MAADS: Mixed-Methods Approach for the Analysis of Debugging Sequences of Beginner Programmers

Chaima Jemmali*, Erica Kleinman*, Sara Bunian*,
Mia Victoria Almeda†, Elizabeth Rowe†, Magy Seif El-Nasr*
* Northeastern University, † EdGE at TERC
{jemmali.c,kleinman.e,banian.s}@husky.neu.edu
{mia_almeda,elizabeth_rowe}@terc.edu
magy@northeastern.edu

ABSTRACT

Debugging is a cornerstone of programming and has been shown to be especially problematic for beginners. While there has been some work trying to understand the difficulties that beginners face with debugging, investigating common mistakes or specific error types they struggle with, there is little work that focuses on in-depth analysis of how novice programmers approach debugging, and how it changes over time. In this paper, we present MAADS (Mixed-Methods Approach for the Analysis of Debugging Sequences), a scalable and generalizable approach that combines quantitative and qualitative methods by using a state/action representation and visualization to gain knowledge about the debugging process through a step by step analysis. To demonstrate the utility of MAADS, we analyzed the debugging processes of middle school students who developed code within *May's Journey*, a game designed to teach basic programming principles. The approach showed great utility in identifying differences in students' debugging techniques and learning paths.

CCS CONCEPTS

• **Human-centered computing** → *Empirical studies in HCI*; *Visualization design and evaluation methods*; • **Applied computing** → *Interactive learning environments*;

KEYWORDS

Debugging, Programming, Process, Beginners, Methodology

ACM Reference Format:
Chaima Jemmali, Erica Kleinman, Sara Bunian, Mia Victoria Almeda, Elizabeth Rowe, Magy Seif El-Nasr. 2020. MAADS: Mixed-Methods Approach for the Analysis of Debugging Sequences of Beginner Programmers. In *The 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20), March 11-14, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3328778.3366824

SIGCSE '20, March 11–14, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6793-6/20/03…$15.00
https://doi.org/10.1145/3328778.3366824

1 INTRODUCTION

Teaching programming is a difficult task that has received increased attention from researchers over the past few decades [10, 25, 30]. An important aspect of learning how to program is debugging, the process of finding and resolving errors within the code. Debugging is difficult, and has been identified as one of the biggest challenges that beginner programmers face [17, 18, 24]. Begum et al. [4] enumerated knowledge requirements for successful debugging as: knowledge of the intended and actual program, understanding of the language, general programming expertise, and knowledge of bugs, debugging methods, and the application domain.

Existing literature [1, 2, 5, 14, 16, 20, 29] has identified patterns and developed high level frameworks that explain general strategic elements of debugging. While details vary, these frameworks follow a similar structure of steps where a programmer would first understand the system enough to recognize that something is wrong [5, 14] and/or develop an initial hypothesis regarding an error [2, 5, 14]. They would then follow various steps of testing the hypothesis until the error is resolved, or looping back to previous steps if the error is not resolved [2, 14].

Learning how to debug is important as it's an effective way to improve general problem-solving skills [19] and can better assist in learning [5]. Therefore, investigating the debugging process in detail is important since it *not only* sheds light on what makes programming difficult for introductory students, but also assists in the development of systems for improving problem-solving techniques and lowering steep learning curves [11]. However, such in-depth investigation into the debugging process can be challenging and studies that tackled this problem are either (a) qualitative [8, 9, 33], which can be critiqued as being time consuming and lacking scalability or generalizability, (b) language construct specific [15, 31, 32] and hence constrained and difficult to generalize, or (c) purely quantitative and predictive rather than exploratory, often focusing on prediction of student success/failure, and not on providing interpretable models of the process itself [27, 28].

In this paper, we propose MAADS, a methodology that allows us to investigate debugging as a process and produce interpretable models using data-driven techniques and visualizations. MAADS enables us to examine students' progress within each problem, clearly showing how they tackle debugging and how their errors evolve over time. To visualize students' paths, we used a state/action representation where a state is the error types present in a submission and the action is the change made to the code between submissions. This approach provides a deeper, more contextually

informed analysis of learning trajectories, similar to what a qualitative study would do, but at a scale that can allow inferences on debugging processes of groups of students' behaviors and not just one student at a time.

