The Impact of Data-driven Positive Programming Feedback: When it Helps, What Happens when it Goes Wrong, and How Students Respond

Preya shabrina
NC State University
pshabri@ncsu.edu

Samiha Marwan
NC State University
samarwan@ncsu.edu

Min Chi
NC State University
mchi@ncsu.edu

Thomas W. Price
NC State University
twprice@ncsu.edu

Tiffany Barnes
NC State University
tmbarnes@ncsu.edu

ABSTRACT

This paper uses a case-based approach to investigate the impact of data-driven positive feedback on students' behaviour when integrated into a block-based programming environment. We embedded data-driven feature detectors to provide students with immediate positive feedback on completed objectives during programming. We deployed the system in one programming homework in a non-majors CS class. We conducted an expert analysis to determine when data-driven detectors were correct or incorrect, and investigated the impact of the system on student behavior on the homework, specifically in terms of time they spent in the system. Our results highlight *when data-driven positive feedback helps students*, *what happens when it goes wrong*, and *how this impacted students' programming behavior*. Results from these case studies can shed light on the design of future data-driven systems to provide novices with the positive feedback that can help them persist while learning to program.

Keywords

Snap, Block Based Programming, Data-Driven Hints, Positive Feedback, Adaptive Feedback

1. INTRODUCTION

Block-based programming environments are intended to provide novices with the ability to engage in motivating open-ended and creative programming with features that limit syntax errors but allow for simplified programming for interactive media [3, 2]. Some of these environments also include automated support such as misconception-driven feedback [4], next-step hints [6], or adaptive feedback [5], which have been shown to improve students' learning. Recently, data-driven automated hints and feedback [13, 12] are being explored as they can be generated automatically using historical or current log data with reduced engagement of experts. Researchers have investigated varied methods to generate automatic feedback (For example, hint and feedback generation from historical data [5], next step hints from current code log [9] etc.) and also explored the quality and impact of feedback on students' perspective to learn [15, 12, 6].

While prior work has evaluated the quality of automated feedback, or its impact on students' performance or learning, not much is known about the impact of this feedback on students' programming behaviour when it *fails* to provide reliable feedback. This needs more investigation to understand what measures can be taken or what support should be provided to students to mitigate adverse effects of data-driven automated feedback. Our prior study showed that data-driven positive feedback increased students' engagement with the programming task, and improved their programming performance [5]. In this paper we present case studies of specific instances where our data-driven positive feedback system helped students to complete a programming assignment. Since, the feedback is given based on detection of objectives extracted from previous students' correct solutions and does not have a way to adapt to new behaviour, it is not always perfect. Thus, we also explore instances where the system either failed to confirm students' correct steps, or provided misleading feedback and investigated students' response to such events in terms of their programming behaviour and time spent on the task.

2. RELATED WORKS

Several block-based programming environments were designed to reduce the difficulties students face while learning a new programming language in various ways. For example, Alice [2], and Snap [3] provide drag-and-drop coding, and immediate visual code execution. Research has shown that these programming environments are more engaging in terms of reduced idle time while solving a programming problem [8] and can produce positive learning outcomes in terms of grades [2] and the number of goals completed in a fixed amount of time [8].

Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2734/paper5.pdf

To provide novice students with individualized tutoring support, researchers have integrated intelligent features into block-based programming environments. These intelligent features dynamically adapt teaching support to mitigate personalized needs [7]. For example, iSnap [10] is an extension of Snap that provides on-demand hints generated from students' code logs using the Source Check Algorithm [11]. Gusukuma el al. [4] integrated automatic feedback based on learners' mistakes and underlying misconceptions into BlockPy [1], and showed that it significantly improved students' performance. Such data-driven approaches are being integrated to provide more automated adaptive tutoring support in novice programming environments. For example, iSnap showcased the first attempt to integrate data-driven support into a block-based programming environment. Zhi et al. [13] proposed a method of generating example-based feedback from historical data for iSnap. They extracted correct solution features from previous students' code and used those features to remove extraneous codes from current student code and produced pairs of example solutions that were provided on an on-demand basis.

