To appear in *Computer Science Education*
Vol. 00, No. 00, Month 20XX, 1–25

Research Article

Understanding Problem Solving Behavior of 6-8 Graders in a Debugging Game

(*Received 00 Month 20XX; final version received 00 Month 20XX*)

Debugging is an over-looked component in K-12 computational thinking education. Few K-12 programming environments are designed to teach debugging, and most debugging research were conducted on college-aged students. In this paper, we presented debugging exercises to 6th-8th grade students and analyzed their problem solving behaviors in a programming game - BOTS. Apart from the perspective of prior literature, we identified student behaviors in relation to problem solving stages, and correlated these behaviors with student prior programming experience and performance. We found that in our programming game, debugging required deeper understanding than writing new codes. We also found that problem solving behaviors were significantly correlated with students' self-explanation quality, number of code edits, and prior programming experience. This study increased our understanding of younger students' problem solving behavior, and provided actionable suggestions to the future design of debugging exercises in BOTS and similar environments.

Keywords: debugging, K-12, educational games, computational thinking

1. Introduction

Due to advances in educational technologies, increased enthusiasm in computer science outreach programs, and the growing number of international primary-age computing curriculums, more students have started learning to program at a young age. The notion of computational thinking is first envisioned by Papert (1980), as derived from the Constructivism Learning Theory. As Papert (1980) wrote, "computers might enhance thinking and change patterns of access to knowledge" (p. 3). In 2006, Wing broadened the concept of computational thinking as "a fundamental skill for everyone, not just for computer scientists" (p. 33). In Wing's (2006) viewpoint, debugging is a key-part of computational thinking, as "computational thinking is thinking in terms of prevention, protection, and recovery from worst-case scenarios through redundancy, damage containment, and error correction" (p. 34). In 2010, Hemmendinger pointed out that finding and correcting errors, creating representations, and analyzing are among key elements of computational thinking that are shared across many disciplines beyond computer science. Similarly, Barr and Stephenson (2011) as well as Grover and Pea (2013) view debugging as a core element to be assessed in the development of computational thinking in K-12 classrooms. In 2012, Brennan and Resnick, researchers investigating Scratch, proposed three dimensions of computational thinking that are applicable in the K-12 context: computational thinking concepts, practices, and perspectives (students' understanding of their relationship with the technological world around them). Testing and debugging was identified as one of the four main parts in the practice

dimension which are "useful in a variety of design activities, not just programming" (Brennan and Resnick, 2012, p. 7).

However, when Lye and Koh (2014) conducted a literature review on the progress of K-12 computational thinking education in 2014, they found that 85% of studies investigated learning outcomes only in terms of computational concepts. This review concluded that to complete the picture of K-12 computational thinking education, more work need to address computational practices such as debugging. Similarly, Werner et al. (2012) designed an assessment tool to measure Computational Thinking in 325 students taking a middle school game-based programming course. They found that students scored lowest on the tasks that involved problem solving skills using debugging.

Moreover, for young novices, most programming environments are not intentionally designed to teach debugging. For example, Scratch (Resnick et al., 2009) and Alice (Cooper et al., 2000) seek to prevent the introduction of syntactic errors in the first place through designing visual components to directly manipulate objects. Such environments also help students 'debug' by making problem states, which include a character's properties and variable values, immediately visible on screen. However, the visibility of problem states provides very little help to novices to bridge the gap between current and desired state. When the objects behave differently than expected, there is no feedback to help novices locate the error and proceed. Other environments such as Greenfoot (Kölling, 2010) and BlueJ (Kölling et al., 2003) provide simple debuggers showing variable values at specified lines of code. However, novices need to first understand debugging as a problem solving process in order to effectively use the debugging tools. Moreover, novices may experience negative emotions when encountering bugs. For example, Kinnunen and Simon (2010) reported college freshmen expressed confusion, or more anguished frustration and anger during debugging. These studies show that debugging education would likely to be benefited from fun and engaging environments that foster problem solving skills.

To design such environments, we need to first improve our understanding on young students' debugging process. For this purpose, we designed a debugging feature in an educational game - BOTS (Hicks et al., 2014a). Twenty-two 6th-8th grade students played through 7 levels of debugging puzzles in 2-hour workshop sessions. We sought to address the following research questions:

RQ1: What are some problem solving behaviors and patterns students exhibited when debugging?

RQ2: How do students' problem solving behaviors relate to performance and prior programming experience?

RQ3: How well can young students participate in self-explanation before and after debugging, and how does the quality of self-explanations correlate to student problem solving patterns and performance?

RQ4: What are young students' perceptions of debugging?

The article starts with a review of problem solving stages, debugging environments for young novices, and learning analytics for programming (section 2). Then, section 3 shows the design of BOTS and its debugging features, the experiment, and the data analyze method. Section 4 presents the quantitative and qualitative results of the study. Section 5 discusses result and design implications. Finally, Section 6 outlines conclusions and future work.

2. Related Work

2.1. *Debugging and Problem Solving*

Computational Thinking is essentially a type of problem solving methodology (Barr and Stephenson, 2011; Grover and Pea, 2013; Wing, 2006) and debugging has been widely considered an essential problem solving skill. For example, Klahr and Carver (1988) identified five debugging phases for the Logo debugging curriculum: program evaluation, bug identification, program representation, bug location and bug correction. Yoon and Garcia (1998) proposed (1) a cognitive model of debugging that includes a comprehension strategy to identify discrepancies from the desired program, and (2) an isolation strategy to localize bugs, hypothesize causes, correct bugs and verify solutions. Both debugging models closely resemble the five stages in modern problem solving models (Bruning et al., 2010), as illustrated in Figure 1.

Figure 1. Five Problem Solving Stages

In the problem solving model, identifying the problem is considered as one of the most difficult stages. This stage requires domain knowledge, time and persistence. In the context of debugging, a shallow problem identification could be noticing the superficial differences in the outcomes between the desired and actual programs. A deeper identification, instead, should locate the bug by identifying where and why was wrong with the program. Domain knowledge facilitates the identification of the problem. For example, when a program never ends, an experienced programmer would immediately look into the program's exit conditions in iterative structures such as loop.

Representing the problem occurs both internally and externally, by either thinking abstractly or using external representations such as a flow chart. Both approaches can be done efficiently through identifying the current state and the goal state in the problem solving space. In the context of debugging, good representation reveals the underlying structures, common patterns, and discrepancies between the buggy and desirable programs.

Selecting Strategy means selecting the method to derive solutions; it does not mean selecting the possible solutions. One good problem solving strategy is mean-end-analysis, which means breaking a problem into sub-parts, and gradually approaching the goal state through completing subparts. In the context of debugging, one such strategy is to isolate bugs and correct one bug at a time. Another strategy is to identify subgoals towards goal state, and fix program to achieve these subgoals.

The successful implementation of a strategy heavily depends on the previous stages. Good problem solvers consider and evaluate more solutions when implementing a strategy. Lastly, the evaluating the solution phase includes the evaluation of both the final solution, and the whole problem solving process. Good problem solvers apply appropriate measurements, such as designing comprehensive unit tests.

Across domains, several problem solving patterns distinguish novices and experts. Experts spend more time at the initial stages of problem solving: experts are more

proficient with domain knowledge, and take more effort to understand the problem deeply before implementing solutions. Moreover, experts tend to break problems into more meaningful subparts and solve these subparts sequentially. Experts also consider more alternative solutions when reflecting on the problem solving process. Moreover, expertise in problem solving skills can be improved through accumulating general problem-solving knowledge, domain knowledge, and deliberate practice (Bruning et al., 2010). Because debugging is commonly considered as a problem solving process (Klahr and Carver, 1988; Yoon and Garcia, 1998), identifying and understanding novices and experts behavioral patterns would benefit debugging education.

The discrepancies between novices and experts at debugging have been studied extensively since the 1980s. Spohrer and Soloway (1986) found that contrary to common belief at the time, not all bugs were results of misconceptions about language constructs. Spohrer and Soloway (1986) concluded that bugs seem likely to occur when novices are unable to coordinate and integrate the goals and plans that underlie program code. Vessey (1985) classified novice and expert based on their abilities to chunk programs. She found that experts who chunk programs into meaningful pieces were significantly better in several measurements of debugging expertise, such as time cost and strategy use. Nanja and Cook (1987) suggested that fragile knowledge is what keeps novices from continuing in sophisticated problem solving strategies. In their work, fragile knowledge refers to misplaced, inert or mixed knowledge that is unable to be applied in context.

