Applying Sequence Analysis to Understand the Debugging Process of Novice Programmers

Qianhui Liu[1] and Luc Paquette[1]

[1] *University of Illinois Urbana-Champaign*

Abstract

Debugging is a challenging task for novice programmers that requires diverse skills as well as iterative practice to find and fix the cause of errors. In this study, we analyzed submission log data to investigate the temporal aspects of the debugging process. We extracted debugging episodes and interpretable debugging constituents – components of experts' interpretations of debugging behaviors – from the data collected from an undergraduate CS1 course. We first applied sequential pattern mining and state transition metrics to examine how debugging constituents occur one after another. We further applied temporal interestingness techniques to reveal the occurrence of debugging constituents within each episode and compared the difference in constituent patterns across the semester. We further investigated how the sequential ordering of debugging constituents changed over the course of the semester. Our findings suggest that novice programmers exhibit frequent printing and submission undo no matter in which phase of the semester. As they proceeded toward the end of the semester, they employed more repetitive printing to understand the problem and long planning before implementation.

Keywords

Debugging process, sequence analysis, debugging constituents

1. Introduction

Debugging is a computational practice in programming that occurs when programmers test their code to find out exactly where the error is and how to fix it [26]. It is a challenging task for novice programmers to learn as they usually do not yet possess a comprehensive knowledge of programming [4] and the strategic skills to gain control of the programming process [30]. Different from general programming skills, debugging focuses on a systematic search for the source of a bug and its removal [19, 20], which is referred to as troubleshooting or problem-solving. As one of the essential problem-solving skills in programming [13], debugging calls for individual pedagogies [28]. Studies examining the debugging process show how learning debugging is especially useful to equip students with the necessary problem-solving skills to be successful at programming and that those skills can further transfer to non-programming domains [25].

Although debugging is an important and challenging topic in CS education, existing log data analyses have shown limited interpretation of the debugging strategies students might be using in the process. Researchers have used aggregated features to describe changes between submissions [2, 36], but their findings are not fine-grained enough to reveal the inner process of debugging. Others have applied sequential analysis methods to code sequences [5, 15]. The results showed the potential of such methods to reveal latent patterns of students' debugging behaviors, but their interpretations are often intrinsic and do not explicitly connect with instructors' or students' thinking, limiting the actionable implications that can be drawn for them.

In this study, we combined sequence analysis and interpretable debugging features to understand students' debugging process with log data. We first extracted debugging episodes and indicators of debugging actions, that we called constituents, from submission log data collected in an introduction to computer science (CS1) course. We applied sequential pattern mining, calculated transition metrics of constituent patterns, and compared temporal features across semester. Following the definition in [26] that debugging focuses more on how students fix logic and semantic errors in programs, this study is more specifically interested in fixing test errors (rather than syntax or checkstyle errors). In this regard, we answer the following research questions:

RQ1: How can sequence analysis help to understand novice programmers' debugging process?

RQ2: How does novice programmers' debugging process change across different phases of the semester?

CSEDM'24: 8th Educational Data Mining in Computer Science Education Workshop, July 14, 2024, Atlanta, GA

✉ ql29@illinois.edu (Q. Liu); lpaq@illinois.edu (L. Paquette)

2. Related work

2.1. Log data analysis of debugging

Previous work has looked at programming submission log data through different perspectives, mainly two types of information are generally extracted: error messages and code editions. For example, [2] focused on the changes of syntax and semantic errors in submissions to distinguish good and bad debuggers. [11] used students' code content and mapped it to correct solutions to measure their progress and detect struggling moment.

Among the submission log analysis of debugging, there has been a growing interest in studying its temporal features through data-driven methods. Studies have identified different sequential patterns of debugging behaviors, as the basis to reveal how different strategies may manifest themselves through the data. The most popular method is sequential pattern mining. For example, [22] conducted a sequential pattern analysis to obtain transition diagrams of programming behaviors between students in flow, anxiety, and boredom. Similar transitions have been identified to predict students' final performance [15], reveal collaborative problem-solving process [36], distinguish programming styles among students [6], or be further summarized as interpretable pathways such as exploring, tinkering, and refining [5]. Other temporal methods have also been explored to analyze the dynamic process of programming. Blikstein et al. [6] applied dynamic time warping to measure the distance between students' code update sequences in their first and last assignments. The distance correlated with students' assignment performance and exam grades, indicating students with higher grades would also change their programming patterns the most. Jemmali et al. [18] visualized the state graph and sequence graph of programmers' error states and actions on each submission to distinguish students' debugging techniques.

Although many studies have focused on sequential analysis of debugging on submission level, their input features were mostly simple error or code-editing frequency. For example, in [18] only four levels of code modifications (no, small, medium, and large change) were included to describe students' actions. The nuance between different types of code modifications was ignored, such as whether they modified printing, comments, or common code lines, whether they added, deleted, or modified code lines, and how long the submission intervals were. The omission of such details limited the interpretations of the debugging strategies students might be employing.

