessful submission and what the subsequent steps taken to rectify coding mistakes

were. Additionally, ONA examines self-transitions within each construct, enabling us to determine whether students repeatedly test a particular debugging strategy and if they persist in making the same type of changes when submission contains a compiler error or unit tests that failed [7].

Using ONA, we conducted a two-stage analysis. First, using a difference model, we examined the primary distinctions between high-performing and low-performing groups based on all their submissions. This model included the two types of unsuccessful submissions as constructs, as well as the student's choice to click on feedback, helping us understand how students react to different error types.

Recognizing that the results of epistemic networks can be influenced by some categories being much more frequent than others (in this case, submissions with compiler errors and failed unit tests), we conducted a second-stage analysis where we compared the debugging behaviors specific to each type of an unsuccessful submission. For this, we developed separate difference models for 1) submissions immediately following a submission with a compiler error and 2) submissions immediately following a submission containing unit tests that failed. We analyzed the actions taken in the submission immediately after an unsuccessful one because these actions represent the students' direct responses to resolve the problem. For the same reason, we used a moving stanza window of size one for all models, pairing only two consecutive submissions to identify the actions that followed each unsuccessful submission in the primary analysis and the action that followed the previous unsuccessful attempt for the same type of error in the secondary analysis.

4 Results

4.1 Descriptive Statistics

Table 2 shows descriptive statistics for total submissions, the number of tasks attempted and correctly solved, incorrect submissions due to a compiler error and failed unit tests, and initial and final scores for 4 groups of students: Experienced, Novices, Novices with the highest final test score (top quartile) and Novices with low performance in the final test (bottom quartile).

Students who self-reported being novices had much lower pretest scores (Mdn=16%) than the other students (Mdn=65%), $p<0.001$, $U=452$ for a Mann-Whitney U test. The primary distinction between these groups in their use of the online platform was in the number of submissions, which was statistically significantly different, $U=2214$, $p<0.001$, despite these two groups having a nearly identical total number of attempted and correctly completed tasks. This difference is the result of the higher number of submissions containing compiler errors (Mdn=19.0) and failed unit tests (Mdn=20.0) that novices made compared to experienced students (Mdn of 11.0 and 10.0 for compiler errors and failed unit tests, respectively).

The difference between experienced and novice students is reduced in the final scores at the end of the semester, although still statistically significant (Mdn of 67% and 56%, respectively, $p<0.001$, $U=2228.5$ for a Mann-Whitney U test). This result indicates that even though they started the semester with lower prior knowledge, some

novices were able to close that knowledge gap, while other novices had much poorer learning (the low-performing group of novices had a Mdn of 22% on the final exam). This sharp contrast between outcomes for the novice groups underscores the importance of recognizing the differences in the interaction with the platform that might help explain these differences in the learning outcome.

Though the two groups of novice students (high-performing and low-performing) achieved different outcomes on the post-test (by definition, given how the two groups were selected), the two groups started with a similar pre-test score (Mdn of 18% and 16%, p=0.15, U=444.5 for a Mann-Whitney U test). This indicates that the difference in their final score is not a consequence of having different levels of prior knowledge of the course content.

Table 2. Descriptive statistics comparing experienced students and novice students with high and low learning gains.

Variable	Experienced (N=69)		Novices (N=100)		Novices High Score (N=40)		Novices Low Score (N=28)	
	Mdn	Avg (SD)	Mdn	Avg (SD)	Mdn	Avg (SD)	Mdn	Avg (SD)
Submissions	46.0	51.0 (27.3)	64.0	71.3 (41.0)	57.0	64.3 (31.6)	73.0	79.8 (49.4)
Comp. Errors	11.0	14.7 (12.7)	19.0	26.6 (24.8)	16.0	21.1 (18.6)	26.0	35.4 (32.4)
Tests failed	10.0	13.7 (11.7)	20.0	21.7 (15.7)	17.5	20.0 (14.3)	21.5	22.7 (15.6)
Tasks	25.0	21.4 (6.6)	25.0	21.9 (5.8)	25.0	22.5 (5.0)	25.0	21.4 (6.5)
Correct Tasks	24.0	21.0 (6.5)	24.0	21.2 (5.9)	25.0	22.1 (4.9)	24.0	20.3 (6.6)
Pre-Test	65%	64% (0.21)	16%	20% (0.18)	18%	25% (0.20)	16%	16% (0.13)
Final Test	67%	67% (0.2)	56%	51% (0.22)	67%	73% (0.09)	22%	21% (0.1)