In more recent studies, Fitzgerald et al. (2008) found that locating bugs is more difficult than fixing them, suggesting the importance of identifying problems at early stages. Murphy et al. (2008) evaluated debugging logs and final solutions from college-level novices. They identified good debugging strategies including efficient use of "tracing", "testing", "understanding the code", "isolating the problem" and "considering alternatives". They identified bad strategies as inefficient use of some previously mentioned strategies, along with "worked around the problem"(e.g. Replaced code student did not understand with completely new code), "just in case" (did unnecessary and meaningless changes), and "tinkering" (randomly and usually unproductive changes (Murphy et al., 2008). These findings helped us identify and quantify students problem solving behaviors in our analyses.

2.2. *Debugging Game for Young Novices*

For decades, serious gaming has been an engaging and effective way to teach computational thinking and programming. One of the earliest efforts is LOGO Programming Language designed by Papert et al. in 1967 (Logo Foundation, 2015). In LOGO, children construct programs that direct the movement of a "turtle" object to draw graphics on screen. In 2014, Vahldick et al. reviewed a total of 40 games that are designed to teach programming, categorized as "LOGO-like" (drag-and-drop commands from a tool-bar), "Adventure Games" (hero explore the world to collect and interact with objects and characters), and "General Puzzles" (simulations, real time strategies and maze games). Vahldick et al. (2014) tagged the games based on the competences the games' activity saught to establish: comprehension, writing, and debugging. Surprising, they found while the majority of games' activity helped practice writing programs (32 out of 40), only 6 games' activities helped practice comprehending programs, and only 5 helped practice debugging.

One of the few gamified environments that were specifically designed to teach

debugging is Gidget created by Lee et al. (2014). Gidget follows a debugging-first principle, in which learners debug existing programs before creating new programs. The objective to fix the main character in the game - a self-blaming, fallible and cooperative robot. Students write code such as "scan bucket" and "goto crate" to order the robot character to perform actions and interact with objects. The Gidget interface also enables students to step through code, view changes in the environment, and receive feedback from the robot character such as "I could not find anything to scan by that name".

Lee et al. (2014) identified five counterproductive problem solving patterns from students debugging in Gidget: blindly trusted the correctness of the original code; deleted the original code without reading/understanding its clues; persisted in using non-applicable programming constructs that worked for earlier levels; asked for help before trying; and failed to adapt examples to specific contexts. Lee et al.'s (2014) study also reported that students overcame more algorithm design and learning phase barriers (Ko et al., 2004) in the programming activities if they had debugged existing programs. The biggest improvements were found in two learning phase barriers: design (knowing what to do), and understanding (knowing what to expect on program's external behavior). These results show the benefits of using educational games to teach debugging. However, Lee et al.'s (2014) study identified debugging strategy mainly from students' verbal scripts and field observations, with only 1 observer for 34 students. The study also did not interpret the results in the context of problem solving, or relate debugging strategies to students' performance or prior programming experience.

2.3. *Learning Analytics for Detecting Patterns in Programming*

To design a gamified environment that fosters problem solving and debugging skills, we must first understand how young students debug. One common approach is coding verbal scripts from a "think-aloud" process, during which participants verbalize their thoughts when solving the problem. In the Gidget study, Lee et al.'s (2014) Gidget study applied the "think-aloud" method to detect those counterproductive problem-solving strategies. Fitzgerald et al. (2005) conducted a study where 37 students were asked to "think-aloud" while working on multiple choice questions featuring code samples of arrays and loops. Later, Fitzgerald et al. (2008) followed a Grounded Theory-based approach, where researchers noted key phrases and strategies from transcripts, finalized strategies via a group-review process, and went back to code to determin which strategies were used for individual transcripts. This study found students deployed a total of 19 strategies with multiple strategies on each question. This study was also able to group strategies based on existing learning theory, connecting students problem solving to large bodies of literature. Loksa and Ko (2016) recorded participants "think-aloud" in a study where students were to write pseudo-code to solve problems related to while-loop usage. They coded "think-aloud" transcripts with 4 problem solving activities related to self-regulation and 5 types of self-regulation activities. These studies found that novice programmers do self-regulate, but in an infrequent, inconsistent and shallow fashion.

Other studies used human judgement to record students activity and analyze program artifacts. For example, Blikstein (2011) conducted an exploratory case study, where he collected log files of user actions and code snapshots from nine students in a three week programming assignment. Blikstein identified novice pro-

gramming strategies and compilation behaviors through visualizing simple measurements such as code size, frequency of compilations and number of errors across time. This case study suggested three student coding profiles (copy and pasteres, mixed-code, and self-sufficient), and three code compilations stages (initial exploration, intense code evolution, and final touch). Similarly, Murphy et al. (2008) conducted a study where students did six programming excersices followed by one debugging exercise containing 3-5 logic errors. The debugging logs were recorded each minute by observers and the final debugging solutions were later noted by two researchers, and finalized in consultation with a third. This work identified 35 distinct strategies in 12 categories (described in the last section of 2.1), and found cases where students employed these strategies effectively or not during debugging.

Another common approach is data mining. Data mining is able to reveal patterns from large number of participants, assisted by automatic data collection and cognitive modeling. Blikstein et al. (2014) used Natural Language Processing techniques to develop a metric to compare differences between programs, modeled students progress as Hidden Markov Model (HMM), and found patterns through clustering the paths student took through HMM. These interpretable patterns were paths through program state, revealed states where students clearly encountered difficulties, and were predictive of student midterm grades. Berland et al. (2013) clustered programming snapshots based on created features such as the number of action and logic primitives. They identified higher level programming and problem solving styles, and delineated initial construction of programming knowledge as a three-step explore, tinker, and refine process. This work also successfully quantified tinkering, refined and emphasized on its importance in the context of programming.

We applied the second approach in our study. While the think-aloud method yields rich information from students' perspective, our participants from grades 6 to 8 were too young to clearly verbalize their problem solving process in a "think-aloud" setting. Our debugging game also involved simple programming tasks and few participants, which makes a data mining approach less useful. Moreover, we observed game behaviors such as moving an object repetitively to see the animations frequently in the BOTS environment. This makes human judgement necessary when interpreting the programming artifacts in BOTS. Given these factors, this study used the BOTS system to log students actions and programming artifacts during gameplay. Later, we extracted features from logged data, and relied on human coders to identify barriers in problem solving strategies.

3. BOTS and Experiment Design

3.1. *BOTS*

BOTS (Hicks et al., 2014a) is a web-based game that teaches fundamental programming ideas to K-12 students. The BOTS gaming experience is very similar to Lightbot (Yaroslavski, 2014), Robozzle (Ostrovsky, 2009), and Program your Robot (Kazimoglu et al., 2012), where students attempt to write programs to guide a robot character through maze-like puzzles. Among these games, BOTS and Program your Robot are specifically designed for teaching and researching computational thinking. Compared to Program your Robot, which has been studied on college students, BOTS targets a much younger audience. The BOTS environment contains simpler objectives and interaction, more realistic 3D puzzles, and more

gamified features to foster creativity among K-12 students.

In BOTS, students solve each puzzle using loops, conditional statements, and functions, the robot must press a number of buttons either by standing on them or placing a crate on them. Most BOTS puzzles are designed to have repetitive patterns, which provide opportunities for students to optimize (shorten) their program through loops and functions. To encourage the practice of these programming structures, BOTS awards students platinum, gold, silver and bronze medals based on the length of their programs. In BOTS, students can play expert-designed puzzles, create new puzzles, and play puzzles created by other players. BOTS has scoreboards and other gamified elements to keep competitive players engaged, such as offering creative play in the form of level designing tools.

First time players must play through the BOTS tutorial. The BOTS tutorial contains eight levels of increasing difficulty, with pop-up instructions explaining the BOTS interface, game mechanics and programming concepts at the beginning of each level. The first tutorial level instructs students to move the robot in a straight line. The next five levels (Level 2-6) teach students to program the robot to make turns (Level 2), climb (Level 3), move boxes (Level 4), and move in repetitive patterns through loops and functions (Level 5-6). The rest levels (Level 7-8) present puzzles for students to practice the concepts learned in the previous levels. The very last level (Level 8) is the designed as the Boss Level to engage and challenge overachievers. The last level of the BOTS tutorial is shown in Figure 2, which requires both function and loop to solve the puzzle efficiently. An example of instructions in BOTS tutorial is shown in Figure 3.

