Investigating the Relationship Between Programming Experience and Debugging Behaviors in an Introductory Computer Science Course

Juan D. Pinto[1](✉), Qianhui Liu[1], Luc Paquette[1], Yingbin Zhang[2], and Aysa Xuemo Fan[1]

[1] University of Illinois Urbana-Champaign, Champaign, IL 61820, USA
jdpinto2@illinois.edu
[2] South China Normal University, Guangzhou Guangdong 510631, China

Abstract. Debugging is a challenging task for novice programmers in computer science courses and calls for specific investigation and support. Although the debugging process has been explored with qualitative methods and log data analyses, the detailed code changes that describe the evolution of debugging behaviors as students gain more experience remain relatively unexplored. In this study, we elicited "constituents" of the debugging process based on experts' interpretation of students' debugging behaviors in an introductory computer science (CS1) course. Epistemic Network Analysis (ENA) was used to study episodes where students fixed syntax/checkstyle errors or test errors. We compared epistemic networks between students with different prior programming experience and investigated how the networks evolved as students gained more experience throughout the semester. The ENA revealed that novices and experienced students put different emphasis on fixing checkstyle or syntax errors and highlighted interesting constituent co-occurrences that we investigated through further descriptive and statistical analyses.

Keywords: Computer Science Education · Debugging · Programming Experience · Epistemic Network Analysis · CS1

1 Introduction

Debugging is an important component of computer programming where students "find out exactly where the error is and how to fix it" [1]. Different from general programming ability, debugging skills cannot be immediately obtained from writing code [2] and thus deserve individual pedagogical attention [1]. Novice programmers often find debugging difficult for two main reasons. First, successful debugging requires a wide range of knowledge—including general programming expertise and knowledge of debugging methods—that novice programmers may not possess [3]. This deficiency of debugging knowledge and strategic skills often hinders students from controlling the programming process [4]. Second, the process of debugging often happens outside of class, so

G. Arastoopour Irgens and S. Knight (Eds.): ICQE 2023, CCIS 1895, pp. 125–139, 2023.
https://doi.org/10.1007/978-3-031-47014-1_9

instructors have limited opportunities to directly support students when they encounter difficulties and misconceptions [5].

Epistemic Network Analysis (ENA) is a technique that detects and measures associations between coded data elements and represents the associations through a dynamic network [6]. It allows researchers to visually and statistically compare different groups' networks. ENA can be applied to understand students' debugging processes by revealing the associations between debugging behaviors within a window slice and how these associations differ in student groups.

In this study, we used ENA in an introductory computer science (CS1) course to investigate the relationship between computer programming experience and debugging behaviors. To achieve this, we elicited what we called "constituents" of the debugging process—i.e., binary variables deduced from expert observations. Each constituent was operationalized based on how experts interpret students' debugging behaviors and was computed on the code submissions. We investigated debugging behaviors for two types of debugging episodes in which students attempted to fix different types of errors: syntax/checkstyle errors or test errors. ENA point plots and networks were created to explore the difference between students with and without prior experience, and to investigate how debugging behaviors evolved as students gained more experience over three different times periods of the semester. Our ENA highlighted interesting constituent co-occurrences that we investigated through further descriptive and statistical analyses.

Specifically, we asked the following research questions:

1. How does the use of ENA allow us to better understand students' debugging behaviors when solving computer programming problems?
2. How does the debugging behavior of students differ based on their prior computer programming experience?
3. How does debugging behavior evolve over the duration of the semester, as students gain more computer programming experience?

2 Related Work

2.1 Debugging

Students' debugging processes have been investigated through qualitative studies based on think-aloud transcriptions [7], grounded theory [8], interviews [9], and researcher coding [1]. These studies have identified insightful debugging challenges and concepts, such as barriers students encountered corresponding to different debugging phases [7] and debugging strategies articulated by students [8, 9] or experts [1].

Other studies have focused on revealing debugging behaviors with log data. Although programming processes have been modeled from different perspectives such as code updates [10] and program state transitions [11], limited studies have focused on specifically analyzing debugging processes using log data. Ahmadzadeh et al. [12] distinguished groups of good and weak debuggers based on how well they corrected logical bugs. While the study included statistical results, students' debugging behaviors were not analyzed quantitatively to explain differences in their performance. Jemmali et al. [13] considered both error states and programming actions in debugging sequence analysis. However, they only considered four levels of code modifications (no/small/medium/large change) and provided limited interpretations of debugging patterns. Since debugging involves a variety of knowledge and skills, debugging processes should be described with more detailed code-changing behaviors instead of simply how much the code has been changed.

2.2 ENA With Process Data

ENA has been applied to various types of educational process data [14], such as action logs [15, 16], coded discourse [17, 18], and affect observations [19]. Although the original ENA methods do not account for sequential order among learning events, they can extract features from process data that have predictive power on learning performance no worse than sequential analysis methods [18]. Recently, directed ENA has been developed to capture sequential information among learning events [20], and it has shown promise in dissecting MOOC learning tactics [21].