4.2 Differences between High and Low-Performing Novices

Figure 2 shows the ONA model of the most commonly occurring codes for all the 100 novices (left) and a difference model comparing the actions of the two groups of novices described in the previous section (right). Table 3 shows the line weights (lw) of the highest differences found when comparing these two groups (differences higher than 0.02). Along the X axis (MR1), a Mann-Whitney U test showed significant differences across both groups (U=319, p=0.005). The most substantial difference between the two groups of novices is observed for the compiler errors code. The difference model indicates that low-performing novices are more likely to repeatedly submit code with compiler errors (lw=0.27) than their high-performing counterparts (lw=0.18). This association reverses for submissions with failed unit tests, where high-performing novices (lw=0.17) seem to have a slightly higher tendency to submit another code with a failed unit test compared to the low-performing group (lw=0.14). It is worth noting that this difference in the line weights for submissions with failed unit tests does not imply that high-performing students make more errors of this type (See Table 2). It instead indicates that these students require fewer attempts to reach a correct submission, and

therefore, these students have a higher percentage of submissions with failed unit tests than the low-performing novices do.

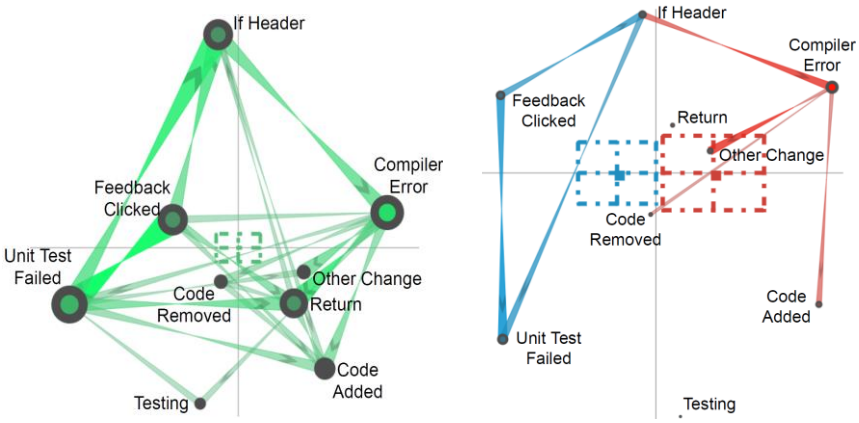

Fig. 2. A summary ordered network considering all novices (left) and a difference model comparing novices with high (blue) and low (red) final test scores (right). Only connections with weights over 0.02 are shown to enhance the clarity of the visualization.

Table 3. Line weights for the summary model and the difference model contrasting novices with high and low scores in the final test. Associations are listed based on the magnitude of the difference between the two groups.