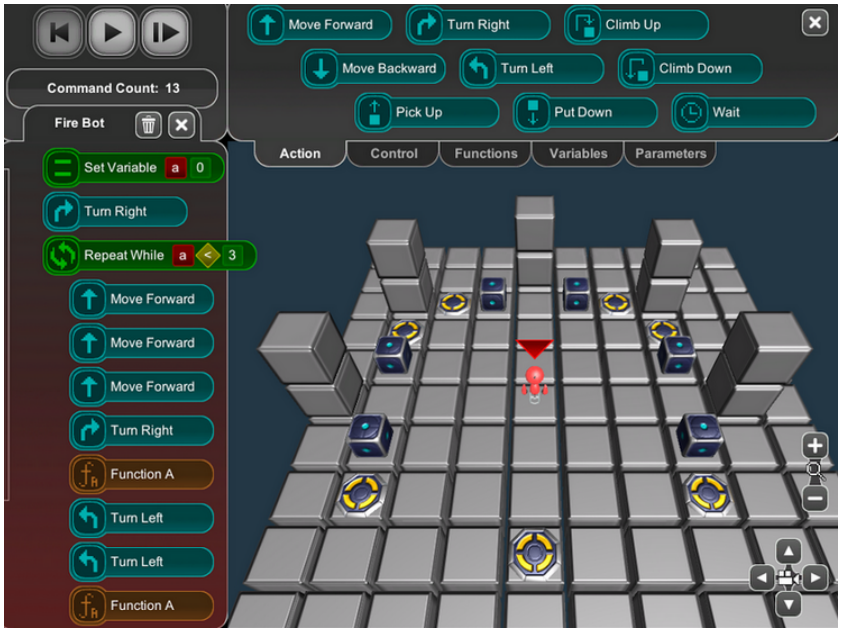


Figure 2. A student program from the last level of BOTS Tutorial. To pass this level, the robot (in red) must press a number of buttons (in yellow) either by standing on them, or placing a crate on them. The robot executes the program on the left panel, written by student dragging and dropping five types of code: action, control, functions, variables, and parameters (in the top right panel).

Previously, BOTS has been used to study user-generated content in serious games, particularly with regards to the design of the creative tools used to build such content. Researchers found that even the best players were likely to build

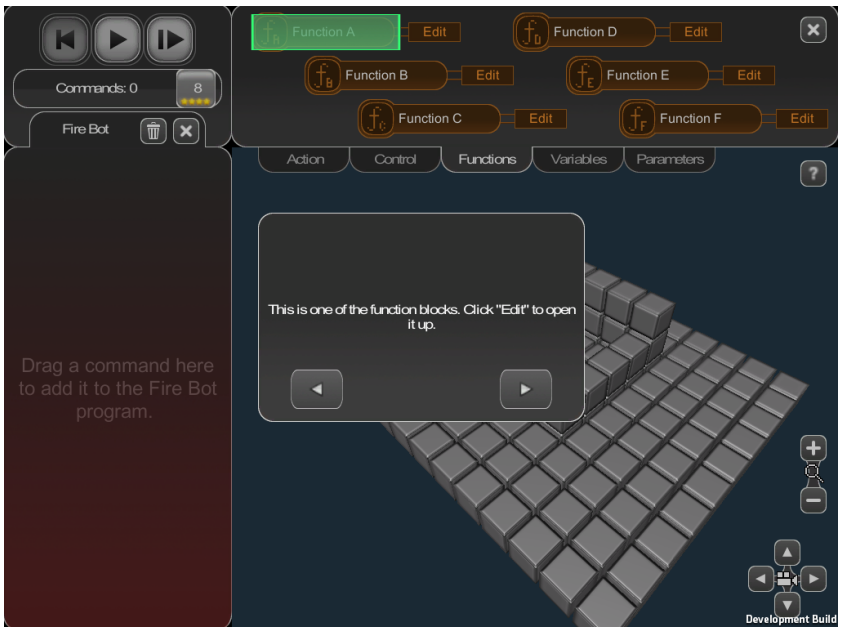


Figure 3. BOTS tutorial interface. At the beginning of puzzle solving, where dialog box explains the interface, or concepts of loops and functions if they are needed in this level. After reading instructions, students start to program from scratch.

puzzles which did not feature the game's core mechanics, and that implementing a "Solve and Submit" system which required players to provide solutions to their own puzzles was a small improvement on this front (Hicks et al., 2014a). Further work with this data showed that it is feasible to use student data to construct low-level hints for user-created levels without requiring expert coding or annotation (Hicks et al., 2014b; Peddycord III et al., 2014). A more effective approach to improving user-generated content was the implementation of gamified level editors whose capabilities correspond more closely with the game's mechanics; both a "problem-posing" style level editor and a programming-oriented level editor resulted in the creation of levels with better affordances for use of the game's core concepts (Hicks et al., 2016).

3.2. *BOTS Debugging Feature Design*

In each puzzle of BOTS debugging, students first view an animated scenario in which the robot character executed a buggy or incomplete program and failed the puzzle. Then, students debug the program using five types of debugging actions: deletion & un-deletion of the original code, addition & removal of new code, and running the code. We refer to the first four debugging actions as code edits in later analyses. To help students keep track of changes, student edits were highlighted to differentiate them from the original code. Another design is that when student deletes an original code, it will be shown on the screen as being crossed out; but if the student removes some of their own new code, it just disappears. This design helps students focus on addressing problems in original code. From the study by Lee et al. (2014), students were frequently observed to delete original code before reading. Leaving original code crossed out instead of disappearing gives student another chance to re-evaluate their decisions. Moreover, when solving the problem, students may add or remove massive amount of lines due to struggles, frustration or

gaming behaviors. Leaving only the original code crossed out avoid students from flushing the programming panel with wrong codes and lose focus. An example of the debugging interface is shown in Figure 4. Lastly, we used the same medal system in BOTS Tutorial to award efficient solutions. The shorter the editing distance between a solution and the original buggy program, the better medal the student gets.

Additionally, because the think-aloud method is not suitable for our young participants, we added a self-explanation feature to gain more insights from the students' perspective, and to encourage reflective thinking for educational purpose. Self-explanation happened twice at each puzzle. When student made the first edit on the original buggy code, a pop-up box asked students to explain what was wrong with the program. After student successfully debugged the program, another pop-up box asked students to submit or update the previous explanation. The pre-explanation was optional, considering that students may wish to experiment first, or may not understand the bug yet. The post-explanation was mandatory, because students have successfully debugged the program. An example of self-explanation feature is shown in Figure 5.

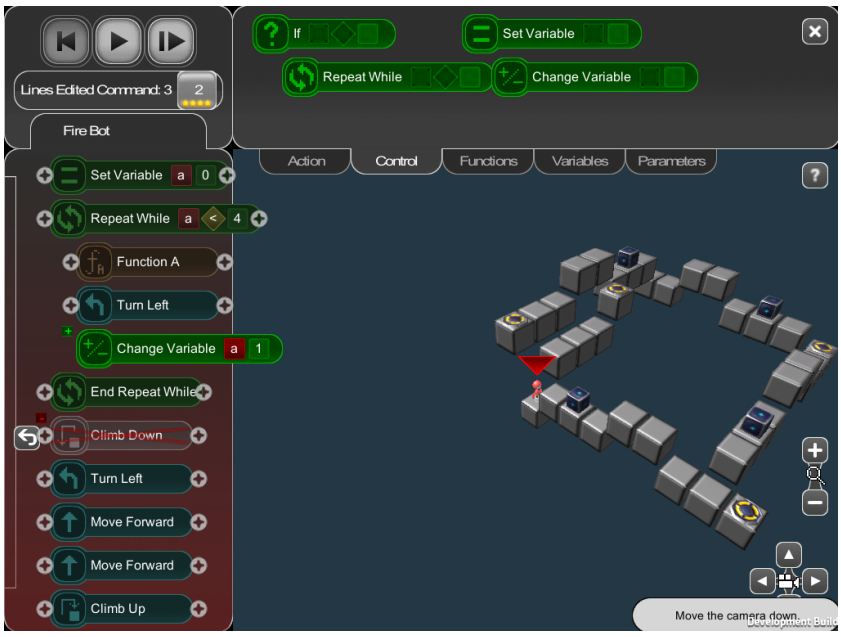


Figure 4. BOTS debugging interface, with a student debugging example (left) In the debugging feature, students edit existing buggy code. Students can perform four types of actions: delete buggy code (line 7), reverse deletion, add a new code (line 5), and remove it later.