The impact and effectiveness of tutoring supports and intelligent features integrated into novice programming environments have been explored by researchers from various perspectives. Zhi et al. [15] demonstrated the adoption of worked examples in a novice programming environment and found out that worked examples helped students to complete more tasks within a fixed period of time, but not significantly more. Price et al. [12] explored the impact of the quality of contextual hints generated from students' current code on students' help seeking behaviour. They found out that students who usually used hints at least once performed as good as students who usually do not perform poorly and also the quality of the first few hints is positively associated with future hint use and correlates to hint abuse. Marwan et al. [6] evaluated the impact of automated programming hints on students' performance and learning, and argued that automated hints improved learning on subsequent isomorphic tasks when accompanied with self-explanation prompts.

In this paper, we adopted a case study based approach to explore the positive impact of data-driven programming feedback when generated accurately in a block-based novice programming environment, and the negative impacts that can occur when the system fails, to shed light on the influence of such feedback on novice students' programming behaviour.

3. SYSTEM DESIGN

3.1 The Novice Programming Environment

We built the data-driven positive feedback system (DDPF) in iSnap [10], a block-based intelligent novice programming environment. This environment provides students with on-demand hints, and can also log all students' edits while programming (e.g. adding or deleting a block) as a *code trace*. This logging feature allows researchers or instructors to replay all students' edits in the programming environment, and detect the time for each edit as well.

3.2 Data-Driven Positive Feedback (DDPF) System

We built a system to provide positive feedback while students program a specific exercise in the block-based programming environment, using a data-driven feature detector algorithm, described in [14]. This algorithm detects features, i.e. sequence of code blocks that reflect properties of correct solutions from previous students' data. We used the 7 features [Table 1 Column 2] extracted using this algorithm on data from one programming exercise solved in the environment as follows : the system converted snapshots of each edit a student made into an abstract syntax tree (AST) to detect completed features. The system generates a sequence of 0s and 1s called feature state (e.g. 1100000, where the first two 1s indicate presence of the first two features, and 0s indicate absence of rest of the features) for each student's snapshots.

3.3 Positive Feedback Interface

We designed an interface, based on our prior work [5], to provide positive feedback using the DDPF system mentioned above. This interface includes a progress panel that displays a set of four objectives students need to complete to finish the programming task. Two experts in block-based programming converted the 7 data-driven features into four objectives with a meaningful description for each to be displayed in the progress panel. While a student is programming, after each edit, the DDPF system detects the feature state of the current student snapshot, and updates the corresponding objectives in the progress panel accordingly. Initially, all the objectives in the progress panel are deactivated. Once the system detects the presence of a feature, the color of its corresponding objective changes to green, but if it detects the absence of a feature that was present before (i.e. a broken feature), its corresponding objective turns red.

4. PROCEDURE

We deployed our system in an introductory computing course for non majors in Spring 2020 in a class of 27 students, which took place in a public research university in the United States. In this course, students used iSnap (Section 3.1) to solve their in-class programming assignments and homeworks. We integrated our DDPF system into the programming environment for one homework called *Squiral*, described in Section 4.1. The data we collected consists of code snapshots for every edit in student code with corresponding timestamps. We also logged all the objective feature states of all students' code snapshots with the time when every objective was completed or broken. Afterwards, we manually checked the sequential code snapshots for each student and documented the following:

Early, Late, Incorrect and Just-In-Time Objective Detection: We investigated the code snapshots for each student, and filtered out these snapshots where the system detected the completion of an objective. Two researchers evaluated whether each detection was *early* (i.e. the objective was detected before the student completed the objective), or *late* (i.e. the student completed the objective earlier than when it was detected by the system), *incorrect* (i.e. an objective was detected that was never completed), or *just-in-time* (i.e. the objective was detected at the *step* when the student just completed it).