Many studies have used ENA to uncover the co-occurrence patterns among self-regulated learning behaviors [15–17] and to understand collaborative learning processes [17, 18]. Researchers have also utilized ENA in the field of computing education [22–24]. Particularly, Hutchins et al. [23] applied ENA to understand high school students' debugging process during block-based programming. Their result showed that a group characterized by tinkering and evaluation debugging strategies was better at integrating physics and computational thinking concepts than a group characterized by multiple code construction actions without testing.

3 Learning Context

In this study, we analyzed data collected from an undergraduate CS1 course in Java at a Midwestern public university. Students used an online auto-grading system to complete homework, quiz, and exam questions throughout the semester. The course included two large midterms, naturally dividing the semester into three periods. Homework was assigned almost every day, and students could submit unlimited attempts to the online system before the midnight deadline. For each submission, the system first checked if there were any syntax (incorrect Java code) or checkstyle (the code did not follow course specific formatting rules) errors. If so, the system would provide error messages. Otherwise, the platform ran a set of problem-specific test cases to evaluate the submission's correctness. The test cases returned information about the first encountered error or no error message if all cases passed.

3.1 Data

Submission log data of 745 students solving 69 homework questions was collected during the Fall 2019 semester. Each student made an average of 7.49 submissions per question. For each submission, the online learning system recorded the student's submitted code, submission time, and any syntax, checkstyle, or test error messages. Students also completed a pre-course survey providing demographic information and self-reported prior experience in programming. Thirty-one percent of the students were female. We investigated two questions related to the students' prior programming experience. The first one asked which programming languages students were familiar with. We grouped the possible answers into four categories—Java, Java+ (Java and at least one other), other, or none—because we assumed that students with Java experience may learn differently in this introductory Java course. The second question asked students to self-rate their programming abilities on a range from 1 (lowest) to 5 (highest).

3.2 Eliciting Constituents

Process log data, such as code submissions, contain rich information that can be used to better understand students' debugging behaviors. However, log data first needs to be pre-processed to compute features that summarize debugging behaviors and filter out irrelevant information. To identify relevant and interpretable features, we elicited binary "constituents" of experts' interpretation of debugging behaviors, each operationalizing an element of debugging behavior that experts identified as meaningful.

We identified constituents using a similar method to the one used by Paquette, de Carvalho, & Baker [25]. The elicitation process involved conducting interviews with programming experts and extracting relevant components of their interpretation of students' code submissions. This section provides an overview of the expert interview process and describes how we identified a specific set of constituents for our study.

We recruited two programming experts, both computer science graduate students with a research focus on computer science education who also had prior teaching or tutoring experience. These experts were expected to possess a deep understanding of Java programming concepts and be familiar with common student misconceptions and debugging strategies. Both experts were presented with the same set of pre-selected examples of student problem solving submissions and were asked to comment on how students approached solving the problems and the debugging strategies they employed. In total, the first expert commented on 20 problems for a duration of 6.18 h, and the second expert commented on 24 problems for a duration of 9.82 h.

To identify key constituents of the experts' interpretations, we first constructed a flowchart representing their approach to interpreting code submissions, based on the recurring themes identified from the interview recordings. The flowchart was reviewed to identify missing components from the experts' interpretation approach, which lead to further review of the interviews. Through this iterative process, we generated a comprehensive list of constituents that captured the essence of the experts' interpretation. Finally, we computed these constituents from the code submission data.

Table 1 presents a list of all the constituents used in the analyses presented in this paper. These constituents represent various aspects of the debugging process, including changes to the submitted code and progress towards fixing errors. It is important to note that some of the elicited constituents, such as the use of print statements, were excluded from our analyses. While print statements can be used during debugging, some problems (especially early in the semester) required students to use print statements as part of the solution. The constituents were removed from the analyses to avoid confounding print statements that were part of the solution with those that were used for debugging.

Table 1. Debugging constituents based on the experts' interpretation.

Constituents	Operationalization
MASSIVE DELETION	The student deleted 4 or more lines of code, and at least 30% of all the lines from the previous submission were deleted
SUBMISSION UNDO	This is not the first submission, and the code is the same as a submission at least 2 submissions back
DELETED LINE W/ ERROR	At least one line with a checkstyle/syntax error from the previous submission was deleted
MODIFIED LINE W/ SYNTAX ERROR	At least one line with a syntax error from the previous submission was modified
CHANGED VAR NAME	The name of a variable was modified
REPEATED CHANGE	At least one identical change (deletion/addition/modification) was made on multiple lines. E.g., submission added the same word on two or more lines
IGNORED CHECKSTYLE BUT ADDRESSED SYNTAX	Lines with syntax errors in previous submission were altered but lines with checkstyle errors were not
REDUCED SYNTAX ERRORS	The current submission has less total syntax errors than the previous submission
REDUCED CHECKSTYLE ERRORS	The current submission has less total checkstyle errors than the previous submission
CHANGE IN TEST ERROR	The provided test error for the current submission is different than the test error that was provided from the previous submission
NEW ERROR	The current submission is not the first and has at least 1 error that was not present in the immediately preceding submission
REPEATED NEW ERROR	At least 1 error identified as new (NEW ERROR) is the same as one of the errors in the previous submissions

4 Epistemic Network Analysis

4.1 Methods

Data Preparation. To focus our analyses on how students debugged specific types of errors, we split our data into debugging episodes that exclusively kept submissions with either syntax/checkstyle errors or test errors. We settled on this approach after a pilot study revealed interpretation issues when keeping all error types together. This is due to the design of the learning platform, which only ran problem-specific tests when no syntax/checkstyle errors were present.