We designed 7 puzzles with incomplete or buggy solutions for debugging, as shown in Table 1. These 7 puzzles covered the same programming concepts and structures taught in the first 5 levels of the tutorial. The lower level puzzles (1-3) aimed to familiarize students with the debugging feature and practice the initial stages of problem solving: identifying bugs, representing the problem and selecting problem solving strategies. As shown in Table 1, these levels contained missing actions, unnecessary steps and incorrect sub-goals. The higher level puzzles (4-7) aimed to practice the identification of multiple bugs, and bugs in more complicated programming concepts and structures such as counting variables, nested loops and functions. Similar to the BOTS Tutorial, the last puzzle (Level 7) was designed to

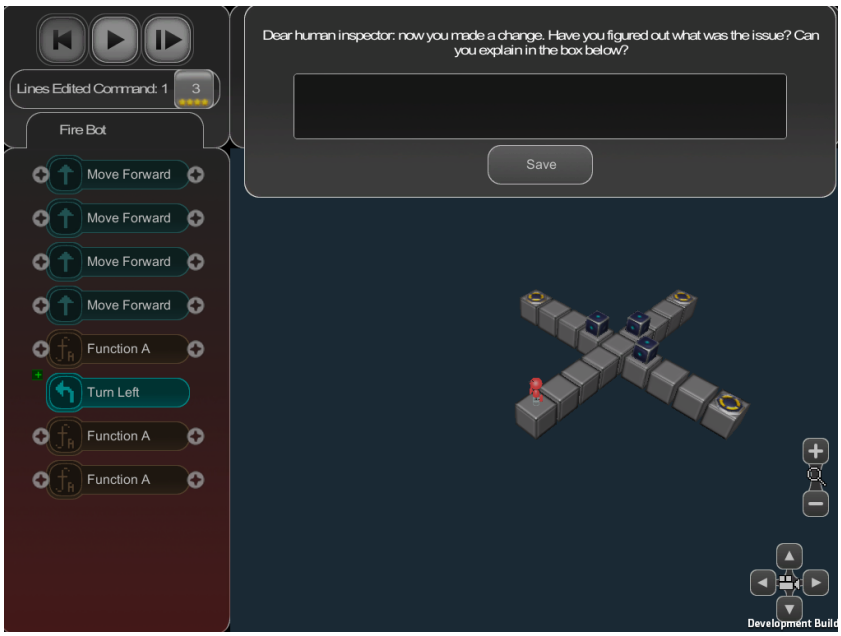
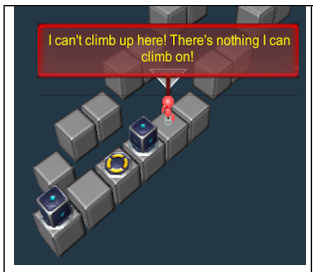
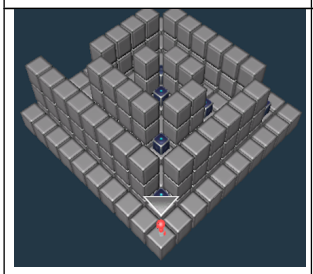


Figure 5. The self-explanation feature in BOTS debugging. This example shows the dialog box when students made their first code edits, where students had the option of entering text or saving an empty explanation. When student finished debugging, the dialog box will pop up again with the saved explanation from the first time. This time, students had to enter explanations by updating their previous answers, which was blank in this example.

be the Boss Level to engage and challenge over-achievers who finished the previous levels early.

Table 1.: The Debugging Puzzle Design (7 levels)

Puzzle	**Main& Func. Programs**		**Error**
	Turn Left Climb Up Turn Left Climb Up Climb Up Climb Up Climb Down	Climb Down Climb Down +Turn Right Move Forward Move Forward Move Forward Move Forward	Missing turn right.
	Turn Right Move Forward ~~Turn Left~~ Move Forward Move Forward Move Forward Move Forward	+Turn Left Move Forward Climb Up Climb Up Climb Up Turn Right Pick Up	Can't climb >1 block. The program also implements a path aims to picks up the crate unnecessarily

	Set Var a=0 Climb Down **While a<4** Move Forward {Climb Up Climb Down } Move Forward Move Forward Var a=a+1} Move Forward **End While** Move Forward Turn Right Pick Up Climb Down Move Backward Move Forward ~~Move Backward~~ Turn Right Put Down Move Forward Climb Up	The program puts a crater in a spot that blocks access to the next crate. Several crater & circled spot combinations are available.
	Set Var a=0 **End While** **While a<3** +Climb Up {Climb Up Climb Down Turn Left +Turn Right Climb Up Climb Down Turn Right Climb Down +Var a=a+1} Move Forward	The program does not increase the loop counter, resulting a never-ending loop. There are also missing steps outside the loop.
	Move Forward Pick Up Move Forward Move Forward Move Forward Move Forward Move Forward Move Forward +Turn Right Put Down Function A Move Backward Function A Move Backward Function A Move Backward **Function A:** +Turn Left	No commands to connect the repetitive use of Function A.
	Set Var a=0 **Function A:** **While a<4** Climb Up {Function A Pick Up Turn Left Move Forward Var a=a+1} Move Forward **End While** Climb Up Turn Left Move Forward Move Forward Climb Down Move Forward +Move Forward Climb Up Put Down	The program does not increase the loop counter, and misses a step in the function inside loop.
	Turn Right Function A} Set Var b=9 **End While** Function A +Move Forward Function A **Function A:** Function A +Set Var a 0 Set Var c=0 **While a<b** **While c<3** {Var a=a+1 {Var c=c+1 Move Forward} Var b=b-2 **End While** Function A Turn Left	The program does not initialize Var a each time the containing function is called. As a result, the loop is skipped on subsequent calls of Function A.

3.3. *Participants and the Experimental Process*

The experiment was conducted in two workshop sessions happened in the morning and afternoon of the same day, on the same site led by the same two instructors, with different groups of participants. The participants registered for the workshop from a pre-college program for science, technology, engineering and math (STEM) education, approved by IRB. A total of twenty-two 6th-8th graders participated. Among these students sixteen were males, and eighteen were non-Caucasians.

Each student was randomly assigned an anonymized ID to log into the BOTS

system. As first time players, students' first activity was to play through the BOTS tutorial. The tutorial consists of a sequence of levels which teach the game's basic mechanics, from how to add instructions to the robot, to how to use loops and functions. The first six levels introduce the game mechanics, while later levels combine previously introduced mechanics. The final "Boss Level" combines all game mechanics together in its optimal solution. Each level in the tutorial sequence has authored help available. Additionally, in this activity, instructors offered as much help as they could to ensure each student learned the game mechanics and programming concepts. The tutorial activity lasted for 45 minutes, during which students either completed the whole tutorial (reaching the last Boss Level), or completing at least the first six levels which completes the teaching of all the game mechanics and programming concepts. After the tutorial, students were given a pre-questionnaire. The pre-questionnaire contained three parts. The first part consisted of basic questions on anonymized ID, age, gender, and prior programming experience. The second part contained open-ended questions such as "what are your favorite and least favorite parts of this activity?". The third part contained Likert-scale questions such as "I enjoy playing this BOTS feature" scored on a range of 1-5, with 1 as strongly agree and 5 as strongly disagree. Two students who had played the BOTS tutorial in the previous year's workshop skipped the tutorial and proceeded directly to the pre-questionnaire.

Students' second activity was to play BOTS using the game's debugging feature. The debugging activity lasted 45 minutes. Additionally, at the start of the debugging activities, instructors told students that their debugging solutions could be used to teach students in the future who may get stuck on the same puzzle. During the debugging activity, instead of proactively offering help, instructors only helped students when they were clearly stuck and raised their hand. When offering help, instructors did not explicitly tell students what to do, but instead gave hints on where and why the code was wrong. During the debugging activity, BOTS logged the following data from students: anonymized student IDs, puzzle IDs, the text of student-authored puzzle explanations, debugging actions (the addition & removal of new code, deletion & un-deletion of original code, and running the code) and their timestamps. Student debugging activities were mapped to questionnaire responses through the same anonymized IDs.

As the final activity in this experiment, students were given a post-questionnaire. The post-questionnaire contained the same three question types as the pre-questionnaire. The only differences were that the post-questionnaire asked about the debugging activity instead of the tutorial activity.