2 RELATED WORK

While research on programming education is abundant, there is little work looking at the debugging process through investigation of student trajectories or strategies, which is our focus. Spohrer and Soloway [33] analyzed students' bugs when they compiled their programs (i.e., after they resolved all syntax errors). However, they only looked at final submissions and did not observe the progression or analyze intermediate code. Webb [35] investigated debugging strategies with students working in groups; they recorded student interactions with one another and all interactions within the computer. They then categorized debugging strategies based on: level of abstraction, size of the program unit, and whether the strategy was in response to a suggestion by an instructor. They found that students showed very little planning, displaying an opportunistic style of debugging. While these approaches are informative, they are time consuming and difficult to recreate at a large scale.

In addition to manual inspection of strategies, several researchers used machine learning techniques to shed light on the debugging process. For example, Piech et al. [28] modeled the paths that students take in debugging using Hidden Markov Models and then clustered them to understand how students varied in the development of pathways for programming assignments, including the type of states students visited and transitions between them. Similar to our approach, they were able to look at the clusters in terms of states and actions taken by the students. While very promising, their approach, being unsupervised, is difficult to interpret. It is, for example, difficult to make sense of the underlying misconceptions behind the different states exhibited in the HMM model. For instance, they defined a "sink" state where students tend to get stuck, but it is not clear why students got stuck in terms of the types of error exhibited, or how the student was approaching debugging. The group also used Recurrent Neural Networks (RNN) to model students' learning [27], in order to predict students' performance in future problems. However, they were primarily focused on clustering of latent topics, using datasets from the ASSISTments tutoring system and Khan Academy, and did not provide an in-depth analysis of students' learning processes over time. Blikstein [6] analyzed code snapshots from several students and found various strategies, highlighting ones that differentiate between novice and expert programmers. To automate the analyses, he focused on the number of characters and the actions of adding, moving and modifying lines of code as well as the differences in size and frequency between the code snapshots. While there was no significant correlation between the size of code updates (i.e., writing larger or smaller codes at a time) and grades, the amount of change in student's programming patterns over time was significantly correlated with course performance [7]. In contrast, our current study evaluates each code submission according to the different types of errors present and the amount of code that was changed each time. Our methodology requires human intervention to identify error types, while the rest of the process is fully automated for fast and scalable analysis.

3 PROGRAMMING ENVIRONMENT

In this paper, we use *May's Journey*, a 3D puzzle game that teaches the basics of programming by having learners type simple instructions in the game's custom programming language to interact with objects, solve puzzles, and navigate an environmental maze [12, 13]. This game was chosen because (1) it introduces programming in a very simple and gradual way allowing us to investigate beginners' learning processes, and (2) the game can log all interaction data, giving us a clear record of what each student did at each time step. However, our methodology can be applied to any programming language or environment that provides appropriate logging.

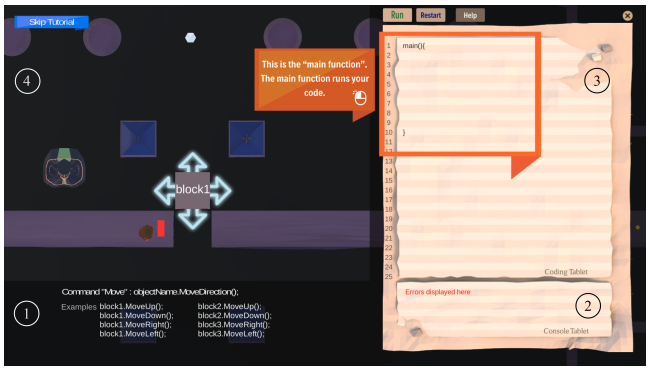


Figure 1: Coding interface: (1) Structure of the command with examples, (2) Error messages, (3) Text area and buttons to run/restart/help, (4) Programmable objects and execution.

3.1 Learning goals

The game introduces Object Oriented Programming (OOP) using a custom programming language. Through observation and exploration, the player is introduced to objects that can be manipulated within the environment. The game language conforms with OOP conventions, while reducing the verbosity to lower the barrier of syntax.

Players solve puzzles and progress to subsequent levels through coding. While exploring the game world, they will find that their paths are blocked, and that they can only clear the obstacles by pulling up a coding interface, as shown in figure 1, to type in their own code. The interface includes further relevant information, such as commands available in that level and how to use them. Objects that can be manipulated by the code have their names displayed along with indications of how they can be interacted with (moving or rotating). For example, in the first level, players have to move the `block1` down by writing `block1.MoveDown();` inside the main function. The main function is provided at the beginning of each level and is reset (with its contents cleared) if the player restarts.