2.2. Towards interpretable debugging process analysis

Besides data-driven approaches, some researchers investigated debugging process in a more meaning-driven way. They first regarded debugging as a problem-solving process and constructed interpretable features with related theories. For example, Liu et al. [24] used the five-stage problem-solving model [8] (identifying the problem, representing the problem, selecting a strategy, implementing the strategy, and evaluating solutions) to conceptualize the possible programming barriers students may encounter. They then proposed corresponding measurable behaviors, such as editing code line where the program stopped instead of where the real issue was, and manually labeled these behaviors with snapshots of students' code in a debugging game.

Most of the interpretable debugging analysis required a coding scheme of students' behaviors to start with. Chao [9] proposed quantitative indicators of problem-solving process, such as nested iteration and problem decomposition, by counting the control flow blocks used by students in a robot-building game. Pellas and Vosinakis [29] coded students' think-aloud transcriptions with the same coding scheme to understand the effect of simulation games on programming. Liu et al. [22] applied a different perspective with five types of problem-solving behaviors including solution development, experiment, solution review, solution reuse, and reading the tutorial. Jayathirtha et al. [17] coded videos of pair debugging into four problem-solving strategies, forward reasoning to isolate the problem, backward reasoning to isolate the problem, hypothesis & solution generation, and verification & testing.

To bring interpretation to the features, many studies need manual coding of submissions. Although these features brought much insight on how and why students debug in different pathways, they are challenging to apply to a larger scale dataset. In this study, we aim to combine both data-driven sequence analysis approach and interpretable debugging features, as a way to move towards more interpretable debugging process analysis.

3. Methodology

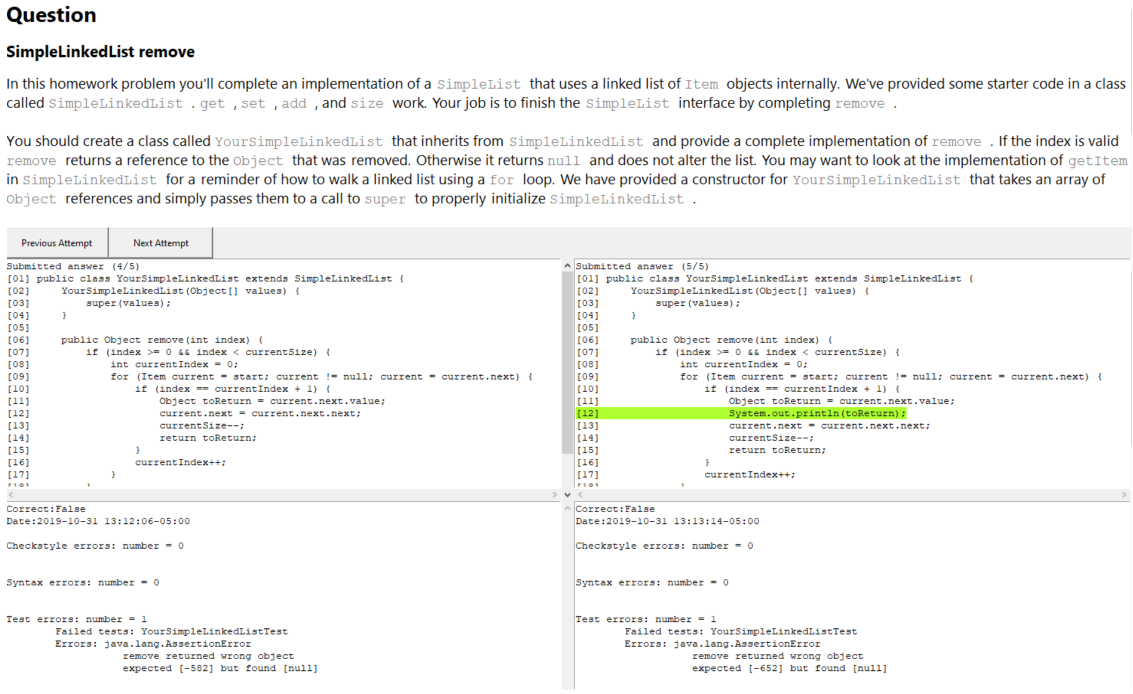
3.1. Dataset

Submission log data was collected from a CS1 course where undergraduates learned to program in Java. Throughout the semester, a homework question was assigned to students almost every day and students had to finish it by making submissions on an auto-grading system before midnight (students were allowed an unlimited number of submissions to achieve the correct answer). For each submission, the system would first check if there were any checkstyle (the code is incorrectly formatted according to the course's guidelines) or syntax (the submission cannot be executed because it contains incorrect Java code) errors in the code. If there were, the system would display error messages without executing the code. Otherwise, students would receive test errors (the submission was executed but was not a valid solution) specifying the error type and error details such as input and expected output. We collected submission log data from the Fall 2019 semester including the code content, error messages, and timestamps. In total, 707 students (31% female, 28% majored in engineering) solved 65 homework questions, with an average of 8.6 submissions per student per question. Furthermore, the course was naturally divided into three phases (t1, t2, t3) separated by midterm exams.