Association	Summary Model	Difference Model		
		High	Low	Diff
Compiler Error → Compiler Error	0.22	0.18	0.27	-0.08
Feedback Clicked → Unit Test Failed	0.26	0.29	0.22	0.07
Unit Test Failed → Feedback Clicked	0.25	0.28	0.22	0.06
If Header → Compiler Error	0.14	0.12	0.18	-0.06
Compiler Error → Other Change	0.13	0.11	0.17	-0.06
Feedback Clicked → If Header	0.17	0.19	0.14	0.05
If Header → Unit Test Failed	0.16	0.18	0.13	0.05
If Header → Feedback Clicked	0.13	0.15	0.10	0.05
Unit Test Failed → If Header	0.21	0.23	0.19	0.04
Compiler Error → If Header	0.14	0.12	0.16	-0.04
Compiler Error → Code Added	0.11	0.10	0.14	-0.04
Unit Test Failed → Unit Test Failed	0.16	0.17	0.14	0.03
Code Added → Compiler Error	0.10	0.09	0.12	-0.03
Feedback Clicked → Feedback Clicked	0.10	0.11	0.08	0.03
Other Change → Compiler Error	0.10	0.08	0.11	-0.03
Code Added → Code Added	0.04	0.03	0.05	-0.03
Compiler Error → Code Removed	0.08	0.07	0.09	-0.02
Compiler Error → Return	0.19	0.18	0.20	-0.02

This contrast in the base rates of submissions with a compiler error and submissions with failed unit tests influences most of the observed differences in the Ordered Network. Although both groups of novices frequently edit the *If Header* following

submissions containing a compiler error or failed unit tests (lw=0.14 and lw=0.21, respectively), this behavior is more pronounced among low-performing novices for compiler errors (lw=0.16) and among high-performing novices for failed unit tests (lw=0.23). In both cases, this temporal association with the *If Header* construct is bidirectional mainly because students' subsequent submissions after experiencing a compiler error also often contain compiler errors. Also, in the event that a submission contains a failed unit test, it usually takes several attempts until fully correct code is submitted (as also shown by the self-transition weights of these two constructs).

Following compiler errors, low-performing novices often make additional changes to their code that do not involve the *If Header*, *Return Statement*, *Testing*, or *Adding* or *Removing* lines of code (lw=0.17), unlike their high-performing counterparts (lw=0.11). Additionally, low-performing novices are more likely to *Add code* (lw=0.14) or *Remove code* (lw=0.09) compared to high-performing novices (lw=0.10 and lw=0.07, respectively), although these differences are weaker than the former. Both behaviors are unexpected responses to compiler errors and suggest that these students may have a misunderstanding of what a compiler error means.

Across all novice students, there is a stronger connection between a submission containing a failed unit test and consulting feedback to identify which unit test failed in the previous submission (lw=0.25). This behavior suggests that students frequently request feedback, but, as also demonstrated by the self-transitions of both error-related constructs, students' submissions containing a compiler error are often followed by another submission with a compiler error. Also, when a student makes a submission with failed unit tests, subsequent submissions often also contain failed unit tests. Although all novices show this behavior (requesting feedback after a submission with failed unit tests), the difference model reveals that high-performing novices (lw=0.28) engage in this action more frequently than their low-performing counterparts (lw=0.22). This indicates that students in the low-performing group may sometimes avoid or forget to request detailed feedback about the failed unit tests from their last submission.

4.3 Specific Differences between High and Low Performing Students for each Type of Error

To account for the specific differences in the frequency of submissions containing a compiler error and failed unit tests between each group, we conducted a secondary analysis and developed separate difference models for each error type, as shown in Figure 3 and Table 4. Along the X axis (MR1), the two groups were statistically significantly different for submissions following a compiler error ($U=270$, $p<0.001$) and for submissions following failed unit tests ($U=334$, $p=0.009$). For compiler errors, the difference model shows that high-performing novices tend to repeatedly modify the *Return statement* more frequently (lw=0.31) than low-performing novices (lw=0.24), although this is still the strongest association for both groups. Additionally, high-performing novices more often adjust the *If Header* after modifying the *Return statement* (lw=0.15) and alter the *Return statement* following other types of code changes (lw=0.12) more than the low-performing groups (lw=0.12 and lw=0.08, respectively).