4. Analyses Method and Results

In this section, we report our methods and results in the order of our four research questions. Table 2 provides a high level overview, listing the average performance on each level by all students who completed the level. From Table 2, students spent an average of 3-8 minutes on each level. Students encountered the most difficulties on Levels 4, 6 and 7 with the loop concept, and on Level 2 where the original program implemented an tedious path when alternative paths were available.

One student submitted two solutions for Level 2; we only used his/her first solution. Two students did not complete Level 6, but because they were near completion, their data were included in the analysis. These two students' performance

Table 2. Performance Statistics of the 7 Levels in Debugging Activity

Levels (#students completed)	**Time spent(s)**	**#compilations**	**#code edits**	**Edit dist of (student, expert) solutions**	**Pre/Post SE scores(#students scored > 0)**
1:Miss a command (22)	199.6	2.1	4.7	(1.0,1.0)	Pre(n=16):1.2; Post(n=19):1.6
2:Tedious path (22)	385.7	11.1	38	(7.1, 2.0)	Pre(n=8):0.4; Post(n=17):1.4
3:Blocked by itself (22)	160.3	2.6	6.0	(5.1, 1.0)	Pre(n=10):0.6; Post(n=18):1.1
4:Miss loop counter (22)	451.4	10.9	35.9	(6.0, 3.0)	Pre(n=12):0.6; Post(n=17):1.1
5:Boundary of Func. (22)	190.1	4.9	9.6	(3.5, 2.0)	Pre(n=9):0.5; Post(n=16):1.0
6:Func.+loop (20)	305.3	7.7	17.6	(2.5, 2.0)	Pre(n=7):0.4; Post(n=16):1.3
7:Func.+nested loops (12)	803.7	11.1	40.9	(8.1, 2.0)	Pre(n=2):0.3; Post(n=9):1.4

scores were estimated as the max (worst overall) performance scores of students who completed Level 6. Level 7, which was designed to be the Boss Level to engage over-achievers, was excluded from analyses because most students did not complete the level.

4.1. *RQ1: Problem Solving Behaviors in Debugging*

We wrote code to recreate program snapshots for each time a student ran their program based on the logged debugging actions. These snapshots showed the state of the program and the resulting positions of the robot and movable objects after the program completed. We also created descriptive statistics such as the time spent, and the frequencies of compilation and other debugging actions.

The process of interpreting student data in BOTS depends on human interpretation for two reasons. First, BOTS is designed as an open "playground" for young students. It is necessary to use human judgment to filter out off-task behaviors such as picking a box up and down repetitively to be entertained with the game's animation. Second, human interpretation suits the purpose of this small-scale pilot study, as commonly used in prior literature (Blikstein, 2011; Murphy et al., 2008).

First, we collected the snapshots, programs, and timestamps for student solutions. Next, from these data, the first two authors noted, discussed, and agreed on general patterns of behavior and mapped them to the conceptual and problem solving barriers. In discussion with the other two authors, these behavior patterns were refined and specific and measurable definitions for these patterns were created.

The rest of this subsection presents our results. We begin by describing the barriers students encountered in programming concepts (CB) and in the five problem solving stages (PB). Then, for each barrier, we summarize a list of measurable behaviors which relate to those barriers, and which we later used to categorize students debugging behaviors.

CB1. Shallow Understanding on Programming Concepts Even though

students wrote loops and functions multiple times in the tutorial, many encountered difficulties in debugging. Students first encountered a missing loop counter bug in Level 4, which happened after they saw a correct loop in Level 3. Unfortunately, only 5/22 students found the missing loop counter on the first attempt. Among these 5 students, 4 went back to edit the corrected loop when the program still did not work due to the missing commands outside the loop. Among the 17 students who did not figure out the missing counter on the first attempt, 8 attributed the error to the loop's upper bound; 4 did not understand the boundary of the loop, as they moved actions that belonged inside loop outside, or vice versa.

Students first encountered functions in Level 5, where the commands connecting the repetitive use of functions were missing. Although adding commands inside the function would give a shorter solution, 12/22 students wrote repetitive code outside of functions.

Students encountered a combined loop and function errors in Level 6. 14/22 figured out the loop's missing counter on the first attempt, 13/22 figured out missing commands inside functions, and 8/22 figured out both. Although this was an improvement from the previous levels, the result was still disappointing given that students also wrote loops and functions multiple times in the tutorial. Similarly, 3 students went back to change a corrected bug when their programs did not run due to another bug. These observed patterns show that debugging may require deeper understanding on programming concepts than writing code from scratch. This observation back up Nanja and Cook's (1987) argument, that novices experience difficulties in applying fragile knowledge during problem solving.

Measurable behaviors: a) Solved a bug on the first attempt. This behavior showed mastery in programming concepts. b) Replaced loop or function with repetitive codes, rather than fixing the original loop or function with fewer code edits. This behavior, similar to what observed in Murphy et al. (2008) as"worked around the problem", implied a lack of understanding on the benefit of loops and functions, or perhaps how to use them.

PB1. Barrier in Identifying the Problem Good problem solvers take time to understand the code, and identify where and why it was wrong. Novices tend to identify the superficial difference in program outcomes. Across levels, we observed students editing code before locating bugs. In these cases, students simply assumed that the line of code where the robot stopped executing was the bug's location. A barrier in identifying the problem can be seen in Level 1 where the robot character walks in a straight line and stops at the puzzle's edge due to a missing turn-right command. In this puzzle, 8/22 students inserted the missing command immediately after the stopped line instead of the correct location several lines before. Similarly, 10/22 students in Level 4 (missing loop counter) and 6/22 students in Level 5 (function) started debugging by editing where the robot stopped instead of where the bug was located. This observation corresponded with previous research (Bruning et al., 2010) that novices do not spend enough time in the initial problem solving stage, identifying the problem.

Measurable behaviors: a) Multiple compilations before editing. We interpreted this as students taking time to identify the problem. b) Edited code line on which the robot stopped, instead of the code that actually caused the problem.

PB2: Barrier in Representing the Problem Good problem solving strategies begin with a good representation of the problem that reveals the discrepancies between the buggy and desirable programs, underlying structures, and common

patterns. Novices may make unnecessary changes if they do not create a correct representation of what the buggy problem accomplishes. A barrier in representing the problem is observed in Level 3, where the robot character puts a crate down which then blocks the robot from accessing the other buttons (yellow circles) to solve the puzzle. Students can solve the puzzle by fixing existing code in multiple ways, or by adding new code to pick the wrongly-placed crate back up and move ahead to complete the puzzle. Surprisingly, 0 students moved ahead from where the robot stopped, even though this strategy was much easier than the next optimal solution of fixing the original code implemented by 5/22 students. This observation implies that students simply viewed debugging as fixing broken code. Student did not transform the problem into a problem solving space, and did not identify the discrepancies between the current state and the goal state. This barrier was also observed in Level 2, where 3 students changed the robot's path through massive deletion of the original code, even though the bug is relatively small. Instead, students should represent the problem to recognize the common patterns between the original and new solutions, and reuse the code that implement these common patterns.

Measurable behaviors: a) Unnecessary massive deletion. Deleting code without recognizing re-usable code. This pattern is similar to the "reinvent the wheel" counterproductive problem-solving strategy found by Lee et al. (2014) in the Gidget debugging game, where "a player deletes the original code without reading it and misses out on clues the code provides" (p. 61).

PB 3: Barrier in Selecting Strategies Good problem solving strategies often break the problem into meaningful sub-parts, then isolate and solve sub-parts sequentially, moving closer and closer to the goal state. In BOTS, some students run their program on completion of each of these sub-goals, demonstrating a good problem solving strategy of breaking problems into steps. An barrier in selecting strategies is observed in some novice solutions to Level 2. Students who did not encounter this barrier first fixed the code for climbing up the hill and picking up the box, and ran the code. Next, these students wrote code to climb down the hill, and ran the code again. Lastly these students wrote code to walk towards the destination crate, and ran the code. However, many students instead ran their programs very frequently and after minor edits that changed less than two lines, demonstrating "tinkering" instead of working towards a subgoal.

Measurable behaviors: a) Frequent runs on minor edits. This behavior, similar to what Murphy et al. (2008) observed as "tinkering" (random and usually unproductive changes), could be caused by failing to focus on meaningful sub-parts of the problem (Vessey, 1985), or a lacking of awareness or confidence on the outcome introduced by code edits.