3.2 Debugging

The game features an error handling mechanism that provides comprehensive feedback, introducing learners to the debugging process. Error messages are designed to be clear, concise, and helpful without giving away solutions. A player can make two types of errors: code-related and puzzle-related. code-related errors, such

as “Missing opening curly bracket { for the for loop in line 5”, are displayed in the console. We further differentiate between these errors in later sections. puzzle-related errors are presented in the game in the form of a sound cue and a reset of the puzzle to its initial layout.

In order to progress through the game, players have to identify errors, make the appropriate code changes to correct them, and re-run the code until they solve the puzzle. These steps align with what they would have to do in other programming environments.

3.3 Topics & Instruction

The work discussed here uses the first 10 levels of the game, covering basic instructions, sequence logic, and loops. Basic instructions are introduced using two methods: *object.MoveDirection();* where Direction can be Left, Right, Up, and Down, and *object. Rotate("direction");* where, in this case, the direction is a *String* argument, introducing functions with arguments as well as a new type. Sequence logic is introduced through puzzles where the player must input commands in the correct order to progress and win the level. Finally, a simplified *for loop* is presented. The structure takes two Integers as the beginning and end and then repeats the commands inside the loop the designated number of times. An example code in this custom programming language is as follows:

```
main() {
    for (0 to 5) {
        block1.MoveLeft();
        block2.Rotate("right");
    }
}
```

4 PARTICIPANTS & DATA

In order to demonstrate the utility of this methodology, and investigate the debugging processes of novice programmers, we conducted a study with participants in 6th, 7th and 8th grade from 2 different public schools. They were presented with the game during normal class time from their instructor alone or accompanied by one or two researchers. When present, researchers observed students as they played and took notes of struggle points. Further, researchers noted levels in which players spent a lot of time, the most common errors they faced, and how they overcame each error and who helped them. Students played during class time and were able to play more during their free time. We collected 9452 code submissions related to 10 levels and generated by 185 different profiles. The actual number of participants is not known, since a player can create multiple profiles with different user-names.

5 METHODOLOGY

The rationale behind MAADS is to leverage qualitative analytical power that allows us to observe debugging sequences and capture the patterns and individual differences in a scalable and generalizable way. The approach requires a 4-step process, where in Step 1: the error types present in the data set are identified, in step 2: All errors in each code submission are automatically extracted and tagged with the appropriate type from step 1, in Step 3: the submissions are converted into a state-action transition, where a state is the labeled errors from Step 2 and actions are related to what type of change had been made to the code to reach that state, and lastly in step 4: the state/action transition is visualized as a sequence. Using the visualized sequences, a researcher can then inspect students’ debugging approaches and progression to identify patterns and build models of the debugging process.

5.1 Error Types

To classify the errors, we needed to first identify them. We collected all error messages possible in the levels studied and noticed that while some of the in-game error messages are very precise, to help the player debug, they can be too specific for a classifier or a clustering algorithm (e.g., “The begin and end arguments of the for loop should be integers. an integer is a whole number. Examples: -4, 0, 2, 10... ”). At the same time, some error messages are too broad (e.g. “X doesn’t match the commands and objects available”), which makes it difficult to understand if it’s a purely syntax error or another type of misunderstanding.

To enable the right abstraction level for analysis, we identified 4 major error types that the players make: Structure, Syntax, Reasoning, and Puzzle. These categories also align with the work of Katz and Anderson [14], where they identified five error types that programmers make: Goal, Misrepresentation, Intrusion, Misconceptions, and Syntactic. Our error types are defined as follows:

Puzzle Errors: Players’ code runs (no compiler errors) but does not solve the puzzle. It is somehow similar to Goal errors which are defined as missing code, but in this case, it can also be extra code or code that does something different from what is intended.

Structure errors: These refer to problems with the structure of the code. For example, when players put code outside of the main function, or don’t have a main function at all. Another example is calling a function without an object. We differentiate these errors from syntax errors such as forgetting a curly bracket or a semicolon. This is also similar to Katz and Anderson’s Misconception errors.

Reasoning errors: These include both Misrepresentation and Intrusion errors. For example, players use a valid function but apply it to the wrong object, such as rotating an object that can only be moved. Another example is trying to use code that worked in a previous level in the current level without making the necessary changes. For example, moving block2 which existed in a previous level but does not exist in the current one.