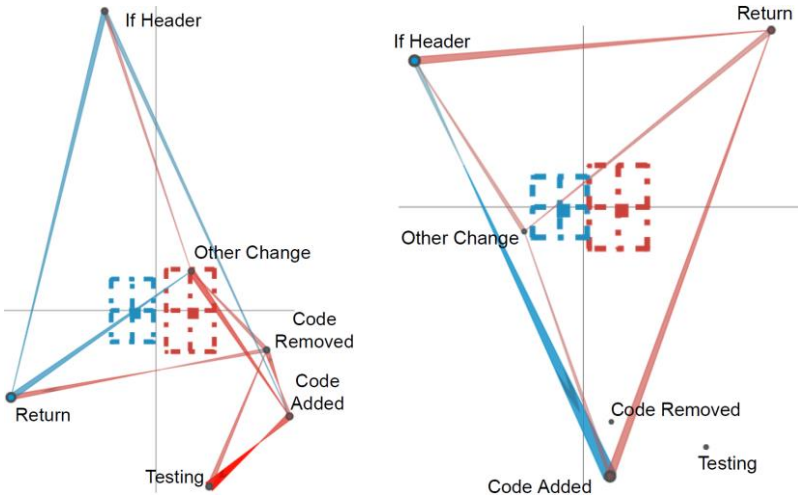

Fig. 3. Difference models comparing high (blue) and low (red) performing novices after the submission containing compiler errors (left) and failed unit tests (right). Only connections with weights over 0.02 are shown to enhance the clarity of the visualization.

Table 4. Line weights of difference models contrasting behaviors after the submission containing compiler errors and failed unit tests of novices with high and low scores in the final test. Associations are listed based on the magnitude of the difference between the 2 groups for compiler errors. Differences greater than 0.03 are shown in bold.

	Compiler Errors			Failed Unit Tests		
Association	**High**	**Low**	**Diff**	**High**	**Low**	**Diff**
Return → Return	**0.31**	**0.24**	**0.07**	**0.18**	**0.22**	**-0.04**
Code Added → Testing	**0.03**	**0.10**	**-0.07**	0.05	0.06	-0.01
Other Change → Return	**0.12**	**0.08**	**0.04**	**0.03**	**0.06**	**-0.03**
Testing → Code Added	**0.05**	**0.09**	**-0.04**	0.05	0.06	-0.01
Return → If Header	**0.15**	**0.12**	**0.03**	**0.20**	**0.24**	**-0.04**
Code Removed → Return	**0.12**	**0.15**	**-0.03**	0.08	0.09	-0.01
Code Added → Code Removed	**0.11**	**0.14**	**-0.03**	0.09	0.10	-0.01
Code Added → If Header	**0.13**	**0.10**	**0.03**	0.18	0.16	0.03
Code Added → Code Added	**0.08**	**0.11**	**-0.03**	0.08	0.10	-0.02
Code Removed → Testing	**0.01**	**0.04**	**-0.03**	0.02	0.02	0.00
Other Change → Code Removed	0.05	0.07	-0.03	0.02	0.02	0.00
Code Removed → Code Added	0.12	0.14	-0.03	0.08	0.09	-0.01
Other Change → Other Change	0.13	0.15	-0.02	**0.06**	**0.03**	**0.03**
Return → Code Removed	0.10	0.12	-0.02	0.09	0.10	-0.01
Other Change → If Header	0.09	0.11	-0.02	0.07	0.06	-0.01
Return → Code Added	0.20	0.21	-0.01	**0.15**	**0.20**	**-0.04**
If Header → Code Added	0.10	0.11	-0.01	**0.23**	**0.15**	**0.07**
If Header → Other Change	0.11	0.10	0.01	0.05	0.07	-0.02
Other Change → Code Added	0.08	0.09	0.01	0.04	0.06	0.02
If Header → If Header	0.21	0.21	0.00	**0.44**	**0.34**	**0.10**

The difference model for submissions with compiler errors shows that the primary action more prevalent among low-performing novices than high-performing ones is

modifying the *Testing* section of the code after *Adding* more lines (lw=0.10 for low-performing vs. lw=0.03 for high-performing). Similarly, low-performing novices are more likely to alter the *Testing* section after *Removing Code* (lw=0.04) and exhibit more frequent fluctuation between *Adding* and *Removing Code* (lw=0.14 for removing after adding, and lw=0.11 for adding after removing). Low-performance novices also modify the *Return statement* more frequently (lw=0.15) than the high-performance group (lw=0.12), particularly after *Removing Code*.