Agreement between Researchers and Automatic Objective Detection: To measure the agreement between researchers and data-driven objective detection, we marked each step of a student's code with: true positive `TP`, where both the researchers and the system detect the completion of an objective at the same step, or true negative `TN`, where

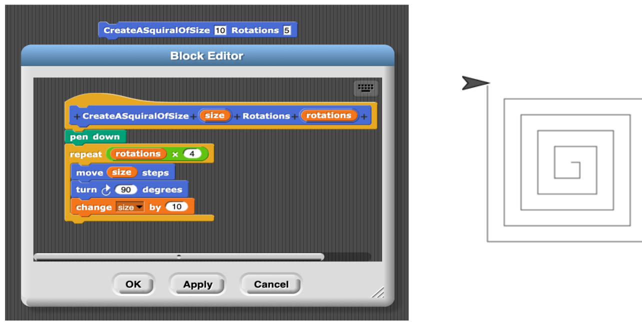

Figure 1: a) A sample expert solution to solve the Squiral assignment; b) Expected output

both the researchers and the system detect that an objective was broken at the same step, or false positive `FP`, where the system detects the completion of an objective; however it was not detected by the researchers, or false negative `FN`, where the system detected an objective as broken at a step where the researchers detect no broken objective.

Idle and Active Time : We measured the total active and idle time spent by each student on the system while solving the programming homework. We also measured, for each student, the active and idle time spent before each objective was detected by the system and the total active and idle time spent before the last change to any objective was detected by the system. We chose a time gap of greater than 3 minutes [75 th percentile based on the frequency distribution of different amount of time gaps] to be considered as idle time. Also, a time gap of greater than 10 minutes [95th percentile] was considered to be the start of a new session and thus, was not added towards either active time or idle time.

4.1 The Squiral Assignment

The Squiral assignment is a programming homework that asks students to create a procedure to draw a spiraling square-like shape. One possible solution and its corresponding output is depicted in Figure 1. Using correct student solutions collected from prior semesters, four objectives were identified [described in section 3.3] and were provided to the students as sub-goals to achieve while solving the problem. The specific features required to complete each objective are shown in Table 1.

4.2 Research Questions

The goal of this study is to explore the impact of automated data-driven adaptive feedback on students' programming behavior. To achieve the goal, we aimed to answer the following research questions:

RQ1: How it Helps.
How did data-driven feedback help students complete the assignment?

RQ2: What happens when it Goes Wrong.
How did the objective detectors impact student behavior, especially with regard to differences between researcher and algorithmic objective completion detection?

RQ3: How Students Respond.
How did data-driven feedback impact students' active and idle time while solving a programming problem?

Table 1: Sample requirements to complete each objective

Objective Number and Label	Required Features for Completion
1: Make a Squiral custom block and use it in your code [similar to creating a function and using it in the program]	- Create and use custom block
2: The Squiral custom block rotates the correct number of times	A nested loop- repeat y * z Or repeat y repeat z y = rotation count z = 4
3: The length of each side of the Squiral is based on a variable	Within loop : move x steps x = length of a side
4: The length of the Squiral increases with each side	- pen down [outside loop] - Within loop : 1. move x steps 2. turn 90 degrees 3. change x by some_value

5. RQ1: HOW IT HELPS

When we observed each student's solution at the end of their attempt, we found out that 25 out of 27 students who were provided feedback had working solutions of the Squiral assignment at the end of their attempt, although, according to researchers, their solutions were not always perfect from a logical perspective [For example, using 'size x size' in the nested loop instead of 'rotations x 4', where 'size' = 10 and 'rotations' = 25, will produce the same Squiral. But, it is not logically correct, since the purpose of the nested loop is to draw 4 sides of the Squiral in each rotation.]. Among the 27 students, two students (Jade and Lime) attempted to solve Squiral using the system both with and without feedback. To get specific insight on how the system helped the students to reach a correct solution, we examined code logs of these two students who each attempted to solve Squiral without Data-Driven Feedback, and failed to complete the assignment. Later, those two students each attempted to solve Squiral with Data-Driven Feedback and succeeded. In this section, we present the case studies of these two students to demonstrate how the feedback system helped the students to fill the gaps in their code and led them to working solutions.