3.2. Debugging episodes and debugging constituents

3.2.1. *Extracting debugging episode*

Since the focus of our study was to investigate how students debug test errors, we defined a debugging episode as our unit of analysis. When solving a problem, each submission usually contains one or multiple errors until the final solution. We first discarded submissions reporting syntax or checkstyle errors. Following this, the submissions were further segmented based on the nature of their test errors (note that the platform only reported one test error at a time). We extracted all subsequent submissions with the same type of test error as one debugging episode. Therefore, a debugging episode represented a submission sequence starting from the first appearance of a test error until the student either submitted code that solved the problem or submitted code that resulted in a different test error. A more detailed illustration of the procedure used to extract debugging episodes from the log data can be found in [23].



Question

SimpleLinkedList remove

In this homework problem you'll complete an implementation of a `SimpleList` that uses a linked list of `Item` objects internally. We've provided some starter code in a class called `SimpleLinkedList` . `get` , `set` , `add` , and `size` work. Your job is to finish the `SimpleList` interface by completing `remove` .

You should create a class called `YourSimpleLinkedList` that inherits from `SimpleLinkedList` and provide a complete implementation of `remove` . If the index is valid `remove` returns a reference to the `Object` that was removed. Otherwise it returns `null` and does not alter the list. You may want to look at the implementation of `getItem` in `SimpleLinkedList` for a reminder of how to walk a linked list using a `for` loop. We have provided a constructor for `YourSimpleLinkedList` that takes an array of `Object` references and simply passes them to a call to `super` to properly initialize `SimpleLinkedList` .

```
Submitted answer (4/5)
public class YourSimpleLinkedList extends SimpleLinkedList {
    YourSimpleLinkedList(Object[] values) {
        super(values);
    }

    public Object remove(int index) {
        if (index >= 0 && index < currentSize) {
            int currentIndex = 0;
            for (Item current = start; current != null; current = current.next) {
                if (index == currentIndex + 1) {
                    Object toReturn = current.next.value;
                    current.next = current.next.next;
                    currentSize--;
                    return toReturn;
                }
                currentIndex++;
            }
```

```
Correct:False
Date:2019-10-31 13:12:06-05:00

Checkstyle errors: number = 0


Syntax errors: number = 0


Test errors: number = 1
        Failed tests: YourSimpleLinkedListTest
        Errors: java.lang.AssertionError
                remove returned wrong object
                expected [-582] but found [null]
```

```
Submitted answer (5/5)
public class YourSimpleLinkedList extends SimpleLinkedList {
    YourSimpleLinkedList(Object[] values) {
        super(values);
    }

    public Object remove(int index) {
        if (index >= 0 && index < currentSize) {
            int currentIndex = 0;
            for (Item current = start; current != null; current = current.next) {
                if (index == currentIndex + 1) {
                    Object toReturn = current.next.value;
                    System.out.println(toReturn);
                    current.next = current.next.next;
                    currentSize--;
                    return toReturn;
                }
                currentIndex++;
```

```
Correct:False
Date:2019-10-31 13:13:14-05:00

Checkstyle errors: number = 0


Syntax errors: number = 0


Test errors: number = 1
        Failed tests: YourSimpleLinkedListTest
        Errors: java.lang.AssertionError
                remove returned wrong object
                expected [-652] but found [null]
```

Figure 1: The screenshot of interview tool for visualizing submissions.

Debugging episodes ranged in length from short episodes (3 submissions) where the student resolved an error quickly to very long (62 submissions). To focus our analyses on more representative episodes of debugging behaviors, shorter episodes – where students solve the problem quickly without needing extensive debugging – and longer episodes – where students are struggling for an extended period of time – were excluded. More specifically, only debugging episodes that were within the 25 to 75 percentiles of episode length (4-9 submissions) were retained.

In addition, we excluded episodes collected from questions that required printing an output as part of their solution. This was done because printing can also be used by students to trace and identify the source of an error. To avoid ambiguity about whether a print statement was used as part of a solution or as a tool for debugging, we excluded such questions from our analyses. The excluded questions mainly appeared in the first third of the semester (t1) before students learned about functions and return statements. After preprocessing the data, 3,290 debugging episodes were obtained.

3.2.2. *Eliciting debugging constituents*

In order to obtain interpretable features describing the students' debugging behaviors, we conducted interviews with two programming experts. Both experts were computer science graduate students researching CS education who also had teaching experience and knew students' common misconceptions and debugging strategies. A tool was built to allow experts to visualize students' submissions (Figure 1), with the top showing questions, left showing the previous submission and errors, and right showing the next submission and errors. Changes between the two submissions are highlighted in colors. Experts were presented with the same set of pre-selected submissions and were asked to think aloud while interpreting the students' debugging. We recorded 6 and 10 hours of interview for each expert respectively. A flowchart representing their interpreting approach was then constructed by iteratively identifying missing components from the process and reviewing the interview recordings. More details about the constituents and the process used to elicit them can be found in [31].