In summary, high-performing novices are more likely to act upon an apparent understanding that the most frequent sources of compiler errors (at least for the specific tasks considered here) are the *If Header* and *Return statements*, which are the sections they primarily modify in their submissions. While low-performing students also focus on these two code sections, they are more inclined to also add or remove code within the function body, testing, or other sections of code that are less likely to resolve a compiler error.

For submissions containing failed unit tests, the high-performing group predominantly focuses on recurrently modifying the *If Header* (lw=0.44), possibly recognizing it as the primary section of the code where a mistake in logic could appear and cause their submissions to fail some unit tests. This is particularly true for the tasks considered in this study, which focus on conditional statements. High-performing novices also more frequently add lines of code after adjusting the *If Header* (lw=0.23) when failed unit tests have been identified in the submission, which is typical debugging behavior for addressing issues in conditional statement tasks.

Although low-performing novices also frequently modify the *If Header* (lw=0.34) and add lines of code afterward (lw=0.15), they do so less often than their high-performing counterparts. Occasionally, they shift to other types of changes (lw=0.07) such as introducing new variables or modifying existing expressions. They also sometimes choose to first modify the *Return statement* and then adjust the *If Header* (lw=0.24) or add new lines of code (lw=0.20). Given the cumulative nature of programming learning across tasks, it is possible that low-performing novices were trying to apply actions from prior coding tasks that they had not yet mastered. Although high performers also engage in these behaviors (lw=0.20 and lw=0.15, respectively), their strategies tend to focus more directly on the *If Header*.

Examination of the two difference models reveals contrasting behaviors for each novice group when they encounter failed unit tests versus compiler errors. When dealing with compiler errors, the transition from modifying the *Return statement* to the *If Header* is more prevalent among high-performing novices, indicating their recognition that many of these errors may stem from simple typos in the *Return statement* (e.g., missing a semicolon, improper operator usage, etc.). For submissions with failed unit tests, high performers recognize that primary logic issues can often be resolved by adjusting the *If Header*. Low-performing novices instead tend to make this transition from modifying the *Return statement* to the *If Header* more frequently for submissions with failed unit tests. This suggests that they might not have the same level of discernment between the types of errors as their high-performing peers, leading them to adopt strategies that may not be as effective in resolving the issues in their program code.

5 Discussion and Conclusion

In this study, we investigated patterns of fine-grained changes in novice learners' debugging behaviors when using an automated assessment tool, and the association between these changes and learning outcomes. Given that manually labeling a complete dataset with thousands of submissions is highly time-consuming to do manually, we used GPT-4 turbo to automate the process of identifying differences in students' programming code across multiple submissions. Next, we employed these automatically labeled examples to track code modifications made by students and used ONA to identify the differences in coding behaviors between groups of novice programmers who ultimately achieved high final scores and those who performed poorly.

Our findings indicate that both groups began the course with similar incoming knowledge, and both groups demonstrated similar rates of activity, correctly solving approximately the same number of tasks involving conditional statements. When examining the complex patterns of behaviors using ordered networks, however, we observed nuanced differences in how each group interacted with the code. These differences became apparent early in the semester.

In particular, students who eventually scored well on the final exam tended to make changes involving key concepts relevant to the current stage of the learning path, such as conditional statements and the use of the return keyword. In contrast, students who scored low on the test frequently made changes unrelated to these core topics. This pattern suggests that, already at an early stage, the lower-performing students likely had a weaker grasp of key initial topics such as types and variables.