In addition, because debugging is a process that unfolds over multiple steps, students who solved problems in few attempts may not be provided with adequate opportunities to demonstrate their debugging process. As such, only episodes with a number of submissions in the 75th percentile or above were included, with a minimum submission threshold of 5 for syntax/checkstyle error episodes and 6 for test error episodes.

We calculated the constituents separately for each episode. We also included relevant grouping data to make our analysis possible: answers to survey questions about prior programming ability and experience with programming languages, as well as information about which third of the semester each problem was assigned to. For each set of episodes, we removed constituents that were not relevant to its type of error. For example, CHANGE IN TEST ERROR was not relevant to syntax/checkstyle errors, whereas REDUCED CHECKSTYLE ERRORS was not relevant to test errors.

Table 2. Number of debugging episodes for each grouping.

	Programming language (T1)				**Prior ability (T1)**					**Time**		
	None	**Other**	**Java**	**Java+**	**1**	**2**	**3**	**4**	**5**	**T1**	**T2**	**T3**
syntax/ checkstyle	832	954	520	1260	804	1357	1064	289	52	3576	4756	4439
test	174	491	237	273	209	413	403	133	17	1178	1485	3191

Epistemic Network Analysis. ENA was conducted using the rENA package [26]. We compared the networks obtained using two different window sizes (2 and 3) and found no meaningful differences. We therefore decided to use a window size of two for our analyses, which we reasoned would be easier to interpret as co-occurrences would be from either immediately adjacent submissions or from the same submission.

We created ENA point plots and networks for each type of debugging episode for each grouping dimension. We also created a separate mean epistemic network for each group, as well as plots showing the difference between pairs of mean group networks. Welch t-tests (Table 3) were conducted to identify statistically significant differences between networks. Table 2 provides detailed information about the number of data points included in each grouping. Since each of our grouping categories included more than two groups, we used singular value decomposition rather than means rotation.

When preparing the plots for prior programming ability and prior language experience groupings, we kept only problems that were part of the first third of the semester (T1). We reasoned that prior experience would have the most impact on debugging behaviors of students early on, but that this effect would decrease with in-course experience, which may equalize students with different prior levels of experience.

This series of plots allowed us to analyze the various co-occurrences between constituents, allowing us to answer our research questions on the impact of experience on

Table 3. Results of Welch t-tests for previously known languages (top), starting programming abilities (left), and semester time (right). In each cell, syntax/checkstyle debugging episodes are on top and test debugging episodes on bottom *in italics*. $^{*}p < 0.05$, $^{**}p < 0.01$, $^{***}p < 0.001$.

Language	Java+	Java	Other
None	6.56*** *1.48*	4.22*** *1.04*	2.60** *0.51*
Java+		0.86 *0.16*	3.90*** *0.97*
Java			2.15* *0.61*

Ability	1	2	3	4
2	1.34 *0.58*			
3	3.53*** *0.23*	2.55* *0.43*		
4	4.28*** *0.97*	3.61*** *0.59*	1.96 *0.89*	
5	2.72** *0.91*	2.39* *1.12*	1.72 *1.00*	0.85 *1.31*

Semester time	T1	T2
T1		20.17*** *3.25***
T3	18.71*** *4.10****	1.51 *0.47*

debugging practices. We also used ENA to identify follow-up questions that warranted further exploration of our data.

4.2 Results

Prior Experience. We compared the debugging-related actions in the first third of the semester (T1) of students with different levels of prior experience. We found that the epistemic networks for students subdivided by either previously known programing languages or self-reported experience level yielded very similar trends—that is, students with lower self-rated ability followed the same general debugging patterns as those with little or no prior knowledge of programming languages when compared with those who either rated their abilities higher or had more experience with languages such as Java. This raised the question of how often these two survey dimensions intersect, which we explored further in a second round of analysis (Follow-Up Question 1).

Debugging Syntax/Checkstyle Errors. The mean networks for syntax/checkstyle debugging episodes revealed that, along the x-axis, checkstyle-focused constituents were located on the left, whereas syntax-focused constituents were on the right, and constituents related to general code edits were closer to the center (Fig. 1, left). Overall, the epistemic networks for novice students were more heavily weighted on the right and more experienced students more on the left, across both starting ability level and previously known languages (Fig. 1, right). Together, these observations suggest that novice students showed more co-occurrences of debugging behaviors addressing syntax errors, whereas more experienced students showed more co-occurrences related to checkstyle errors.