In total, thirty-seven constituents were obtained from expert knowledge about fixing checkstyle, syntax, and test errors, but not all constituents could be computed from the submission log data. Therefore, we calculated nine constituents that are relevant to debugging test errors for each submission, as shown in Table 1. Constituents are binary features that indicate whether a submission meets a given criterion. They are not necessarily exclusive to each other, and it is possible for a submission to contain multiple constituents. For some constituents such as *massive deletion or change*, experts did not explicitly say how much is massive. We checked data distribution and set 95 percentiles in numbers of lines deleted or changed as the threshold.

Table 1
Debugging constituents elicited from expert interviews

Constituents	**Description**
submission undo	Returning the code to a previous state
long interval	Submitting code after a break of 2 or more hours
change var name	Changing the name of variables
repeated changes	Making the same change in multiple places
print diff	Printing different outputs
print var	Printing the value of variables
massive deletion	Deleting more than 4 lines or 30% of the code
massive change shorter5	Changed at least 10 lines or 70% of the code in less than 5 minutes since the last submission
massive change longer25	Changed at least 10 lines or 70% of the code with more than 25 minutes since the last submission

3.3. Sequence analysis approach

3.3.1. Sequential pattern mining

Sequential pattern mining (SPM) [1] is a method in educational data mining to uncover frequent patterns within ordered sequences of student problem-solving behaviors. We applied SPM to identify constituents that frequently co-occured in a sequential pattern as one of the ways to answer RQ1. Specifically, we calculated and ranked patterns based on the following three metrics: support, confidence and lift. Each metric provides a different perspective that can be used to interpret the frequent patterns. Support indicates the percentage of all debugging episodes that contained a given pattern. Confidence describes the likelihood of observing the second constituent in a sequence given the first constituent. For a transition X⟶Y, a confidence of 0.5 represents that X is followed by Y 50% of the time. Both support and confidence are commonly reported SPM metrics used to identify the most common sequential patterns. In addition, we computed the pattern's Lift, a measure of pattern interestingness that has been argued to be suitable for educational data and easy to interpret [27]. Lift shows how many times more likely the second constituent in a pattern is to occur given that it is preceded by the first constituent, compared to the average case. For the transition X⟶Y, Lift is the ratio between the likelihood of observing Y when X is observed and the likelihood of observing Y among all transitions. Therefore, a Lift value larger than 1 indicates that Y is more likely to be observed after X compared to average and that X and Y are dependent. The SPM method was conducted using the cSPADE algorithm [34] with Python using the pycspade package, with a maximum gap of 1 to ensure that constituents in the identified patterns occurred in consecutive submissions and a minimum support of 0.01 to include extreme rare patterns. We initially set the maximum length of patterns to be 3 constituents but found length-3 patterns were mostly repetitive print similar to length-2 patterns, so we omitted length-3 results.

3.3.2. Transition metrics

Beyond metrics commonly calculated in SPM, we also considered other transition metrics taking the base rates of constituents into account. For example, the transition X⟶Y may appear to be unusually frequent or infrequent simply because Y is especially common or uncommon. In particular, confidence and lift are metrics that can be influenced by such differences in occurrence [7]. Thus, we complemented our SPM analysis by computing two additional transition likelihood metrics, D'Mello's L [10] and lag sequential analysis (LSA; [12]). The values of L range from negative infinite to 1, with 0 indicating a transition occurring as often as in random-ordered data and higher positive values indicating a stronger dependence between two consecutive constituents. LSA produces a z score where an absolute value larger than 1.96 implies that the transition between two constituents occurs significantly more or less than in random-ordered data. We used a modified version of the code in [7] to calculate the two metrics.

To be noted that since there can be multiple constituents in one submission, our interpretation of the transitions is slightly different from previous sequential analysis studies such as [7]. In common cases, there will be only one state at each timepoint, while for our study, multiple constituents can be identified in the same submission. Therefore, when discussing transitions between constituents such as X⟶Y, we refer to the situations where Y occurs following X, without constraining on other constituents that might occur in the same submission.

To compare the debugging process across different phases of the semester, we conducted ANOVA to compare the L values of constituent patterns in t1, t2, and t3.

3.3.3. Temporal interestingness

Temporal interestingness is a metric to describe variations in a pattern occurrence across time [21]. This technique first segments each of the students' behavioral sequences into *N* ordered bins with equal sizes and counts the occurrence of each pattern in each bin. Then, it calculates the information gain (IG) [33] of pattern occurrences across bins, which measures the extent to which knowing the pattern occurrences informs us about its bin number. Later, [35] proposed an effect-size-based calculation for temporal interestingness by conducting a one-way repeated ANOVA taking the occurrences as the dependent variable and the bin number as the independent within-subject variable. This method returns each pattern's adjusted p value and smaller than .05 indicating significantly interesting.

We segmented debugging episodes into 4 bins to ensure there is at least one submission per bin. We then calculated the occurrences of each constituent in each bin, identified significantly interesting constituents, and visualized them in heatmaps.