Syntax errors: These are purely syntax errors such as upper-lower case, missing semicolon, etc.

Undefined errors: These are errors that could not be categorized by the parser.

5.2 Error Types Extraction

Table 1 shows how submissions have been automatically labeled by our parser. Some errors can have one of two types depending on the specific case. For example, *object does not exist* and *command does not exist* could be due to a simple syntax error, like a typo or it could be due to a reasoning error, such as using objects from previous levels or from examples. In this case, we check the name they typed against previously used object names to differentiate the two cases. For example, the error “block2 doesn’t match the commands and objects available” and block2 is in the examples given to the player

or used in a previous level, is different from the error “blocck2 doesn’t match the commands and objects available” where it is a typo. The same process is applied to the wrong argument by checking if they made a syntax error “block1.Rotate("leftt");)” or a reasoning error by putting the wrong type such as an int instead of a string “block1.Rotate(2);)”.

5.3 State/Action Representation

Looking more closely at the code submissions, it becomes clear that the number of errors in a single submission varies a lot. The game parser will read the code submitted and throws the first error it encounters without considering the rest of the code. One code submission could have a wrong object, a missing semi-colon, and a missing curly bracket and the parser would only catch one of those. Therefore, we post-processed each code submission, with an augmented parser that identifies all errors within each submission as seen in table 1. Then, each submission is represented as a state $S = (x, y, z, w)$ where x, y, z, w symbolize the number of errors for, respectively: structure, reasoning, puzzle, and syntax. For instance, $S_i = (0, 2, 0, 1)$ means the submission i has two reasoning errors and one syntax error. Note that we have excluded undefined errors since they don’t bring any insights into the issue encountered.

While the augmented parser works in most cases, there are still certain instances (7.9% of submissions) in which it mislabels data. These generally lead to a wrong error message followed by unidentified errors. These are common compiler misinterpretations that exist in various other programming languages [34]. Making compiler error messages more comprehensible and less misleading is still an ongoing research problem [3, 26].

While the state represents the errors in a code submission, the action shows how the player transitioned between those states. Table 2 describes the different actions that make up each player’s action sequence, and how they are derived from the change made between code submissions. Similar to the work of Blikstein [7], players can either add, remove, or change previous code. However, in our case, we do not look at averages or absolute values, but instead abstract the amount of change to be relative to the previous code. In fact, in the different puzzles studied, the length of the solution is usually small. it also varies between levels which makes absolute values of length not accurate in this case. Therefore, the difference between small, medium, or large changes is relative to the previous submission’s code length as explained in Table 2. Additionally, for add and change, we check if the added code exists in the previous submission and if so, the action is extended with "added existing code". For example, "added medium chunk of code/added existing code" is a possible action.

5.4 Sequences & Visualization

With these states and actions, for each level and each player, a sequence is made from the start of the level to the correct answer or until the player quit. To visualize the process, we adopt the work on visual representation introduced in [21]. This visualization allows an easy understanding and investigation of player behavior [22, 23]. *Glyph*, shown in Figure 2, is composed of two visual representations: a state graph (right) and a sequence graph (left). The State graph is the node-link diagram of the game states and actions of the students.

Code Submission	Labeled Errors	Type
main(){ block1.Rotate(left);}	wrong argument object does not exist	syntax reasoning
for(0 to 4) {stone1.MoveRight(); Stone1.MoveRight();}	case error missing main	syntax structure

Table 1: Code submissions labeled and classified by the augmented parser that recognizes multiple errors per submission.

Action	Description
no changes	did not make any code change between the last two states
(added/removed/changed) **small** chunk of code	the length of the code (added/removed/changed) represents **less than 15%** of the previous code’s length
(added/removed/changed) **medium** chunk of code	the length of the code (added/removed/changed) is between **15% and 50%** of the previous code’s length
(added/removed/changed) **large** chunk of code	the length of the code (added/removed/changed) represents **more than 50%** of the previous code’s length

Table 2: Actions that represent the modification to the code made by a player between two states.

The size of the states and the thickness of the links vary with the number of players visiting the states and the links, respectively. The Sequence graph displays nodes that represent the sequence patterns exhibited by users. Each node represents a full sequence and the distance between the nodes represents how similar the sequence patterns are. Closer nodes are more similar.