We found further evidence of this when comparing the mean networks of students with different levels of prior ability (Fig. 1, left). Experienced students (self-reported

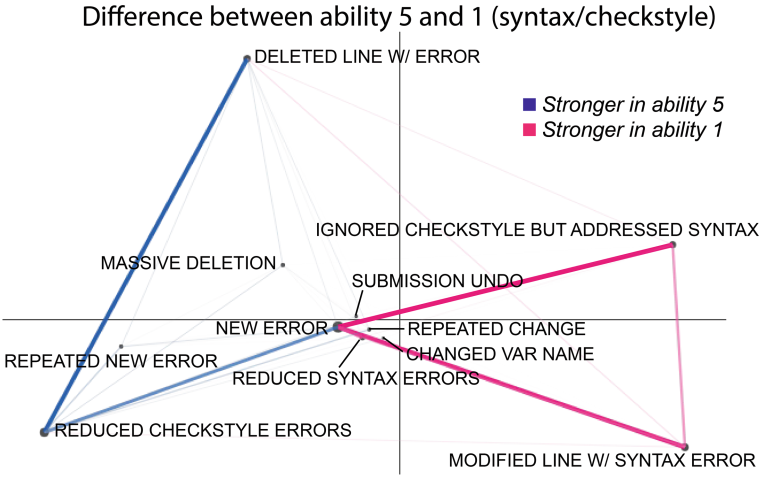

Fig. 1. Difference network between ability levels 5 and 1 in first third of semester (left); Means and confidence intervals across ability levels and prior know languages (right).

ability of 5) had stronger co-occurrences between NEW ERROR and REDUCED CHECKSTYLE ERRORS than novice students (self-reported ability of 1), as well as between DELETED LINE W/ ERROR and REDUCED CHECKSTYLE ERRORS.

One potential explanation is that experienced students may have simply encountered less syntax errors and, as such, encountered proportionally more checkstyle errors. Alternatively, experienced students could be addressing both syntax and checkstyle errors within the same submission. We decided to pursue these questions further in our second round of analysis (as Follow-Up Question 2) by exploring the distribution of checkstyle vs. syntax errors and the order in which students fixed checkstyle vs. syntax errors (checkstyle>syntax, syntax>checkstyle, or simultaneously).

With novice students, we found the opposite. They had many more co-occurrences between NEW ERROR and IGNORED CHECKSTYLE BUT ADDRESSED SYNTAX when both were present. Echoing some of our previous questions, this could be explained by experienced students simply making less syntax errors, giving them less opportunities for this constituent than novice students. There is some evidence for this in the fact that novice students had more co-occurrences between NEW ERROR and MODIFIED LINE W/ SYNTAX ERROR than experienced students.

Debugging Test Errors. We again found that the mean epistemic networks for less experienced students were more heavily weighted on the left and vice versa across the two dimensions of prior experience (self-rated ability levels and familiarity with various programming languages). There was one exception where the most advanced ability level (5) conspicuously did not follow this pattern (Fig. 2, right). However, given this group's much larger confidence interval, along with the consistency in the network graphs between the two experience dimensions aside from this group, we did not include the advance ability level further in our analyses at this stage, instead choosing to set its significance aside for later investigation as Follow-Up Question 3.

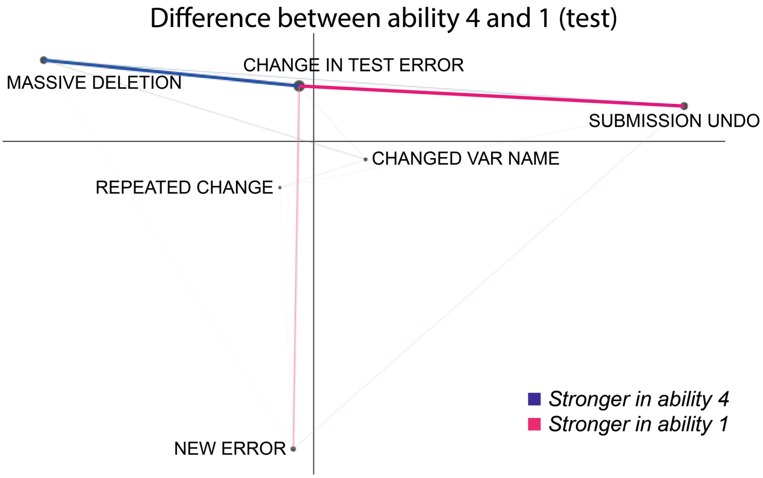

Fig. 2. Difference network between ability levels 4 and 1 for test error debugging (left); Means and confidence intervals across ability levels and prior know languages (right).

We found that experienced students had more co-occurrences between MASSIVE DELETION and CHANGE IN TEST ERROR than novice learners. A possible explanation is that experienced students are more confident in their abilities and make more changes to the code before submitting. However, this hypothesis should be verified in future studies.

As explained previously, one limitation of our dataset is that only a single test error can be identified in each submission, whereas all syntax/checkstyle errors are presented. This means that the NEW ERROR constituent is only measuring new syntax/checkstyle errors. In test error debugging episodes, this could only occur in the very last submission, indicating that a student's code had no more test errors but introduced new syntax/checkstyle errors. We found that novice students have more co-occurrences between this action and CHANGE IN TEST ERROR than experienced students. This may suggest that novices are more likely to make new syntax/checkstyle errors while debugging for test errors, though in this case we're only capturing that information at the very end of a test error debugging episode.