5.1 Case Study Lime

Student, Lime, without Data-Driven Positive Feedback: Student Lime, when attempting to solve Squiral without any hints or feedback [Figure 2] given, used a custom block with a parameter. The student used 'move' and 'turn' statements within a loop in the custom block. However, there were three gaps in the code that the student could not figure out. First, a nested loop was required to iterate for 'rotation count x 4' times. Second, the move statement used 'length x 2' as its parameter whereas only 'length' would be sufficient. Finally, the variable used in the 'move' statement

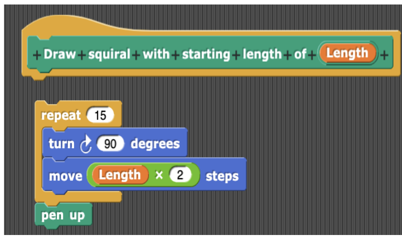

Figure 2: Student Lime's Solution when no Feedback was given

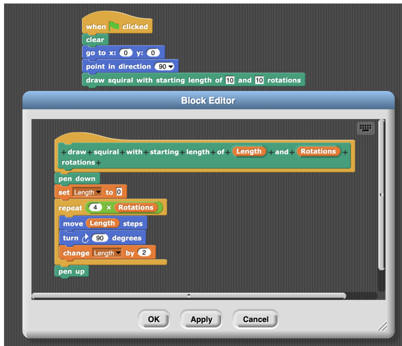

Figure 3: Student Lime's Solution when Feedback was Given

must be incremented at each iteration. The student spent 18 minutes and 18 seconds before giving up, being unable to figure out these issues.

Student Lime with Data-Driven Positive Feedback: Later, student Lime attempted the homework again after receiving notice that the DDPF system was made available. When Lime received feedback, they figured out the 3 issues and reached a correct solution [Figure 3]. The second objective suggests that there is a correct number of rotations that is needed to be used within the custom block. With this feedback, Lime used '4 x Rotations' in the 'repeat' block instead of using '15' and completed the second objective. The third objective suggests the use of a variable in the 'move' statement. Lime used an initialized variable 'length' in the 'move' statement instead of 'length x 2' and the objective was marked green. Finally, Lime incremented 'length' within the loop and all objectives were completed and they reached a correct solution. With data-driven adaptive feedback, Lime spent 29 minutes 51 seconds before reaching the correct solution. Recall that Lime gave up with an incorrect solution after around 18 minutes when no feedback was given.

5.2 Case Study Jade

Student Jade without Data-Driven Positive Feedback: Student Jade initially attempted to solve Squiral without data-driven positive feedback, spent 16 minutes and 55 seconds before giving up with an incorrect solution. Jade's code [Figure 4] contains 'repeat', 'move', and 'turn' statements on the stage. Jade created a custom block and only used the block to initialize a variable, 'length', that was also a parameter to the block. The components to complete the objectives were partially there in Jade's code but it suffered