6 RESULTS

In this section, we present the patterns we identified through the use of the visualized sequences. We first discuss common patterns in sequences, showing the utility of the approach in identifying general trends in a large dataset. We then examine individual differences in the paths of players with multiple code submissions. With aggregate data alone, these players would be considered the same. However, from their progression, we can identify the differences between them, demonstrating that the number of errors alone does not always inform about a player’s performance.

6.1 Patterns & Common Sequences

In Figure 2, we can observe the various sequences in level_1_3, with the three most frequent highlighted. This level has 134 players, 21 state nodes and 85 different paths. For each path, we can observe the list of states and actions leading to them. *Glyph* allows us to quickly identify and differentiate between players who made no mistakes, those who made a few mistakes, but were efficient at solving them, and those who struggled to debug their code. Most players (15.3%) encountered no errors in this level, followed by (10.6%) who encountered only a single puzzle error and were able to quickly debug the mistake in order to reach the level solution. This can be observed in Figure 2. An examination of the action sequence that corresponds with the sequence containing the error reveals

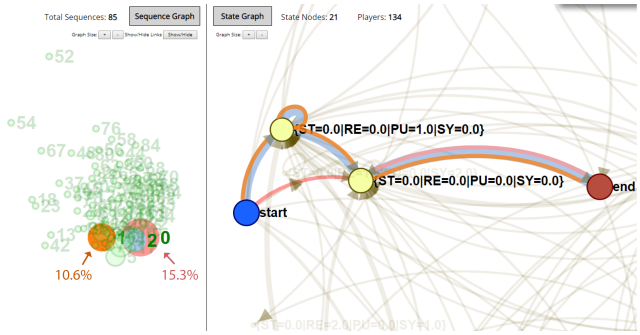


Figure 2: Highlighted are the 3 most frequent sequences. In pink, the most frequent sequence with no errors. In blue, the second and third most frequent sequences with 1 or more visits to a state with a puzzle error.

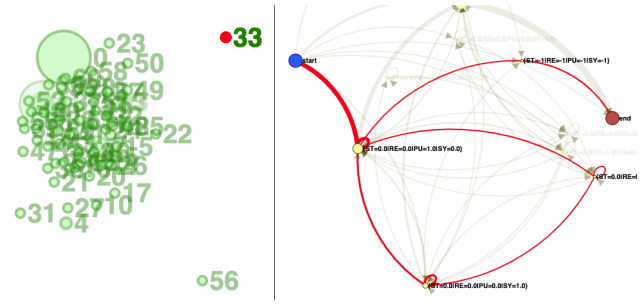


Figure 3: An example of a player who struggled to debug their errors and ultimately quit the game. Their sequence is characterized by numerous loops between the same state as well as looping between various error states.

that these"higher performin" players fixed this error by adding a medium chunk that was syntactically similar to the code that already existed, a systematic way of debugging based on smaller, targeted changes and knowledge of code that is known to work.

The players who struggled to debug their code efficiently were identified by their sequence node being farther from the most common patterns. Their sequences were characterized by loops, both between different error states and a single state. These indicate subsequent visits to the same error state as well as a tendency to reintroduce old errors while trying to fix new ones. An example of this can be seen in Figure 3. Note that while two states that are similar in our representation may not be the exact same code, we argue that they still carry the same information as to what type of error the player is struggling with. An examination of the sequences of players such as the one seen in Figure 3 reveals that many players who displayed similar state visualizations tended to perform large code changes between submissions, rather than medium or small ones. Similarly, such players also resubmitted incorrect code with no changes at much higher frequencies than their "high performing" counterparts.

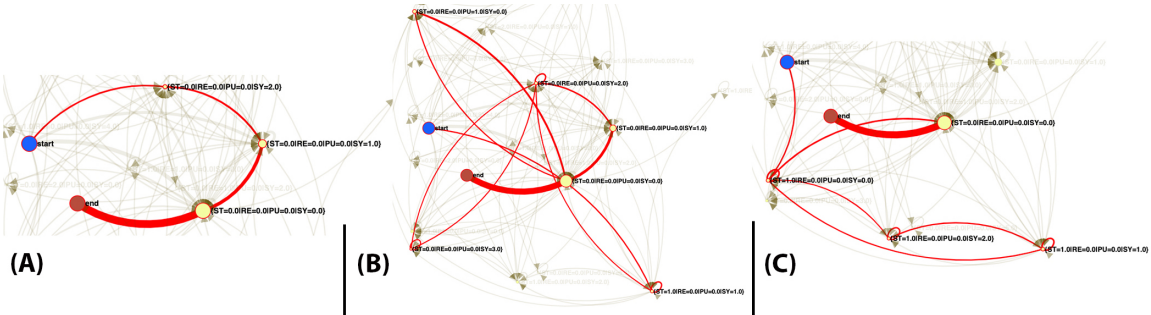
6.2 Individual Differences in Player Sequences