PB4: Barrier in Implementing the Strategy Good problem solvers consider and evaluate different solutions when implementing the strategy. Students may encounter a barrier by becoming "locked in" to one solution, even though that solution may be sub-optimal. Instead, students should consider other paths, or consider alternatives once as they encounter difficulties. A barrier in implementing the strategy can be observed in Level 2, where the program as written implements a tedious and incomplete path involving climbing up the hill to pick up a crate unnecessarily. Even though BOTS awards medals to more efficient solutions, only 2/22 students started by switching to a better path that did not pick up the crate, costing less debugging actions to drive a much shorter solution. 7/22 solved

the puzzle with the original tedious path; 13/22 students switched to the better path only after midway; among these 13 students who switched midway, 5 clearly struggled with the original path and 3 switched paths through massive deletion. Many of these students asked for help, and instructors told them to "consider other paths to solve the puzzle" during the activity.

Measurable behaviors: a) Starting with the tedious path. When two strategies are available, students started with the obviously more tedious path (using fewer loops or functions, or involving unnecessary actions).

PB5: Barrier in Evaluation Good problem solvers choose appropriate measurements to evaluate the product. Instead, students may incorrectly identify why their attempt has failed, and undo their last actions instead of looking for further problems. A barrier in evaluation is observed in multi-bug puzzles. Some students did not know when a specific bug was fixed. After fixing the first bug, when the code did not solve the puzzle due to additional bugs, students went back to change the first bug even if correctly fixed. Examples are described in CB1. This may indicate that used solving the puzzle as the single measurement to evaluate code. Instead, students should evaluate each individual bug, and focus on whether the code edit introduced desirable behaviors.

Measurable behaviors: a) Undid a corrected bug. Players undid a correctly-fixed bug instead of looking for other problems.

4.2. *RQ2: Problem Solving Behaviors, Performance, and Prior Programming Experience*

Table 3. RQ2: Correlations between Problem Solving Patterns, Performance and Prior Programming Experience

Problem Solving Patterns	**Time Spent(s)**	**# Code Edits**	**Edit Dist.**	**Programming Exper.**
CB1-a. Solved on 1st attempt (Mf=4.18)	-0.52*	-0.77**	-0.17	0.42
CB1-b. Replaced loop/func. (Mf=0.27)	0.35	0.39	0.50	-0.14
PB1-a. Compiled before editing (Mf=1.41)	0.45	0.20	-0.03	-0.21
PB1-b. Edited where stopped (Mf=1.64)	0.42	0.37	0.05	-0.43
PB2-a. Massive deletion (Mf=1.18)	0.17	0.63**	0.29	-0.12
PB3-a. Run on minor edits (Mf=1.4)	0.55*	0.68**	0.03	-0.40
PB4-a. Started with tedious path (Mf=1)	-0.36	-0.07	0.13	0.36
PB5-a. Undid a corrected bug (Mf=0.27)	-0.43	-0.03	-0.23	0.03

*Note: Green and red indicate positive and negative correlations with $p < .05$. Darker colors indicate statistically significant correlations after Benjamini-Hochberg Correction, with ** $p < .05$, and * $p < .10$.*

Using the identified behaviors, the first author then counted the frequencies of these measurable behaviors across levels to create problem solving patterns for each student. These problem solving patterns were then correlated with students' performance measurements, as well as their self-reported prior programming experience. Student performance measures included time spent, number of code edits from debugging actions, and edit distance from the original code to the final solution.

Low edit distance implies that the student precisely located the bug, and understood the partial program well. We hypothesized that students with better

debugging behaviors and more programming experiences were faster at debugging (lower time spent), better at explaining the bug (higher Self-explanation scores), and had a lower edit distance. After calculating the performance measures, we normalized them to be within [0,1] at each level, so that levels were considered with equal importance when calculating the average across levels. We used the formula $(value - min) \div (max - min)$ where min was either the smallest time spent by students, or the minimum possible numbers of debugging actions and solution edit distance. Max was the largest statistic from students that was within 3 standard deviations of the mean; values above the Max were replaced with 1. Prior program experience was collected from the pre-questionnaire, reported by students in the format of open response. The responses were put into three bins: "less than 1 years"(10 students), "1-2 years"(6 students) and "more than 2 years"(6 students).

Table 3 reports the r-values of Spearman correlation between student problem solving patterns with student performance measure and prior programming experience. We used Spearman correlation because it does not assume straight linear relationship between variables, and normal distribution within variables. The first column reports the identified behaviors and their mean frequencies across levels. For example, a $Mf = 4.18$ on the first row means that students solved an average of 4.18 bugs on their first attempts, out of the 8 bugs from the 6 levels in debugging activities.

4.3. *RQ3: Self-explanation(SE)*

We then investigated students' self-explanations (SE). BOTS logged two self-explanations for each puzzles: when students did their first code edits, and when students completed debugging. These self-explanations were scored by the first two authors who designed the debugging puzzles. A score of 0 represents meaningless (e.g. "hiiii") or incorrect explanations; a score of 1 represents a shallow explanation that described the robot character's behaviors but did not relate to solving the problem (e.g. "the robot could not move forward"); a score of 2 represents explanations that identified the cause of the problem in the program (e.g. "the robot could not move because of a missing loop counter"). The two coders agreed on all explanation scores after discussion. Note that, due to the young ages of students, SE is a limited measurement of their ability on this task. However, the activity itself is pedagogically beneficial to students. Therefore we use this score mainly to provide insights on students' understanding of the problem.

Table 2 at the beginning of section 5 reports the number of students who participated in SE and their average SE scores. Across levels, the average post-explanation scores were between 0.4 and 1.0 higher than the pre-explanation scores, with over 70% students input correct and meaningful post-explanations. This result show that students improved their understanding on the original program through debugging. An example of this improved understanding is that a student wrote "the character turned so soon" in pre-explanation, and "the loop does not have a change [changing] variable" in post-explanation.

Table 4 reports the r-values of Spearman Correlation between students' pre/post explanations, problem solving patterns, performance, and prior programing experience.

Table 4. RQ2: Correlations between Problem Solving Pattern, Performance and Prior Experience

Problem Solving Patterns	**Pre SE**	**Post SE**
CB1-a. Solved on 1st attempt (Mf=4.18)	0.42	0.62**
CB1-b. Replaced loop/func. (Mf=0.27)	0.09	-0.22
PB1-a. Compiled before editing (Mf=1.41)	-0.37	-0.25
PB1-b. Edited where stopped (Mf=1.64)	-0.46	-0.45
PB2-a. Massive deletion (Mf=1.18)	-0.27	-0.38
PB3-a. Run on minor edits (Mf=1.4)	-0.44	-0.40
PB4-a. Started with tedious path (Mf=1)	0.23	-0.07
PB5-a. Undid a corrected bug (Mf=0.27)	-0.14	0.11
Performance		
Time Spent(s)	-0.25	-0.37
# Code Edits	-0.37	-0.71**
Edit Distance	0.16	-0.01
Prior Programming Experience	0.36	0.29

*Note: Green and red indicate positive and negative correlations with p <.05. Darker colors indicate statistically significant correlations after Benjamini-Hochberg Correction, with ** p<.05, and * p<.10.*

4.4. *RQ4: Students Perceptions of Debugging*

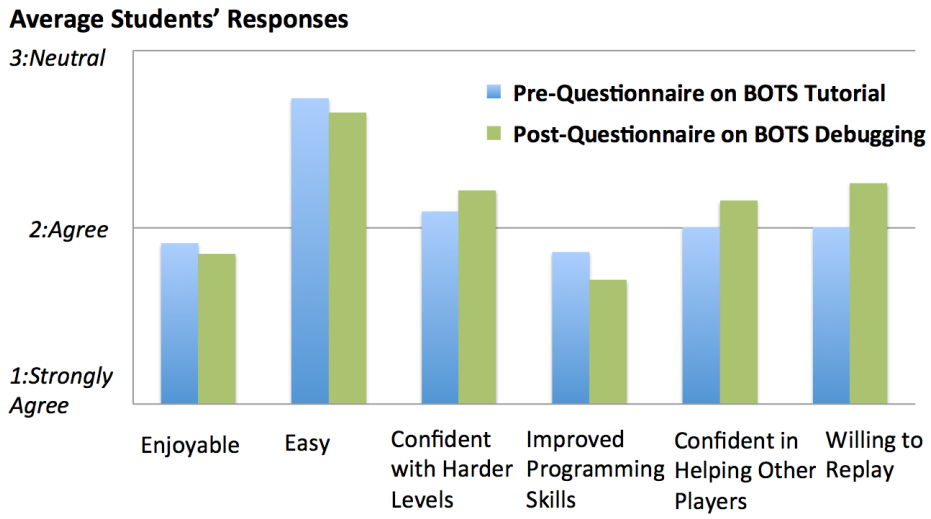


Figure 6. Pre/Post-Questionnaires' Results

We found no significant differences in the six Likert-type answers between the pre and post questionnaires, as reported in Figure 6. The majority of students felt positively about both the tutorial (where they wrote code from scratch) and debugging activities in BOTS. The 6-8 graders in this experiment found that debugging in the BOTS gamified environment was equally fun and engaging compared to programming.