4. Results

4.1. RQ1: sequence analysis of debugging

We calculated the support of each constituent to gain a general idea of their occurrences. As shown in Table 2. massive deletion was the most frequent constituent, occurring 1,405 times and occurring at least once in 39% of the episodes. The following frequent constituents are long interval, print var, and print diff. The least frequent constituent is massive change shorter5 that occurred at least once in only 15% of the episodes.

Table 2
The support of debugging constituents

Constituents	**Occurrences**	**Support**
massive deletion	1405	0.39
long interval	949	0.26
print var	939	0.26
print diff	704	0.19
repeated changes	681	0.19
massive change longer25	660	0.18
change var name	590	0.16
massive change shorter5	535	0.15

To investigate the transitions between constituents, we first applied SPM to extract frequent patterns of consecutive constituents and calculated their transition metrics. No z-score computed using *LSA* had an absolute value larger than 1.96. As such, we only report the confidence, lift, support, and *L* values of constituent transitions. Table 3 shows the patterns with the 10 highest support. Patterns with the highest supports are repetitions between *print var* and *print diff.* Their lifts were also larger than one, indicating these transitions were more frequent compared to average cases. The following patterns occurred much less frequently, with supports of 0.06 or lower.

4.2. RQ2: debugging process across semester phases

To answer RQ2, we first compared how debugging constituents distributed across time in three phases of the semester, t1, t2, and t3. As shown in Figure 2, we visualized the occurrence of each constituent in each bin with a single-dimensional heatmap. The value in each bin is its percentage of the total occurrences for that constituent. Deep blue refers to the highest percentage and white refers to zero percentage. We also applied temporal interestingness techniques to identify constituents whose occurrences significantly varied across time and labeled them with * before each heatmap. There was one constituent in t1 that was significantly interesting, five in t2, and six in t3.

Table 3
The confidence, lift and support, and *L* of constituent patterns

Patterns	**Confidence**	**Lift**	**Support**	***L***
print var ⟶ print var	0.85	3.30	0.22	0.48
print diff → print diff	0.86	4.45	0.17	0.58
print var ⟶ print diff	0.60	3.09	0.15	0.24
print diff ⟶ print var	0.79	3.06	0.15	0.40
print diff ⟶ massive deletion	0.29	0.76	0.06	0.04
long interval ⟶ massive deletion	0.10	0.27	0.03	-0.02
massive change longer25 ⟶ massive deletion	0.13	0.33	0.02	-0.10
print diff ⟶ long interval	0.11	0.42	0.02	0.005
print var ⟶ change var name	0.08	0.51	0.02	0.001
print var ⟶ massive change longer25	0.08	0.45	0.02	-0.004

For many constituents, their occurrence distributions were similar across t1, t2, and t3. submission undo mostly occurred in the first three bins, print var reached its peak in the second and third bins, massive deletion and massive change shorter5 occurred more in the last two bins. In terms of distribution difference, t2 and t3 were more similar to each other. print diff occurred more in the last three bins for both t2 and t3, but in t1 it occurred more in the first bin. long interval was mostly in the first bin and decreased in the second and third bin in t1, but it was distributed more equally in the first three bins in t2 and t3. change var name was most frequent in the last bin in t1, but shifted in the third bin in t2, and first bin in t1. massive change longer25 was most frequent in the first bin. repeated changes occurred more in the first bin in t1, but more in the last bin in t2 and t3.

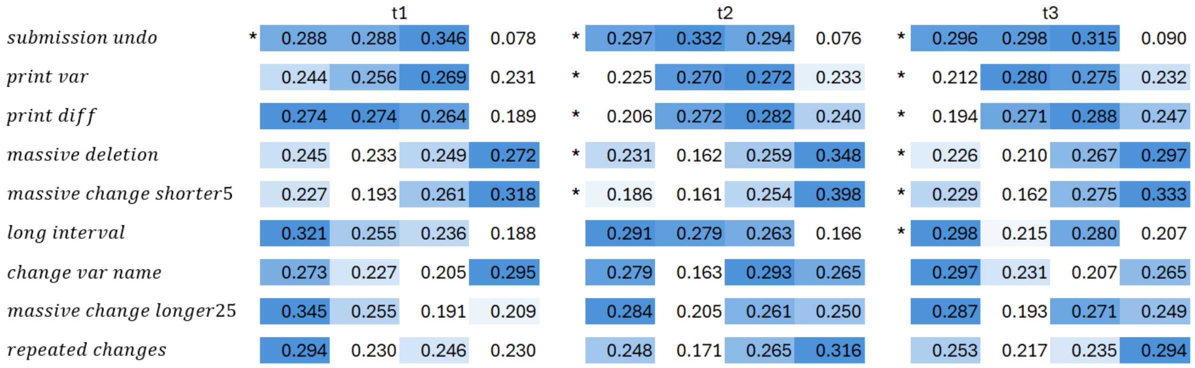


Figure 2: The heatmaps of constituent occurrences.