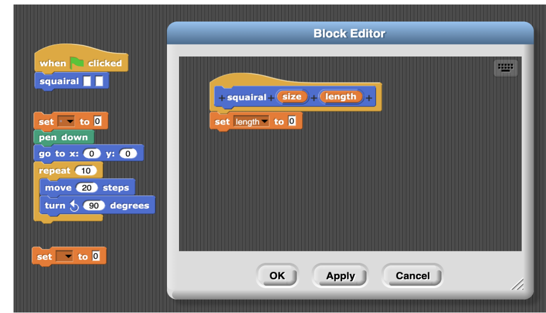

Figure 4: Student Jade's Solution without Feedback

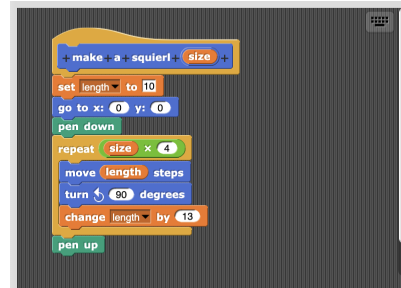

Figure 5: Student Jade's Solution when Feedback was Given

from organizational issues. Also, Jade couldn't figure out that the 'move' statement should use a variable instead of a constant and the same variable needs to be incremented at each iteration. The number of repetitions in the repeat block was also incorrect.

Student Jade with Data-Driven Positive Feedback: Like Lime, Jade attempted the homework again when the DDPF system was provided. When given feedback, Jade first created a custom block and used it on the stage which got the first objective marked green. The second objective hints on using a loop that repeats for correct number of rotations within the custom block. This time Jade implemented the loop within the block and got the second objective correct. Within the loop, Jade used 'move', 'turn', and 'change' statements and reached the correct solution [Figure 5] with all objectives marked green. With adaptive feedback, it took Jade 14 minutes 29 seconds to reach a correct solution. Whereas without feedback, Jade gave up with an incorrect solution after spending over 16 minutes on the problem.

5.3 Findings

The 2 case studies of Lime and Jade presented in this section demonstrated that the feedback system was able to help students in filling up the gaps in their code to reach a correct solution. In one case, this achievement came at the cost of a higher active time and in the other case the student reached a correct solution in less time when feedback was provided. Also, we observed most of the students had a working solution [capable of drawing a Squiral] at the end of their attempts, although not logically 100% correct according to researchers. Moreover, almost all (25 out of 27) the students explored syntactic constructs required to complete the objectives, (e.g. move, turn, iteration, variables), which

potentially indicates that they closely followed the objectives to accomplish the assignment.

6. RQ2: WHAT HAPPENS WHEN IT GOES WRONG

To answer RQ2, we manually walked through the sequential code snapshots of each student when they attempted to solve Squiral and generated case studies where the system went wrong and observed students' problem solving approach. Below we present three case studies demonstrating our system's potential impact on student behavior when the system could not provide reliable feedback. We selected one case where the student completed an objective but the system could not detect it (FN case) and two cases where the system detected completed objectives when the objectives were not completed (according to researchers), which led students to an incorrect solution or made them stop early.

6.1 Case Study Azure : FN Cases Causing Students to Work More than Necessary

Student Azure started solving Squiral by creating a custom block and got the first objective correct. Azure used 2 parameters, 'size' and 'length', to denote the number of rotations and length of the first side of the innermost loop. They created a loop with a 'repeat' block with the correct number of rotations ('size x 4') and got the second objective correct. As Azure used the 'length' parameter in the 'move' statement within the loop, they got the third objective correct. Then Azure added a 'turn' statement and incremented the 'length' variable. At this point, the fourth objective was completed, according to researchers. However, the objective was undetected by the system [FN case], because Azure used a 'turn' statement that was different from those used in the previous students' solutions [that were used to extract and detect the objectives]. According to researchers, Azure's solution was 100% correct at this point [Figure 6]. It took this student only 2 minutes 24 seconds to reach the correct solution.

However, the fourth objective was not detected by the system. Azure kept working on their code. Azure made several changes to their code which led them to an incorrect solution. Finally, Azure ended up submitting a solution that was also 100% correct according to researchers, but was slightly different from their initial solution. In the submitted solution, Azure removed the 'length' variable from the parameter list of the custom block. The fourth objective was still undetected. While doing these changes, the student spent 12 minutes 5 seconds more in the system which is almost 5 times the amount of time the student spent to get a correct solution in the first place.