Changes Across Semester. Here we report on the variations in debugging patterns that we observed using ENA to compare students at the beginning (T1), the middle (T2), and the end (T3) of the semester, regardless of prior experience.

Debugging Checkstyle and Syntax Errors. Flipping the trend we observed earlier, where students with more prior experience showed more connections to checkstyle errors, it appears that, as the semester progresses and they develop more experience with programming and the expectations of the course, students increasingly spend more time on syntax errors and less on checkstyle errors. Figure 3 (left) reveals that the majority of co-occurrences between NEW ERROR and REDUCED CHECKSTYLE ERRORS take place during the first third of the semester. Figure 3 (right) further highlights how different the early part of the semester is from the latter two thirds.

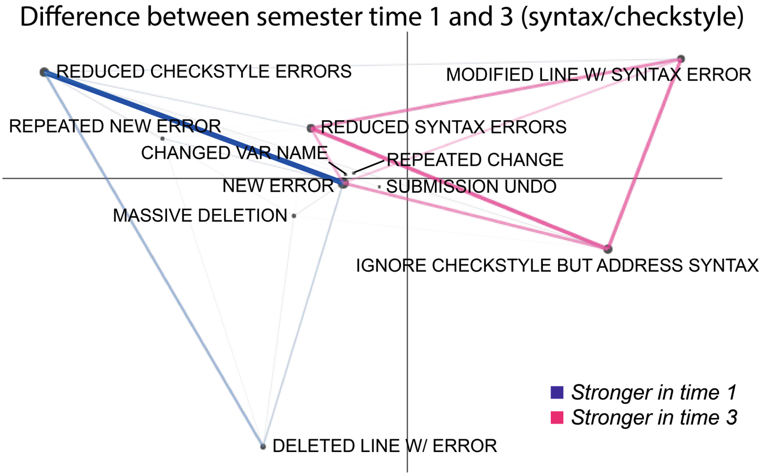

Fig. 3. Syntax/checkstyle debugging episodes. Difference network between semester time 1 and 3 (left); Means and confidence intervals across semester times (right).

The strong co-occurrences in the latter part of the semester with the constituent IGNORED CHECKSTYLE BUT ADDRESSED SYNTAX suggest that this may not simply be a case of students making less checkstyle errors, but rather of students consciously choosing to fix syntax errors first. We explore this further as Follow-Up Question 4.

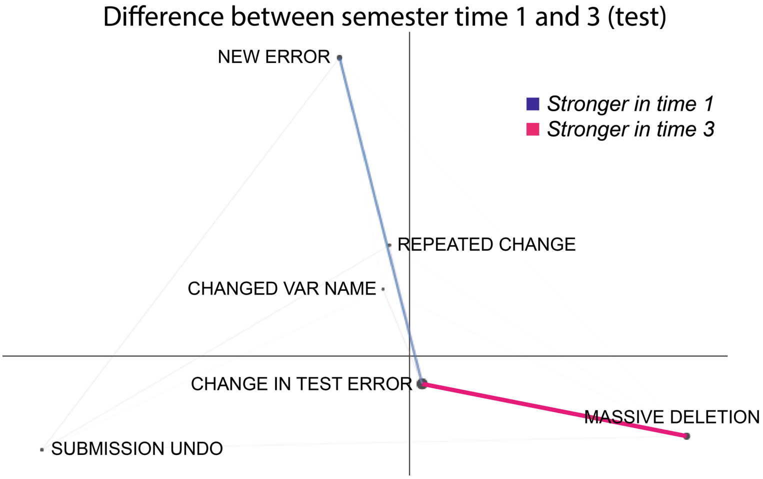

Fig. 4. Test debugging episodes. Difference network between semester time 1 and 3 (left); Means and confidence intervals across semester times (right).

Debugging Test Errors. With debugging for test errors, it is the last third of the semester that stands out as unique (Fig. 4, right). During these later weeks, students have many more co-occurrences between MASSIVE DELETION and CHANGE IN TEST ERROR. This is consistent with what we found among more experienced students during the early part of the semester, suggesting that experience leads to this debugging pattern.

The first two thirds of the semester included more co-occurrences between CHANGE IN TEST ERROR and NEW ERROR than the last third. This is similar to what we found among novice learners when compared with more experienced ones early in the semester, suggesting a consistent trend. Because of the nature of our data, this co-occurrence can be interpreted as students making new syntax/checkstyle errors while debugging a test error and either giving up or successfully solving the problem without making more test errors. We explored this further in Follow-Up Question 5.

5 Follow-Up Questions from ENA

The ENA analyses provided insights into the students' debugging behaviors. It highlighted ways in which debugging behaviors varied based on prior programming experiences and how debugging behaviors evolved throughout the semester. However, it also raised questions that could not fully be answered by examining the co-occurrence networks by themselves. In this section, we investigate each of these questions.