Aside from looking at the sequences as a whole, examining the data in this sequential manner allows us to make observations about how the students' debugging strategies changed as they worked on a given puzzle. We look at the state graphs and sequences of three players (seen in Figure 4) who completed the first level, in order to compare their processes.

In the *Glyph* visualization, we can easily see that player (A) had a clear linear progression from start to finish. By taking a closer look at their sequences, we can see that, though they started with two syntax errors, they only ever made small changes to their code and that they progressed from two errors, to one error, to error-free code. We can hypothesize that this player had a highly efficient debugging strategy that focused on small, localized changes targeting a single error at a time. As discussed previously, such a strategy was common among higher performing debuggers. Another observation about player (A), is that they never submitted code with no changes, a behavior that was performed by most players in the data-set. This may be the result of the player being at a high level of understanding in terms of how their code affected the environment/interacted with the puzzle, the simplicity of the puzzle in the given level, or a combination of the two.

A very different story is told by the graph of player (B). Unlike player (A), player (B) visits a larger number of error states. Their graph contains frequent looping between and within error states, and there is less of a sequential, linear path from the beginning to the solution. This indicates that player (B) struggled to debug their code, frequently resubmitting code with the same error, or introducing new errors while trying to correct older ones. Looking at player (B)'s sequence, we can see that they frequently made medium sized code changes, and had very few small changes, indicating that their code edits were less targeted, perhaps because they were less certain of where the error was than player (A). However, they never make large code changes, and do progress into a brief series of smaller code changes towards the end of their sequence. Additionally, while they do resubmit code without changes, they do not do so an exhaustive number of times. These resubmits likely represent a strategy of observation in which the code is run multiple times to confirm the output is consistent. The fact that these appear between almost every other state in the sequence indicate a possible strategy of trial and error, and it is noteworthy that they become less frequent towards the end of the sequence, implying that the need for such confirmatory tests may have diminished. This, combined with the smaller code changes towards the end of the sequence, indicate that the player's trial and error approach did eventually allow them to identify their error, debug their code, and solve the puzzle.

Finally, the graph of player (C) is an interesting one that sits somewhere between the previous two in terms of general shape and complexity. We can see a sequence that hits an error state, goes on to hit two others, then loops back to that initial error state before reaching the solution, indicating that they may have made code edits that did not move them closer to any answer. In addition, unlike the previous two examples, there are loops within each error state, indicating that the player made numerous subsequent code submissions with the same error. This graph implies that the player struggled to a great degree to debug their code. An examination of their sequence supports this theory, as we see a sequence with a large number of subsequent no-change submissions (between 9 and 20) between predominantly large and medium sized code changes.



(A)

changed small chunk of code(1), added small chunk of code(1), changed small chunk of code(1)
{ST=0|RE=0|PU=0|SY=2}(1),
{ST=0|RE=0|PU=0|SY=1}(1),
{ST=0|RE=0|PU=0|SY=0}(1),

(B)

changed medium chunk of code(3), no changes(3), changed medium chunk of code(1), no changes(6), changed medium chunk of code(1), no changes(2), changed small chunk of code(1), added small chnuk of code(2), changed medium chunk of code(2)
{ST=1|RE=0|PU=0|SY=1}(6),
{ST=0|RE=0|PU=0|SY=2}(7),
{ST=1|RE=0|PU=0|SY=3}(3),
{ST=0|RE=0|PU=0|SY=2}(1),
{ST=0|RE=0|PU=0|SY=1}(1),
{ST=0|RE=0|PU=0|SY=0}(1),
{ST=0|RE=0|PU=1|SY=0}(1),

(C)

changed large chunk of code(1), no changes(20), changed medium chunk of code(1), no changes(13), changed large chunk of code(1), no changes(7), added medium chunk of code(1), no changes(9), changed small chunk of code(1), no changes(16), changed medium chunk of code(1), no changes(1), added medium chunk of code(1)
{ST=1|RE=0|PU=0|SY=0}(35),
{ST=1|RE=0|PU=0|SY=2}(8),
{ST=1|RE=0|PU=0|SY=1}(10),
{ST=1|RE=0|PU=0|SY=0}(19),
{ST=0|RE=0|PU=0|SY=0}(1),

Figure 4: Graphs and Sequences from the first level of players who (A) were a quick and efficient debugger, (B) struggled but found their way through continuous adjustment, and (C) struggled to progress in a sequential manner towards a correct solution.