Although future research still needs to investigate how these differences in debugging behaviors also manifest for more advanced tasks throughout the semester (such as loops or recursion), they appeared to have a lasting impact on students' progress. Despite visible engagement with the tasks, these foundational misunderstandings seemed to propagate throughout the course, culminating in lower final scores. This study highlights the critical role of early diagnostic and intervention strategies in programming courses where early foundational understanding is crucial for later success. Scaffolds could include tailored messages that can help indicate where errors can originate, or hints or notes to remind students of key concepts.

GPT-based models have previously been employed to generate personalized feedback for students by analyzing the submissions that led to their most recent errors [9]. These models have demonstrated promising results in helping students learn to produce correct solutions, even after the feedback is turned off. Our findings suggest that these types of interventions can be further refined to not only consider students' recent errors, but also to scaffold the behaviors they adopt while addressing compiler errors or failed unit tests, specifically helping them learn to focus on modifying the right sections of their programs. Additionally, it may be beneficial for instructors to analyze the individual ordered network models for each student as well as the overall classroom model to identify whether certain students or a significant portion of the class lacks foundational knowledge. By supporting students in learning to focus their debugging in the right areas, and addressing foundational knowledge gaps promptly, we may be able to mitigate the difficulties that students encounter later in the semester.

Acknowledgments. Andres Felipe Zambrano thanks the Ministerio de Ciencia, Tecnología e Innovación and the Fulbright-Colombia commission for supporting his doctoral studies through the Fulbright-MinCiencias 2022 scholarship.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Ou, Q., Liang, W., He, Z., Liu, X., Yang, R., Wu, X.: Investigation and analysis of the current situation of programming education in primary and secondary schools. Heliyon. 9, (2023).
2. Vegas, E., Hansen, M., Fowler, B.: Building skills for life: how to expand and improve computer science education around the world (2021).
3. Paiva, J.C., Leal, J.P., Figueira, Á.: Automated assessment in computer science education: A state-of-the-art review. ACM Transactions on Computing Education (TOCE). 22, 1–40 (2022).
4. Pankiewicz, M., Baker, R., Ocumpaugh, J.: Using intelligent tutoring on the first steps of learning to program: affective and learning outcomes. In: International Conference on Artificial Intelligence in Education. pp. 593–598. Springer (2023).
5. Lobb, R., Harlow, J.: Coderunner: a tool for assessing computer programming skills. ACM Inroads. 7, 47–51 (2016).
6. Edwards, S.H., Murali, K.P.: CodeWorkout: Short Programming Exercises with Built-in Data Collection. In: Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education. pp. 188–193. (2017).
7. Jadud, M.C., Dorn, B.: Aggregate Compilation Behavior: Findings and Implications from 27,698 Users. In: Proceedings of the Eleventh Annual International Conference on International Computing Education Research. pp. 131–139. Association for Computing Machinery, New York, NY, USA (2015).
8. Rivers, K., Koedinger, K.R.: Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. International Journal of Artificial Intelligence in Education. 27, 37–64 (2017).
9. Pankiewicz, M., Baker, R.S.: Large Language Models (GPT) for Automating Feedback on Programming Assignments. In: International Conference on Computers in Education. pp. 68–77 (2023).
10. Pereira, F.D., Fonseca, S.C., Oliveira, E.H.T., Cristea, A.I., Bellhäuser, H., Rodrigues, L., Oliveira, D.B.F., Isotani, S., Carvalho, L.S.G.: Explaining Individual and Collective Programming Students' Behavior by Interpreting a Black-Box Predictive Model. IEEE Access. 9, 117097–117119 (2021).
11. Theobold, A.S., Wickstrom, M.H., Hancock, S.A.: Coding Code: Qualitative Methods for Investigating Data Science Skills. Journal of Statistics and Data Science Education. 32, 161–173 (2024).
12. Zambrano, A.F., Liu, X., Barany, A., Baker, R.S., Kim, J., Nasiar, N.: From ncoder to chatgpt: From automated coding to refining human coding. In: International Conference on Quantitative Ethnography. pp. 470–485. Springer (2023).
13. Barany, A., Nasiar, N., Porter, C., Zambrano, A.F., Andres, A., Bright, D., Choi, J., Gao, S., Giordano, C., Liu, X., Mehta, S., Shah, M., Zhang, J., Baker, R.S.: ChatGPT for Education