Follow-Up Question 1. Through our use of ENA, we found that grouping debugging episodes using the two survey dimensions designed to measure experience provided very similar observations of debugging behaviors. We asked whether this was because students who self-rated as more capable were also those who listed prior knowledge of more programming languages. As part of a follow-up analysis, we calculated the Spearman's rank correlation between these two categories and found a moderate relationship for student in both the syntax/checkstyle error dataset ($r_s = 0.63, p < 0.001$) and the test error dataset ($r_s = 0.63, p < 0.001$). A confusion matrix revealed, however, that the majority of students—especially those with experience in Java and other languages—rated their abilities at the midpoint of 3 (Fig. 5). Because we are interested in comparing more vs. less experienced students, this neutral option makes it difficult to categorize students. We found that by removing all students in ability level 3, the Spearman correlation became stronger in both the syntax/checkstyle and the test error datasets ($r_s = 0.71, p < 0.001$ and $r_s = 0.71, p < 0.001$, respectively). For the rest of our follow-up analysis, we define experienced students as those with prior Java experience (*Java* or *Java+*) AND who rated their abilities as either 4 or 5. Similarly, we define novice students as those with either no prior known programming languages (*none*) or with only languages other than Java (*other*) AND who rated their abilities as either 1 or 2. This fits with how we analyzed these categories during our ENA as well.

Follow-Up Question 2. Another group of questions that arose during ENA were those regarding the connection between NEW ERROR and REDUCED CHECKSTYLE ERRORS. By looking at the frequency at which experienced students simultaneously fixed checkstyle and syntax errors (65 times) vs. performing these actions one after the other (310 times), we found that a certain trend we discovered via ENA—that experienced students placed more emphasis on checkstyle errors than syntax errors in comparison with novice students—was not simply a case of simultaneous debugging after all. We also found that experienced students had a ratio of checkstyle to syntax errors of about 1.71 (1.62 checkstyle and 0.95 syntax errors on average per submission), whereas novice students had a much smaller ratio of about 0.69 (0.99 checkstyle and 1.43 syntax errors on average per submission). Since the average number of syntax and checkstyle errors

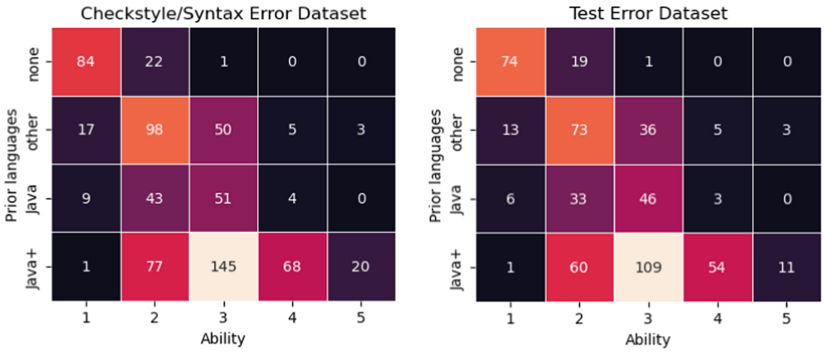

Fig. 5. Confusion matrix for prior ability and prior programming languages for both datasets: checkstyle/syntax and test debugging (T1 only). Lighter color indicates more instances.

per submission was similar between the groups (2.57 vs. 2.42), such a stark contrast in ratio seems to confirm that the trend we discovered was due to experienced students making more checkstyle errors while novices made more syntax errors.

Follow-Up Question 3. In our ENA, we found that students who self-rated their abilities as 5 (the highest) did not fit the trend set by the other ability levels. We investigated this and discovered that, as suspected, these students accounted for a disproportionally small subset of students (4%). As such, it is unclear whether the unexpected observation that high-ability students had more co-occurrences between SUBMISSION UNDO and CHANGE IN TEST ERROR than other students is simply due to the lower sample size.

Follow-Up Question 4. Some trends in our ENA led us to ask whether students made less checkstyle errors in the latter parts of the semester or if the proportion stayed the same, but they simply chose more often to ignore these and fix their syntax errors first. We found that the number of checkstyle errors per submission indeed decreased drastically after the first third of the semester, going from 1.18 to 0.43, and then only slightly increased to 0.73. We also found that the number of syntax errors per submission progressively increased from 1.27 to 1.67 and then to 2.07. These observations reveal a significant shift in the types of errors that students struggled with as they gained experienced. They also leave room for an acknowledgement of the shift in debugging strategy we observed in which students, as they gained experience, more often chose to focus on syntax errors first, even when they also had checkstyle errors present.

Follow-Up Question 5. Our ENA revealed that it was more common during the first two thirds of the semester for students to make a syntax/checkstyle error while debugging a test error and not receive any more test error messages in that problem. However, it was unclear if this was because students in these cases fixed both the new errors and the previous test error simultaneously, or if they simply gave up after the frustration of encountering a new type of error during the process of debugging. We found that in most cases—about 92%, regardless of the time in the semester—these students indeed solved the problem, indicating that they simultaneously fixed both types of errors.