Table 4
The *L* of constituent patterns for t1, t2, and t3

Patterns	**t1**	**t2**	**t3**	***p***
print var ⟶ print diff	-0.06	0.19	0.20	0.00
submission undo ⟶ massive change longer25	0.00	0.00	0.04	0.01
print diff ⟶ print var	-0.16	0.37	0.29	0.01
submission undo ⟶ print diff	0.00	0.00	-0.07	0.06
massive change longer25 ⟶ change var name	0.00	-0.06	-0.04	0.29
change var name ⟶ print var	0.07	-0.06	-0.06	0.40
print var ⟶ submission undo	0.03	0.00	0.00	0.44
long interval ⟶ print var	0.01	-0.03	-0.04	0.55
print diff ⟶ massive change longer25	-0.01	0.00	0.00	0.69
long interval ⟶ change var name	0.02	0.00	0.00	0.70

Distribution heatmaps showed how single constituent can occur differently across semester. We were also interested in how constituent patterns might change as students gain more programming experience throughout the semester. We chose to compare *L* values between t1, t2, and t3 as *L* considers base rates and is easy to interpret (with positive value indicating that constituents are more dependent than randomness). The average *L* values in three phases and *p* values of ANOVA are shown in Table 4. Three patterns were found to be significantly different across the three phases, print var ⟶ print diff, submission undo ⟶ massive change longer25, and print diff ⟶ print var. As the semester proceeded, they generally occurred more frequently in t2 and t3 than t1.

5. Discussion

The goal of this study was to combine sequence analysis and interpretable debugging features to investigate novice programmers' debugging process by answering the following two research questions.

5.1. RQ1: sequence analysis of debugging

Our first research question was "How can sequence analysis help to understand novice programmers' debugging process?" To answer this question, we applied both SPM and transition metrics to calculate support, confidence, lift, D'Mello's *L*, and *LSA* to rank constituent patterns. We found that constituents did not occur frequently across episodes, with the highest one occurring only in 39% of the episodes. These sparse occurrences lead to the relatively low support of constituent patterns. However, lift and *L* values showed that printing constituents are mostly dependent on each other, even when accounting for their base rate. Since printing for tracing usually can help with understanding the problem and locating the bug, this suggests that novices may require multiple submissions to navigate the early stages of problem-solving before implementation. Such findings echo the previous work where debugging appeared more in an iterative manner for both expert and novice programmers [3, 17].

5.2. RQ2: debugging process across semester phases

Our second research question was "How does novice programmers' debugging process change across different phases of the semester?" We first identified temporal interesting constituents and visualized their occurrences within each episode using heatmaps. We found that similarly for the three phases, submission undo commonly covered the beginning and middle of an episode. Printing reached its peak in the middle of an episode, while the peaks of massive change shorter5 and massive deletion were at the end. This was generally reasonable as it may take a while before students started tracing to identify and locate the real issue in the code. And in the end, they would implement more through faster massive change or massive deletion. What was different is that long interval occurred more at the beginning in t1. It may suggest that students took longer to get used to the auto-grading system when starting the course. Also, questions in t2 and t3 can be more difficult and require more thinking not just at the beginning of debugging. change var name kind of shifted mostly from the end to the middle and then the beginning across semester, indicating students may gain more experience in understanding and representing the code logic throughout the semester. repeated changes shifted more from the beginning of an episode to the end, which aligned with the expectation as it is one of the ways to fix bugs.

We then compared *L* values of constituent patterns between three phases. Results showed t2 and t3 included more patterns that are related to the early stages of problem-solving, such as print var → print diff, and transitioning to long planning like submission undo → massive change longer25. This finding was unexpected as expertise in debugging was found to be able to be improved through practice [31], so we assumed that students should be more efficient in understanding the problem and fixing the bug as the semester proceeded. However, previous studies suggested that experts can spend more time at the initial stages of problem-solving [24] and planning is not determinant to programming success [6]. It is possible that, as the questions become more complex and longer, students needed to engage in more extensive preparation and tended to be more cautious before implementing a strategy. Therefore, intensive understanding of the code or transitions into long planning may not necessarily be a sign of unproductive debugging. Students may strategically adjust their behaviors considering question difficulty, their debugging experience, and many other factors.

6. Conclusion and limitation

In this study, we combined sequence analysis and interpretable debugging constituents to understand novice programmers' debugging process. We identified frequent constituent patterns by conducting SPM analysis and calculating transition metrics. To compare students' debugging process across semester, we applied temporal interestingness techniques and visualized constituent occurrences within episodes. We also compared *L* values between the three phases. We found that it was common for students to transit between different types of printing actions, an early stage in problem-solving to identify and understand the problem. It was also common to undo their submission across semester, as a way to restart from a previous point when debugging. Comparing what was different in the three phases, we found extensive printing and longer planning occurred more as semester proceeded.

A potential avenue for future research is to extend this analysis by comparing various sequential methods on programming submissions. Clustering of sequences could be applied to first filter out episodes with less meaningful debugging behaviors. Other sequential methods, such as process mining, could also be applied to obtain various outputs other than transition ranking, as a way to obtain a more holistic view of the debugging process. Besides, more quantitative features can be added to the current set of constituents to capture additional debugging behaviors. In addition, students' reflections on their debugging process could be collected through think-aloud or interviews, as supplementary evidence to interpret their debugging strategies.