Additionally, the frequency of no-change submissions does not prominently diminish towards the end of the sequence as it does with player (B). These imply that player (C) was not employing an exploratory, trial and error strategy, but instead may have been resubmitting code many times out of frustration.

7 DISCUSSION

While the observations discussed in Results could be observed from the code submissions of the students, observation, or similar qualitative approaches, such methods would prove time consuming and would not scale efficiently with the number of students and lines of code. MAADS methodology and the *Glyph* visualization allow for quick identification of common and uncommon trajectories, and grant the ability to pinpoint specific trajectories for a deeper examination. Further, they facilitate the ability to do this in an efficient and scalable manner. Being able to observe the trajectories of student debugging in this way provides us with useful insight into the patterns and strategies of debugging employed by novice programmers. Additionally, it can also be used to determine ways in which *May's Journey*, and similar educational environments, can be improved to facilitate learning. For example, the visualization allows us to observe the quit state for each level, and from there we can identify the different paths leading to it. This allows us to trace the trajectories of students who quit, in order to understand the difficulties they faced that led them to do so. Such information can be used to better design instruction and feedback, or even enable an intelligent system to assist players on this path without reducing the challenge for other players. In fact, the need for an adaptive help system is reinforced by looking at the diverse paths players went through. The only clustered sequences were for "high performing" students, however for students who struggled, they went through different states. This may be because most players had never programmed before and had various issues and ways to tackle the problems. We note that in our representation, we did not take the time spent into account. This is because we observed that players would often pause progression to seek help, provide help, or because of unrelated distractions. As a result, time spent does not act as an accurate indicator of performance in this case.

8 CONCLUSION AND FUTURE WORK

We presented MAADS, a novel approach to analyze debugging paths for beginners that is systematic and automated and thus providing a scalable method to investigate learning processes in a deeper way. The methodology allowed us to explore the debugging processes and identify patterns for groups of students, as well as investigate paths taken by single students. Future work should include a larger data set with a bigger range of possible errors. In fact, it will be interesting to observe how patterns change when including errors related to variables, Boolean logic, conditionals, etc. Furthermore, there can be many ways to enrich this methodology by including time spent debugging into the representation or more information about how the code was written with a more granular look between submissions. However, adding too much information could make it harder to see patterns since the sequences would be more specific. It is necessary to strike the right balance during data abstraction. Finally, future work should compare the outcomes of this methodology to qualitative analysis or machine learning models to fully grasp advantages and disadvantages of each.

9 ACKNOWLEDGEMENTS

This research is supported by NSF AISL (Advancing Informal STEM Learning) Award Id: 1810972.

REFERENCES

[1] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An analysis of patterns of debugging among novice computer science students. In *Acm sigcse bulletin*, Vol. 37. ACM, 84–88.

[2] Keijiro Araki, Zengo Furukawa, and Jingde Cheng. 1991. A general framework for debugging. *IEEE software* 8, 3 (1991), 14–20.

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

[4] Marjahan Begum, Jacob Nørbjerg, and Torkil Clemmensen. 2018. Strategies of Novice Programmers. In *The 41st Information Systems Research Seminar in Scandinavia*.

[5] Marina Umaschi Bers, Louise Flannery, Elizabeth R Kazakoff, and Amanda Sullivan. 2014. Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. *Computers & Education* 72 (2014), 145–157.

[6] Paulo Blikstein. 2011. Using learning analytics to assess students' behavior in open-ended programming tasks. In *Proceedings of the 1st international conference on learning analytics and knowledge*. ACM, 110–116.

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

[8] Shirley A Booth. 1998. Learning to program: A phenomenographic perspective. (1998).