6 Discussion and Conclusion

In this study, we elicited and calculated debugging constituents from episodes where students fixed syntax/checkstyle errors or test errors. We used ENA to compare the debugging behaviors of students based on their prior programming experience and to study how debugging behaviors evolved as students gained more programming experience throughout the semester. Results of our analyses showed how novices and experienced programmers put different emphasis on fixing checkstyle or syntax errors. The differences between co-occurrences of debugging patterns in the epistemic networks we generated raised further questions about the causes of such differences and about how debugging behaviors may shift as students gain more experience. We highlight our findings in the following paragraphs.

First, ENA for the two pre-course survey questions (self-rated programming ability level and previously known programming languages) showed similarities in the debugging behaviors of novice and experienced programmers in the first third of the semester. Experienced students placed more emphasis on fixing checkstyle errors while novice students focused more on syntax errors. An analysis of the distribution of syntax and checkstyle errors showed that this may largely be due to experienced students making less syntax errors (and proportionally more checkstyle errors). This may be related to the experienced students' greater familiarity with the Java language and its syntax and their lack of knowledge of the checkstyle rules specifically enforced in the course.

Second, as students gained more experience through the semester, they made significantly less checkstyle errors, causing them to focus more on fixing syntax errors. Even when they encountered checkstyle errors, they more often chose to tackle syntax errors first. This may be the result of an increase in problem complexity or the continuous introduction of new concepts with new syntax as the semester progressed. The change in programming concepts and problems can create more occasion for making syntax errors or encourage students to focus first on big problems (syntax errors) and worry about the details (checkstyle errors) later. During the constituent elicitation interviews, one of the experts explicitly commented on how students should ignore checkstyle errors until they fixed all syntax errors. As such, this may be evidence of students learning better debugging strategies with experience. Another potential factor could be the order in which errors are presented to students. Checkstyle errors are always put above syntax errors, so students might follow the displayed order to fix errors in earlier parts of the semester when still becoming familiar with the system. This suggests designing auto-grading systems to downplay less critical checkstyle rules so as to highlight the importance of addressing syntax and test errors first.

Third, similar debugging patterns for test errors appeared among more experienced students, whether they came to the class with more prior experience or gained additional experience throughout the semester. More experienced students more often performed massive deletions that led to changes in test errors. It is unclear why this is the case, as performing massive deletion doesn't appear, on the surface, to be an efficient debugging strategy. It may be evidence of increased confidence in deciding when to make significant—instead of incremental—changes to code. But further analyses will be required to better understand this phenomenon. More experienced students also finished their test debugging episodes attempting to fix syntax/checkstyle errors less often. While this didn't appear to prevent less experienced students from successfully solving problems, having to fix both types of errors simultaneously might slow down the debugging process. This suggests that it might be helpful to remind more novice students to focus primarily on test errors instead of being distracted by syntax/checkstyle errors.

While the use of ENA in this study revealed interesting findings about debugging patterns, existing limitations call for improvements in future work. First, the context in which our dataset was collected may have influenced the type of debugging behaviors that we observed. Unfortunately, the online homework that was used by the students was limited to displaying only one test error at a time. Because of this, it may have been difficult for students to assess whether they were making progress towards the correct solution when debugging test errors. It also limited us in our analyses of how much progress students were making towards the correct solution. Second, the self-reported survey data posed some difficulties in defining student experience groups. Students exposed to more programming languages did not consistently rate themselves at a higher ability level, as stated in Follow-Up Question 1. More objective measures of prior experience would help avoid such bias. Third, while ENA networks served as an insightful tool to explore debugging patterns in different student groups, the connection between constituents can only represent co-occurrence without showing which constituent comes before another. Given that debugging is a sequential process, this ambiguity caused difficulties in interpreting some behaviors and their causes. Directed ENA networks [20] could be applied to account for the sequential aspects of ENA. Fourth, additional qualitative analyses can be combined with ENA to explain the co-occurrence between constituents. For example, we found that the connection between massive deletion and a change in test error appeared more often for more experienced students. This result was unexpected because we would assume incremental changes to be more efficient during debugging, in most cases. Further analyses will be needed to qualitatively examine the submission log data and better understand this behavior.

References

1. Murphy, L., Lewandowski, G., McCauley, R., Simon, B., Thomas, L., Zander, C.: Debugging the good, the bad, and the quirky-a qualitative analysis of novices' strategies. ACM SIGCSE Bull. **40**(1), 163–167 (2008)
2. Kessler, C., Anderson, R.: A model of novice debugging in LISP. In: Proceedings of the First Workshop on Empirical Studies of Programmers, Ablex, Norwood, NJ (1986)
3. Begum, M., Nørbjerg, J., Clemmensen, T.: Strategies of novice programmers. In: Proceedings of the 41st Information Systems Research Seminar in Scandinavia (IRIS), Odder, Denmark (2018)
4. Perkins, D., Martin, F.: Fragile knowledge and neglected strategies in novice programmers. In: Soloway, E., Iyengar, S. (eds.) Empirical Studies of Programmers, pp. 213–229. Ablex, Norwood, NJ (1986)
5. Fitzgerald, S., et al.: Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. Comput. Sci. Educ. **18**(2), 93–116 (2008)
6. Shaffer, D.W., Collier, W., Ruis, A.R.: A tutorial on epistemic network analysis: analyzing the structure of connections in cognitive, social, and interaction data. J. Learn. Anal. **3**(3), 9–45 (2016)
7. Liu, Z., Zhi, R., Hicks, A., Barnes, T.: Understanding problem solving behavior of 6–8 graders in a debugging game. Comput. Sci. Educ. **27**(1), 1–29 (2017)