Some limitations of this study need to be considered. First, occurrence of the debugging constituents was low overall, limiting further sequential analysis of constituent patterns. This could be due to the introductory level of the course where we collected the submission log data. Unlike previous studies where programming questions were designed for specific research goals [14], all the homework questions in this study were self-contained problems designed to be successfully finished within one day of their assignment. These questions generally needed shorter code to be solved successfully and less effort to debug. Another reason could be the debugging constituents highly relied on experts' experience and can be somewhat insufficient to describe all the possible debugging behaviors. Second, there were no indications of students' behaviors outside of the homework system. Students might search online for information about their errors, ask for help or use paper to trace the execution of their code to identify the problem, but submission log data cannot capture such behaviors. Third, more sequential analysis methods can be applied in combination with current study to deepen our understanding of debugging. For example, clustering of sequences can be applied to help filter out episodes that contain fewer debugging behaviors, so that constituents would not be too sparse to hinder the following analysis. In addition, SPM and transition metrics can only identify short patterns of debugging activities and do not support identifying a path starting from the beginning of an episode to its end. Methods such as process mining may be complementary to the ones presented in this study.

Acknowledgements

This study is funded by National Science Foundation Award #1942962.

References

[1] Agrawal, R. and Srikant, R. 1995. Mining sequential patterns. *Proceedings of the Eleventh International Conference on Data Engineering* (Taipei, Taiwan, Mar. 1995), 3–14.

[2] Ahmadzadeh, M., Elliman, D. and Higgins, C. 2005. An analysis of patterns of debugging among novice computer science students. *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education* (Caparica Portugal, Jun. 2005), 84–88.

[3] Alaboudi, A. and LaToza, T.D. 2023. What constitutes debugging? An exploratory study of debugging episodes. *Empirical Software Engineering*. 28, 5 (Sep. 2023), 117. DOI:https://doi.org/10.1007/s10664-023-10352-5.

[4] Begum, M., Nørbjerg, J. and Clemmensen, T. 2018. Strategies of Novice Programmers. *Information Systems Research Seminar in Scandinavia (IRIS)* (Odder, Denmark, Aug. 2018).

[5] Berland, M., Martin, T., Benton, T., Smith, C.P. and Davis, D. 2013. Using Learning Analytics to Understand the Learning Pathways of Novice Programmers. *The Journal of the Learning Sciences*. 22, 4 (2013), 564–599.

[6] Blikstein, P., Worsley, M., Piech, C., Sahami, M., Cooper, S. and Koller, D. 2014. Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming. *The Journal of the Learning Sciences*. 23, 4 (2014), 561–599.

[7] Bosch, N. and Paquette, L. 2021. What's Next? Sequence Length and Impossible Loops in State Transition Measurement. *Journal of Educational Data Mining*. 13, 1 (Jun. 2021), 1–23. DOI:https://doi.org/10.5281/zenodo.5048423.

[8] Bruning, R.H., Schraw, G.J. and Norby, M.M. 2011. *Cognitive psychology and instruction.* Allyn & Bacon/Pearson.

[9] Chao, P.-Y. 2016. Exploring students' computational practice, design and performance of problem-solving through a visual programming environment. *Computers & Education.* 95, (Apr. 2016), 202–215. DOI:https://doi.org/10.1016/j.compedu.2016.01.010.

[10] D'Mello, S., Taylor, R. s and Graesser, A. 2007. Monitoring Affective Trajectories during Complex Learning. *Proceedings of the Annual Meeting of the Cognitive Science Society* (2007).

[11] Dong, Y., Marwan, S. and Shabrina, P. 2021. Using Student Trace Logs To Determine Meaningful Progress and Struggle During Programming Problem Solving. *Proceedings of 14th International Conference on Educational Data Mining (EDM21)* (Paris, France, Jul. 2021).

[12] Faraone, S.V. and Dorfman, D.D. Lag Sequential Analysis: Robust Statistical Methods. *Psychological Bulletin.* 101, 2, 312–323. DOI:https://doi.org/10.1037/0033-2909.101.2.312.

[13] Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L. and Zander, C. 2008. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education.* 18, 2 (2008), 93–116. DOI:https://doi.org/10/fsj2gj.

[14] Fitzgerald, S., Simon, B. and Thomas, L. 2005. Strategies that students use to trace code: an analysis based in grounded theory. *Proceedings of the 2005 international workshop on Computing education research(ICER'05)* (Seattle, WA, USA, Oct. 2005), 69–80.

[15] Gao, G., Marwan, S. and Price, T.W. 2021. Early Performance Prediction using Interpretable Patterns in Programming Process Data. *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event USA, Mar. 2021), 342–348.

[16] Hong, J.-C. and Liu, M.-C. 2003. A study on thinking strategy between experts and novices of computer games. *Computers in Human Behavior.* 19, 2 (Mar. 2003), 245–258. DOI:https://doi.org/10.1016/S0747-5632(02)00013-4.