Students' favorite parts of debugging activities included the feeling of learning, focused, and achieving. Some students wrote:

"I think that it is a great way to learn the basics of coding."

"The best part is figuring out exactly what's happening."

"I liked the debugging because I get to see some cool code."
"debugging makes me feel focused."
"You get very excited when you beat the levels.".

Students' least favorite parts included certain interface designs such as "the robot moved too slow", and the difficulties as the natural of the debugging activity. However, these difficulties are often mentioned together with learning and engagement. As students wrote:

"My favorite part about debugging is knowing I made the game better. My least favorite part is you have to really think [and] concentrate."
"Favorite- learning; Least favorite - being confused."
"It's very fun yet is very difficult."

5. Discussion & Implication

The goal of this study is to understand young students' problem solving behaviors in a gamified debugging environment, and to use our understandings to refine the environment. In this section, we summarize our answers to our four research questions and discuss their implications to future K-12 debugging environment design.

5.1. *RQ1: Problem Solving Behaviors in Debugging*

Our first research question is "What are some problem solving behaviors and patterns students exhibited when debugging?". To answer this question, we relied on human interpretations of logged student debugging actions. We found 1 conceptual barrier, from which we concluded that debugging requires deeper understanding of programming concepts than programming. We also found 5 barriers that corresponded to the 5 problem solving stages: identifying the problem, representing the problem, selecting a strategy, implementing the strategy, and evaluation. The list below summarizes what we learned from twenty-two 6th-8th grade students debugging in this experiment:

- In BOTS, debugging required deeper understanding of programming concepts than solving the puzzles from scratch. Many students successfully wrote loops in tutorial, but were unable to debug incorrectly written loops in the debugging activity.
- Some students did not spend enough time identifying the problem during the initial stage of debugging. Many students started by deleting the line where the program stopped, instead of several lines previous where the bug actually occurred.
- Some students treated buggy programs as entirely wrong, instead of as incomplete or partially correct solutions. These students lacked the ability to correctly represent the consequences of the program, and identify the discrepancies between the current state and the goal state. In other words, some students did not know how close the current program was to the solution, and how to minimally edit existing code rather than delete and start over.
- Some students lacked the ability to break down problems into meaningful sub-parts. For example, some students compiled and ran their program after very small edits of one or two lines, instead of compiling and running when

they believed a sub-goal had been reached.
- Some students did not consider alternative solutions early enough, or were inflexible when encountering difficulties. For example, some students failed to consider whether a better path existed than the one in the partial solution, and switched to a better path only after encountering much difficulty.
- Some students did not correctly locate which bug is causing an identified problem when multiple errors are present. For example, this could cause them to reconsider their work on the previous bug rather than searching for any additional bugs.

These observed behaviors correspond to the conclusion from prior work that novices lack domain knowledge and general problem solving skills (Bruning et al., 2010; Spohrer and Soloway, 1986). Thus, designers for future debugging environment should consider helping students develop general problem solving knowledge. As prior work found that locating bugs is more difficult then fixing them for novices (Fitzgerald et al., 2008), special attention should be paid at the initial stages of problem solving. For example, in the BOTS environment, designers can scaffold a debugging task into sub-goals corresponding to problem solving stages, ask students to mark and explain where can be wrong, guide students on good problem solving strategies, or ask students to list out possible solutions before editing codes. Another approach would be to provide feedback based on the detection of certain behaviors. For example BOTS could be modified to intervene when detecting massive code deletion of frequent compilation on minor changes, by displaying a pop up message to remind students of recognize similar patterns between solutions, or demonstrate a way to break problems into steps.

5.2. *RQ2: Problem Solving Patterns, Performance, and Prior Programming Experience*

Our second research question is "How do students' problem solving patterns relate to performance and prior programming experience?". In general, our results indicate that students' problem solving behaviors are related to their self-reported prior programming experiences, and that in turn we can infer some performance aspects from detected student behaviors. More specifically, we found that many problem solving patterns correlated with the time spent and code edits in the process, but not with the edit distance in the final solution. The only pattern which correlated with code edit distance was the "replace loop/function with code", and the significance of this correlation disappeared after preforming a False Discovery Rate correction. We hypothesized that more advanced students would be able to precisely locate the problem and fix it with minimal changes, so this result was somewhat surprising. One possible explanation is that some problem solving behaviors, such as "run on minor edits", require less cognitive effort than running on the completion of meaningful sub-goals. This could help students to focus and reach correct solutions, at the cost of longer time and more code edits during debugging. This could be especially true for students with less proficiency in programming

Additionally, student prior programming experience was found to be positively correlated with frequency of solving on 1st attempts, and negatively with the frequency of editing where the robot stopped. However, these correlations are not significant after False Discovery Rate correction is applied. Nevertheless the direction of this result is expected based on prior literature, which concluded that

problem solving experts have more "domain knowledge" and take more time at initial stage of problem solving, such as identifying the problem (Bruning et al., 2010; Fitzgerald et al., 2008). As an adaptation the the system, BOTS could be modified to pop up messages to encourage them spending more time on understanding the code before editing in cases when students report low prior programming experiences or demonstrate low performance.

5.3. *RQ3: Self-explanation*

Our third research question is "How well can young students participate in self-explanation before and after debugging, and how does the quality of self-explanations correlate to student problem solving patterns, performance and prior programming experience? ". We found that participation in pre-explanation varies per game level, but over 70% students input correct and meaningful post-explanations across levels. Moreover, after students successfully debugged, the average post-explanation scores were higher than average pre-explanations at all levels. Even though prior work would suggest that 6-8 graders often disengage in this kind of voluntary workshop activity, our results show that most students in this workshop participated, and their self-explanation quality reflected their improved understanding of the problem. The high engagement in self-explanation activity may be resulted from the instructors' telling students that their solution could be used to help other students who were stuck in the same puzzles. Additionally the verbal encouragement could make students perceive their efforts as important and useful, as some students wrote "make the game better" and "help others" as their favorite BOTS debugging experience in the questionnaires. As shown in earlier work by Aleahmad et al. (2008) on community authored problems, users are more invested when they are given evidence that their activity will have a real impact. Thus, we believe that BOTS' designers could improve the perceived value of learning activities, and perhaps enhance students motivation to participate through informing students on the value and purpose of their activities. For example, designers can call out that the program they are debugging could be another user's program, and their work could directly help that user.

For the second part of the question, our result implies that students' understanding of the problem influenced their problem solving behaviors, and in return, different problem solving behaviors related to students' understanding after they successfully debugged. We found that post-explanation scores and the number code edits were negatively significantly correlated, with $r = -0.71, p < 0.05$. Prior result shows that the number of code edits is positively significantly correlated with the frequency of "Massive deletion" and "Run on minor edits" behaviors. A high number of code edits may also due to multiple attempts to find a solution, or attempting an inefficient solution such as replacing loops and functions with repetitive codes. On the other hand, post-explanation scores where positively significantly correlated with the frequency of "solved on 1st attempt" with $r = 0.62, p < 0.05$. Our results suggest that even though some students successfully debugged solutions, the struggles in the process may distract them from understanding the problem or engaging in post-explanation activities. Thus, we suggest that BOTS should be modified to intervene when detecting certain problem solving patterns, even when these patterns may not affect the final solution.

Additionally, pre-explanation scores were correlated with two measurable behaviors at the earlier stages of problem solving, however, they were not significant after

false discovery rate correction. This result does suggest that weak pre-explanation scores could result from the barriers in the earlier stages of problem solving. For example, students who did not spend time identifying the problem would simply edit on the line where the robot stopped, and did not know what to explain when the pre-explanation box popped up. Thus, we recommend that BOTS could be modified to intervene early when detecting missing pre-explanations. For example, with messages encouraging students to spend more time on the earlier problem solving stages.