8. Fitzgerald, S., Simon, B., Thomas, L.: Strategies that students use to trace code: an analysis based in grounded theory. In: Proceedings of the 2005 International Workshop on Computing Education Research (ICER'05), pp. 69–80. ACM, Seattle, USA (2005)
9. Fitzgerald, S., McCauley, R., Hanks, B., Murphy, L., Simon, B., Zander, C.: Debugging from the student perspective. IEEE Trans. Educ. **53**(3), 390–396 (2010)
10. Blikstein, P., Worsley, M., Piech, C., Sahami, M., Cooper. S., Koller, D.: Programming pluralism: using learning analytics to detect patterns in the learning of computer programming. J. Learn. Sci. **23**(4), 561–599 (2014)
11. Berland, M., Martin, T., Benton, T., Petrick Smith, C., Davis, D.: Using learning analytics to understand the learning pathways of novice programmers. J. Learn. Sci. **22**(4), 564–599 (2013)
12. Ahmadzadeh, M., Elliman, D., Higgins, C.: An analysis of patterns of debugging among novice computer science students. ACM SIGCSE Bull. **37**(3), 84–88 (2005)
13. Jemmali, C., Kleinman, E., Bunian, S., Almeda, M.V., Rowe, E., Seif El-Nasr, M.: MAADS: mixed-methods approach for the analysis of debugging sequences of beginner programmers. In: Proceedings of the 51st ACM Technical Symposium on Computer Science Education, pp. 86–92. ACM, Portland, OR, USA (2020)
14. Elmoazen, R., Saqr, M., Tedre, M., Hirsto, L.: A systematic literature review of empirical research on epistemic network analysis in education. IEEE Access **10**, 17330–17348 (2022)
15. Li, S., Huang, X., Wang, T., Pan, Z., Lajoie, S.P.: Examining the Interplay between self-regulated learning activities and types of knowledge within a computer-simulated environment. J. Learn. Anal. **9**(3), 152–168 (2022)
16. Paquette, L., Grant, T., Zhang, Y., Biswas, G., Baker, R.: Using epistemic networks to analyze self-regulated learning in an open-ended problem-solving environment. In: Proceedings of the 2nd International Conference on Quantitative Ethnography, pp. 185–201 (2021)
17. Melzner, N., Greisel, M., Dresel, M., Kollar, I.: using process mining (PM) and epistemic network analysis (ENA) for comparing processes of collaborative problem regulation. In: Proceedings of the 1st International Conference on Quantitative Ethnography (ICQE 2019), pp. 154–164. Springer, Cham, Madison, WI, USA (2019)
18. Swiecki, Z., Lian, Z., Ruis, A., Shaffer, D.: Does order matter? Investigating sequential and cotemporal models of collaboration. In: Proceedings of the 13th International Conference on Computer Supported Collaborative Learning (CSCL), pp. 112–119. ISLS, Lyon, France (2019)
19. Karumbaiah, S., Baker, R.S.: Studying affect dynamics using epistemic networks. In: Proceedings of the 2nd International Conference on Quantitative Ethnography, pp. 362–374. Springer, Malibu, USA (2021)
20. Fogel, A., et al.: Directed epistemic network analysis. In: Proceedings of the 2nd International Conference on Quantitative Ethnography, pp. 122–136. Springer, Malibu, USA (2021)
21. Fan, Y., et al.: Dissecting learning tactics in MOOC using ordered network analysis. J. Comput. Assist. Learn. **39**(1), 154–166 (2022)
22. Arastoopour Irgens, G., et al.: Modeling and measuring high school students' computational thinking practices in science. J. Sci. Educ. Technol. **29**, 137–161 (2020)
23. Hutchins, N.M., et al.: Analyzing debugging processes during collaborative, computational modeling in science. In: Proceedings of the 14th International Conference on Computer-Supported Collaborative Learning, pp. 221–224 (2021)
24. Xu, W., Wu, Y., Ouyang, F.: Multimodal learning analytics of collaborative patterns during pair programming in higher education. Int. J. Educ. Technol. High. Educ. **20**(1), 1–20 (2023)
25. Paquette, L., de Carvalho, A., Baker, R.: Towards understanding expert coding of student disengagement in online learning. In: Proceedings of the 36th Annual Meeting of the Cognitive Science Society, pp. 1126–1131, Québec City, Canada (2014)
26. Marquart, C., Swiecki, Z., Collier, W., Eagan, B., Woodward, R., Shaffer, D.W.: rENA (0.2.4) [R package] (2022). https://cran.r-project.org/web/packages/rENA/index.html