[17] Jayathirtha, G., Fields, D. and Kafai, Y. 2020. Pair Debugging of Electronic Textiles Projects: Analyzing Think-Aloud Protocols for High School Students' Strategies and Practices While Problem Solving. *Procedings of the 14th International Conference of the Learning Sciences (ICLS)* (Nashville, Tennessee, Jun. 2020), 1047–1054.

[18] Jemmali, C., Kleinman, E., Bunian, S., Almeda, M.V., Rowe, E. and Seif El-Nasr, M. 2020. MAADS: Mixed-Methods Approach for the Analysis of Debugging Sequences of Beginner Programmers. *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland OR USA, Feb. 2020), 86–92.

[19] Katz, I.R. and Anderson, J.R. 1987. Debugging: An Analysis of Bug-Location Strategies. *Human–Computer Interaction.* 3, 4 (Dec. 1987), 351–399. DOI:https://doi.org/10.1207/s15327051hci0304_2.

[20] Kessler, C.M. and Anderson, J.R. 1986. A model of novice debugging in LISP. *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers* (USA, Jun. 1986), 198–212.

[21] Kinnebrew, J.S., Mack, D.L.C. and Biswas, G. Mining Temporally-Interesting Learning Behavior Patterns. *Proceedings of the 6th International Conference on Educational Data Mining (EDM 2013)* (Memphis, TN, USA), 252–255.

[22] Liu, C.-C., Cheng, Y.-B. and Huang, C.-W. 2011. The effect of simulation games on the learning of computational problem solving. *Computers & Education.* 57, 3 (Nov. 2011), 1907–1918. DOI:https://doi.org/10.1016/j.compedu.2011.04.002.

[23] Liu, Q. and Paquette, L. 2023. Using submission log data to investigate novice programmers' employment of debugging strategies. *Proceedings of the 13th International Learning Analytics and Knowledge Conference* (Arlington TX USA, Mar. 2023), 637–643.

[24] Liu, Z., Zhi, R., Hicks, A. and Barnes, T. 2017. Understanding problem solving behavior of 6–8 graders in a debugging game. *Computer Science Education.* 27, 1 (Jan. 2017), 1–29. DOI:https://doi.org/10.1080/08993408.2017.1308651.

[25] Lye, S.Y. and Koh, J.H.L. 2014. Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior.* 41, (Dec. 2014), 51–61. DOI:https://doi.org/10.1016/j.chb.2014.09.012.

[26] McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L. and Zander, C. 2008. Debugging: a review of the literature from an educational perspective. *Computer Science Education.* 18, 2 (Jun. 2008), 67–92. DOI:https://doi.org/10.1080/08993400802114581.

[27] Merceron, A. and Yacef, K. 2008. Interestingness Measures for Association Rules in Educational Data. *Proceedings of 1st International Conference on Educational Data Mining* (Montréal, Québec, Canada, Jun. 2008), 57–66.

[28] Murphy, L., Lewandowski, G., McCauley, R., Simon, B., Thomas, L. and Zander, C. 2008. Debugging: the good, the bad, and the quirky -- a qualitative analysis of novices' strategies. *ACM SIGCSE Bulletin.* 40, 1 (Mar. 2008), 163–167. DOI:https://doi.org/10.1145/1352322.1352191.

[29] Pellas, N. and Vosinakis, S. 2018. The effect of simulation games on learning computer programming: A comparative study on high school students' learning performance by assessing computational problem-solving strategies. *Education and Information Technologies.* 23, 6 (Nov. 2018), 2423–2452. DOI:https://doi.org/10.1007/s10639-018-9724-4.

[30] Perkins, D.N. and Martin, F. 1986. Fragile knowledge and neglected strategies in novice programmers. *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers* (USA, Jun. 1986), 213–229.

[31] Pinto, J.D., Liu, Q., Paquette, L., Zhang, Y. and Fan, A.X. 2023. Investigating the Relationship Between Programming Experience and Debugging Behaviors in an Introductory Computer Science Course. *Advances in Quantitative Ethnography* (Cham, 2023), 125–139.

[32] Polya, G. 1957. *How to Solve It: A New Aspect of Mathematical Method.* Princeton University Press.

[33] Quinlan, J.R. 1986. Induction of decision trees. *Machine Learning.* 1, 1 (Mar. 1986), 81–106. DOI:https://doi.org/10.1007/BF00116251.

[34] Zaki, M.J. 2000. Sequence mining in categorical domains: incorporating constraints. *Proceedings of the 9th international conference on Information and knowledge management* (McLean Virginia USA, Nov. 2000), 422–429.

[35] Zhang, Y. and Paquette, L. 2020. An effect-size-based temporal interestingness metric for sequential pattern mining. *Proceedings of The 13th International Conference on Educational Data Mining* (2020), 720–724.

[36] Zhou, Y., Liu, Q., Yang, S. and Alawini, A. 2023. Identifying Collaborative Problem-Solving Behaviors Using Sequential Pattern Mining. *Proceedings of 2023 ASEE Annual Conference & Exposition* (Jun. 2023).