We observed a similar situation in case of student Blue who worked for a total of 1 hour 43 minutes 16 seconds. Blue reached a correct solution at 1 hour 7 minutes 11 seconds according to researchers. But one objective (fourth objective) was undetected. Blue kept working for another 36 minutes 5 sec (almost 50% of the time taken to reach the correct solution at first attempt).

6.2 Case Study Cyan : FP Cases Leading Students to an Incorrect solution

Student Cyan created a custom block and used it on the stage and got the first objective correct. Cyan used two parameters in the custom block and used one of them in a 'move' statement within a nested 'repeat' block that got him the second and third objective correct. However, Cyan implemented another nested loop and added a 'change' statement within that loop in the stage instead of adding them to the custom block. The system detected the objective and marked the fourth objective green. At this point, the student Cyan had all objectives correct [FP cases] but the code [Figure 7] was unable to draw a Squiral.

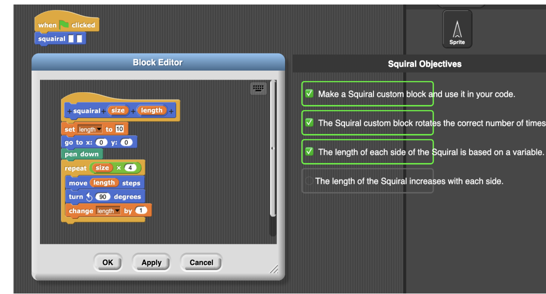

Figure 6: Correct Solution Initially Implemented by Azure

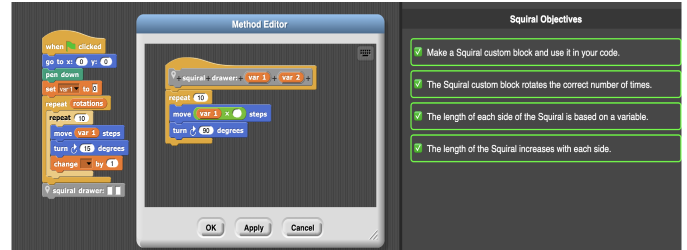

Figure 7: Incorrect Solution Initially Implemented by Student Cyan

Later, Cyan removed the 'change' statement from the stage that caused the fourth objective to be broken. However, removing the custom block from the stage was causing other objectives to be broken. Cyan then moved the 'change' statement to the custom block and corrected the rotation count in the 'repeat' statement. At this point, the solution was correct and similar to Figure 1a. The incorrectly detected objectives led Cyan to a non-working solution. In this case, Cyan had to ignore the detectors and do extra work to reach a correct solution.

We observed a similar situation in the case of three other students. One of them got four objectives correct and the code was able to draw a Squiral, but according to researchers the code had one programmatic problem. The code was drawing three sides of the Squiral using an inner loop and one side manually. The problem detected by researchers remained undetected in the system and the student ended up submitting a partially correct solution. For the other two students, four objectives got detected when there were syntactical problems present in their programs, as for student Cyan. The students realized that completing the objectives did not necessarily mean that their code could draw a Squiral. Although FP cases led the students to incomplete code with four checked objectives, incorrect output eventually compelled each student to modify their codes and to reach a 100% correct solution at the end.

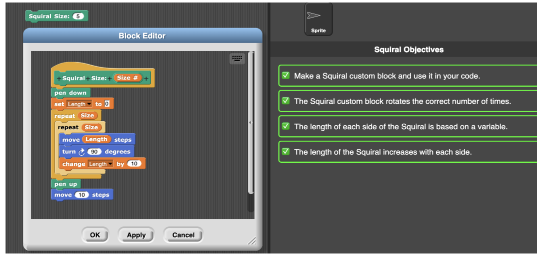

Figure 8: Solution Submitted by Student Indigo

6.3 Case Study Indigo : FP Cases Causing Students to Stop Early at a Partially Correct Solution

We found 6 additional cases where the students got 4 objectives correct, but the code was *partially correct* (FP Case) according to researchers. However, their programs were able to draw a Squiral as required. In such cases, the students finished the attempt early and submitted the partially correct solution. We present the case study of Student Indigo here.