[9] Christine Bruce, Lawrence Buckingham, John Hynd, Camille McMahon, Mike Roggenkamp, and Ian Stoodley. 2004. Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university. *Journal of Information Technology Education: Research* 3 (2004), 145–160.

[10] Mark Guzdial. 2004. Programming environments for novices. *Computer science education research* 2004 (2004), 127–154.

[11] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. In *ACM SIGCSE Bulletin*, Vol. 35. ACM, 153–156.

[12] Chaima Jemmali, Sara Bunian, Andrea Mambretti, and Magy Seif El-Nasr. 2018. Educational game design: an empirical study of the effects of narrative. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*. ACM, 34.

[13] Chaima Jemmali and Zijian Yang. 2016. *May's Journey: A serious game to teach middle and high school girls programming*. Master's thesis. Worcester Polytechnic Institute.

[14] Irvin R Katz and John R Anderson. 1987. Debugging: An analysis of bug-location strategies. *Human-Computer Interaction* 3, 4 (1987), 351–399.

[15] Claudius M Kessler and John R Anderson. 1986. Learning flow of control: Recursive and iterative procedures. *Human-Computer Interaction* 2, 2 (1986), 135–166.

[16] Andrew J Ko, Thomas D LaToza, Stephen Hull, Ellen A Ko, William Kwok, Jane Quichocho, Harshitha Akkaraju, and Rishin Pandit. 2019. Teaching Explicit Programming Strategies to Adolescents. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, 469–475.

[17] Andrew J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. IEEE, 199–206.

[18] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A study of the difficulties of novice programmers. In *Acm Sigcse Bulletin*, Vol. 37. ACM, 14–18.

[19] Tami Lapidot and Orit Hazzan. 2005. Song debugging: merging content and pedagogy in computer science education. *ACM SIGCSE Bulletin* 37, 4 (2005), 79–83.

[20] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies. In *ACM SIGCSE Bulletin*, Vol. 40. ACM, 163–167.

[21] Truong-Huy D Nguyen, Magy Seif El-Nasr, and Alessandro Canossa. 2015. Glyph: Visualization Tool for Understanding Problem Solving Strategies in Puzzle Games.. In *FDG*.

[22] Truong-Huy D Nguyen, Magy Seif El-Nasr, and Derek M Isaacowitz. 2015. Interactive visualization for understanding of attention patterns. In *Workshop on Eye Tracking and Visualization*. Springer, 23–39.

[23] Truong-Huy D Nguyen, Michael Richards, Magy Seif El-Nasr, and Derek M Isaacowitz. 2015. A Visual Analytic System for Comparing Attention Patterns in Eye-Tracking Data. *Eye Tracking and Visualization (Proceedings of ETVIS 2015)* (2015).

[24] Roy D Pea and D Midian Kurland. 1984. On the cognitive effects of learning computer programming. *New ideas in psychology* 2, 2 (1984), 137–168.

[25] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. 2007. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin* 39, 4 (2007), 204–223.

[26] Raymond S Pettit, John Homer, and Roger Gee. 2017. Do Enhanced Compiler Error Messages Help Students?: Results Inconclusive.. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 465–470.

[27] Chris Piech, Jonathan Bassen, Jonathan Huang, Surya Ganguli, Mehran Sahami, Leonidas J Guibas, and Jascha Sohl-Dickstein. 2015. Deep knowledge tracing. In *Advances in Neural Information Processing Systems*. 505–513.

[28] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. 2012. Modeling how students learn to program. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. ACM, 153–160.

[29] Kathryn M Rich, Carla Strickland, T Andrew Binkowski, and Diana Franklin. 2019. A K-8 Debugging Learning Trajectory Derived from Research Literature. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, 745–751.

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

[31] RENAN Samurçay. 1989. The concept of variable in programming: Its meaning and use in problem-solving by novice programmers. *Studying the novice programmer* 9 (1989), 161–178.

[32] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. 1983. Cognitive strategies and looping constructs: An empirical study. *Commun. ACM* 26, 11 (1983), 853–860.

[33] James G Spohrer and Elliot Soloway. 1986. Analyzing the high frequency bugs in novice programs. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*. Ablex Publishing Corp., 230–251.

[34] V Javier Traver. 2010. On compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction* 2010 (2010).

[35] Noreen M Webb, Philip Ender, and Scott Lewis. 1986. Problem-solving strategies and group processes in small groups learning computer programming. *American Educational Research Journal* 23, 2 (1986), 243–261.