5.4. *RQ4: Students Perceptions of Debugging*

Our last research question is "What are young students' perceptions of debugging?". We conducted a pre-questionnaire after students completed the BOTS tutorial where they wrote code from scratch, and a post-questionnaire after students played through the BOTS debugging feature. We found no significant differences in responses, which shows that students perceived debugging similarly to programming in this gamified environment. From open-ended responses we found that our students generally viewed debugging as a positive yet intellectually challenging experience, as one student wrote "It's very fun yet is very difficult". These results indicate that gamified environments have the potential to positively influence students learning experiences with debugging activities.

5.5. *Conclusions & Future Work*

to summarize, in this study, we analyzed the problem solving behavior of 6th-8th grade students when presented with debugging exercises in BOTS programming game. We identified student behaviors in relation to programming concept and problem solving stages. We found correlations between student behaviors, prior programming experience, and performance (self-explanation scores, the time spent on puzzles, the number of code edits, and the edit distance from original code to solution). We found that students perceived debugging in this gamified environment as fun and intellectually challenging. Our results suggest that designers for future debugging environments to educate students on general problem solving knowledge, to consider early interventions, to inform value and purpose of similar activities, and to consider gamification.

One intuitive next step is to integrate the design implications learned from this study to BOTS debugging feature. To evaluate the effectiveness of new designs, it is necessary to improve on controlled experimental design and data-collection method. For example, researchers can conduct face-to-face interviews or develop metrics for coding observed behaviors on site. Researchers can also improve the assessment of psychology responses through questionnaires. For example, applying the method in the experiment by Moser et al. (2012) to rapidly assess game experience in public setting.

Another future direction is to implement a learning gain assessment. This study showed that students improved self-explanation scores after debugging, and many commented that they learned and benefited from debugging activities in questionnaires. However, stronger assessments are needed to answer whether debugging helped students learn. Future work could quantify how much students learned, and whether they can transfer knowledge to general programming outside this game

through designed pre- and post- tests. It could also be interesting to compare learning gains between programming, debugging, and a combination of both in a controlled experimental design. This design would provide valuable information on when to present debugging questions that best help students learning.

Lastly, as future work creates more debugging solutions, researchers may crowdsource these peer debugging solutions to create hints (Liu, 2015). Previously, Peddycord III et al. (2014) applied the Hint Factory method created by Barnes and Stamper (2008) in BOTS, and generated next-step hints from students' past solutions. However, this work found many dead-end states where too few students derived solution from these states to generate hints. These dead-end states may represent important conceptual error or creative problem solving solution. Thus, future work can recycle these dead-end states as puzzles in BOTS debugging activity. Students will be able to help their peers who stuck in dead-end state by providing debugging solutions from which we can derive next-step hints. Moreover, filtered self-explanations in debugging can also be used as higher-level hints, which may be more pedagogically beneficial than next-step hints.

References

Aleahmad, T., Aleven, V., and Kraut, R. (2008). Open community authoring of targeted worked example problems. In *the 9th International Conference on Intelligent Tutoring Systems*, pages 216–277.

Barnes, T. and Stamper, J. (2008). Toward automatic hint generation for logic proof tutoring using historical student data. In *the 6th International Conference on Intelligent Tutoring System*.

Barr, V. and Stephenson, C. (2011). Bringing computational thinking to k-12: what is involved and what is the role of the computer science education community? *Acm Inroads*, 2(1):48–54.

Berland, M., Martin, T., Benton, T., Petrick, C., and Davis, D. (2013). Using learning analytics to understand the learning pathways of novice programmers. *Journal of the Learning Sciences*, 22(4):564–599.

Blikstein, P. (2011). Using learning analytics to assess students' behavior in open-ended programming tasks. In *the 1st International Conference on Learning Analytics and Knowledge*, pages 110–116.

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. *Journal of the Learning Sciences*, 23(4):561–599.

Brennan, K. and Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. In *Annual American Educational Research Association meeting*.

Bruning, R., Schraw, G., and Norby, M. (2010). *Cognitive Psychology and Instruction*. Pearson, New York, NY, USA.

Cooper, S., Dann, W., and Pausch, R. (2000). Alice: a 3-d tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*, 15(5):107–116.

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):93–116.

Fitzgerald, S., Simon, B., and Thomas, L. (2005). Strategies that students use to trace code. In *the 2005 international workshop on Computing education research. ACM.*, pages 69–80.

Grover, S. and Pea, R. (2013). Computational thinking in k–12 a review of the state of

the field. *Educational Researcher*, 42(1):38–43.

Hemmendinger, D. (2010). A plea for modesty. *Acm Inroads*, 1(2):4–7.

Hicks, A., Catete, V., and Barnes, T. (2014a). Part of the game: Changing level creation to identify and filter low-quality user generated levels. In *the International Conference on the Foundations of Digital Games*.

Hicks, A., Liu, Z., and Barnes, T. (2016). Measuring gameplay affordances of user-generated content in an educational game. In *the International Conference on Educational Data Mining*.

Hicks, A., Peddycord III, B., and Barnes, T. (2014b). Building games to learn from their players: Generating hints in a serious game. In *the International Conference on Intelligent Tutoring Systems*.

Kazimoglu, C., Kiernan, M., Bacon, L., and Mackinnon, L. (2012). A serious game for developing computational thinking and learning introductory computer programming. *Procedia-Social and Behavioral Sciences*, 47:1991–1999.

Kinnunen, P. and Simon, B. (2010). Experiencing programming assignments in cs1: the emotional toll. In *the Sixth international workshop on Computing education research. ACM*, pages 77–85.

Klahr, D. and Carver, S. M. (1988). Cognitive objectives in a logo debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology*, 20(3):362–404.

Ko, A. J., Myers, B. A., and Aung, H. H. (2004). Six learning barriers in end-user programming systems. *IEEE Symposium on Visual Languages and Human Centric Computing*, pages 199–206.

Kölling, M. (2010). The greenfoot programming environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):14.

Kölling, M., Quig, B., Patterson, A., and Rosenberg, J. (2003). The bluej system and its pedagogy. *Computer Science Education*, 13(4):249–268.

Lee, M., Bahmani, F., Kwan, I., Laferte, J., Charters, P., Horvath, A., Luor, F., Cao, J., Law, C., Beswetherick, M., Long, S., Burnett, M., and Ko, A. (2014). Principles of a debugging-first puzzle game for computing education. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, pages 57–64.

Liu, Z. (2015). Data-driven hint generation from peer debugging solutions. In *the 8th International Conference on Educational Data Mining*.

Logo Foundation (2015). Logo History. Retrieved from `http://el.media.mit.edu/logo-foundation/what_is_logo/history.html`.

Loksa, D. and Ko, A. J. (2016). The role of self-regulation in programming problem solving process and success. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, pages 83–91. ACM.

Lye, S. and Koh, J. (2014). Review on teaching and learning of computational thinking through programming: What is next for k-12? *Computers in Human Behavior*, 41:51–61.

Moser, C., Fuchsberger, V., and Tscheligi, M. (2012). Rapid assessment of game experiences in public settings. In *the 4th International Conference on Fun and Games*.

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 Bulleti*, 40(1):163–167.

Nanja, M. and Cook, C. R. (1987). An analysis of the on-line debugging process. In *Empirical studies of programmers: second workshop*, pages 172–184. Ablex Publishing Corp.

Ostrovsky, I. (2009). Robozzle online puzzle game. *RoboZZle Online Puzzle Game*.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc.

Peddycord III, B., Hicks, A., and Barnes, T. (2014). Generating hints for programming problems using intermediate output. In *the 7th International Conference on Educational Data Mining*.

Resnick, M., Maloney, J., Monroy-Hernndez, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., a. S. J., Silverman, B., and Kafai, Y. (2009). Scratch:

programming for all. *Communications of the ACM*, 52(11):60–67.

Spohrer, J. C. and Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, 29(7):624–632.

Vahldick, A., Mendes, A. J., and Marcelino, M. J. (2014). A review of games designed to improve introductory computer programming competencies. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, pages 1–7. IEEE.

Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494.

Werner, L., Denner, J., Campe, S., and Kawamoto, D. C. (2012). The fairy performance assessment: measuring computational thinking in middle school. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 215–220. ACM.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, 49(3):33–35.

Yaroslavski, D. (2014). Lightbot. *Retrieved on 25th May*.

Yoon, B. D. and Garcia, O. N. (1998). Cognitive activities and support in debugging. In *Fourth Annual Symposium on Human Interaction with Complex Systems*, pages 160–169.