Research: Exploring the Potential of Large Language Models for Qualitative Codebook Development. In: International Conference on Artificial Intelligence in Education. Springer (2024).
14. Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 313–324. (2014).
15. Liu, K., Kim, D., Bissyandé, T.F., Yoo, S., Le Traon, Y.: Mining Fix Patterns for FindBugs Violations. IEEE Transactions on Software Engineering. 47, 165–188 (2021).
16. Ade-Ibijola, A., Ewert, S., Sanders, I.: Abstracting and Narrating Novice Programs Using Regular Expressions. In: Proceedings of the Southern African Institute for Computer Scientist and Information Technologists Annual Conference 2014. pp. 19–28. (2014).
17. De Lucia, A., Di Penta, M., Oliveto, R., Panichella, A., Panichella, S.: Labeling source code with information retrieval methods: an empirical study. Empirical Software Engineering. 19, 1383–1420 (2014).
18. Mohammadi-Aragh, M.J., Beck, P.J., Barton, A.K., Reese, D., Jones, B.A., Jankun-Kelly, M.: Coding the Coders: A Qualitative Investigation of Students' Commenting Patterns. In: 2018 ASEE Annual Conference & Exposition. ASEE Conferences, Salt Lake City, Utah (2018).
19. Lewis, C.M.: The importance of students' attention to program state: a case study of debugging behavior. In: Proceedings of the Ninth Annual International Conference on International Computing Education Research. pp. 127–134. (2012).
20. Su, Y.-S., Wang, S., Liu, X.: Using Epistemic Network Analysis to Explore Primary School Students' Computational Thinking in Pair Programming Learning. Journal of Educational Computing Research. 62, 559–593 (2024).
21. Vandenberg, J., Lynch, C., Boyer, K.E., Wiebe, E.: "I remember how to do it": exploring upper elementary students' collaborative regulation while pair programming using epistemic network analysis. Computer Science Education. 33, 429–457 (2023).
22. Wu, B., Hu, Y., Ruis, A.R., Wang, M.: Analysing computational thinking in collaborative programming: A quantitative ethnography approach. Journal of Computer Assisted Learning. 35, 421–434 (2019).
23. Pinto, J.D., Liu, Q., Paquette, L., Zhang, Y., Fan, A.X.: Investigating the Relationship Between Programming Experience and Debugging Behaviors in an Introductory Computer Science Course. In: International Conference on Quantitative Ethnography. pp. 125–139. (2023).
24. Zambrano, A.F., Pankiewicz, M., Barany, A., Baker, R.S.: Ordered Network Analysis in CS Education: Unveiling Patterns of Success and Struggle in Automated Programming Assessment. In: International Conference on Innovation and Technology in Computer Science Education. pp. 443–449. Association for Computing Machinery (2024).
25. Cai, Z., Siebert-Evenstone, A., Eagan, B., Shaffer, D.W., Hu, X., Graesser, A.C.: nCoder+: A Semantic Tool for Improving Recall of nCoder Coding. In: Eagan, B., Misfeldt, M., and Siebert-Evenstone, A. (eds.) Advances in Quantitative Ethnography. pp. 41–54. (2019).
26. Dubovi, I., Tabak, I.: Interactions between emotional and cognitive engagement with science on YouTube. Public Understanding of Science. 30, 759–776 (2021).
27. Izu, C., Denny, P., Roy, S.: A Resource to Support Novices Refactoring Conditional Statements. In: Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1. pp. 344–350.