Student Indigo's solution [Figure 8] had objectives 1, 3, and 4 correctly completed according to researchers and the objectives were detected by the objective detection system as well. Indigo created a custom block and used it in the stage [required to complete objective 1]. They used 'pen down' and added 'move', 'turn', and 'change' statements accordingly [required to complete objective 4] within a nested loop implemented with two 'repeat' statements. In the 'move' statement, Indigo used an initialized variable, 'Length' [required to complete objective 3], and incremented the value of 'Length' by 10 at each iteration. However, the rotation count used in the nested loop was wrong. One of the 'repeat' statements should have the count of rotations and the other should have a constant 4, indicating the 4 sides of the square drawn at each rotation. The objective detection system detected the use of the nested loop and marked objective 2 green [FP case]. The code was able to draw a Squiral. However, the implementation was not completely correct. But, once the student Indigo got 4 objectives correct, they submitted their solution.

6.4 Findings

We observed cases where students reached a correct solution, but could not recognize the correct solution because their completed objectives were not detected by the system. They continued working on the assignment for a longer time than was necessary. We also observed cases when students got 4 objectives correct but their solution was not even working, i.e. their code was not able to draw a Squiral because of organizational or syntactical problems. Only in these cases, the students realized completing the objectives is not enough. Thus, they modified the code overriding the objective detectors to reach a working solution. Our third case study showed that students who got 4 objectives correct with a working code submitted their solutions even if their code was not fully correct according to researchers. In these cases, all the students relied on objective detection and submitted partially-correct solutions. These case studies potentially indicate students' high reliance on objective detection feedback, since they did not seem to question the feedback system as long as they had a working program that drew the correct shape. However, students did not rely on their own skill to determine when their code was correct. Furthermore, students sometimes even ignored the produced output that showed they had a working solution, in the cases when objectives were not detected.

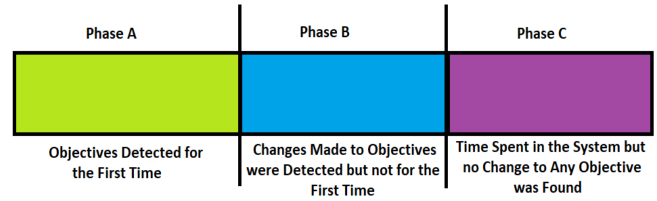

Figure 9: Phases in a Student's Attempt to Solve Squiral

Table 2: Active and Idle Time range before Objectives got detected for the first time

Objective	Active Time (min.)	Idle Time(min.)
1	~0-12.45	0-9.27
2	0.1-84.5	0-10.1
3	0.4-17.9	0-13.25
4	0.5-18.2	0-19

7. RQ3: HOW STUDENTS RESPOND

To answer RQ3, we investigated how the correctness of objective detection at early phases of a problem solving attempt impacted later phases of the attempt. We divided the total time each student spent on the system into three phases [Figure 9] - a) Phase A : when objectives were detected for the first time; b) Phase B : changes in previously detected objectives were detected. c) Phase C : students spent time in the system but no change in the objectives were detected. For example, a student got objectives 1, 3, and 4 marked green within 20 minutes of starting the attempt [Phase A]. Then the student continued working for another 10 minutes [Phase B] but did not see a new objective [objective 2] go green. However, the previously detected objectives were broken and corrected several times. Finally, the student spent another 5 minutes [Phase C] when no change in any of the objectives were detected. We tried to relate correct, incorrect, early, and late detection ratios in phase A with the active and idle time spent in phases A, B, and C to understand if correct, incorrect, early, or late detection regulates the time or effort students put on the assignment.

Active and Idle Time Observed in Phase A: In this phase, objectives got detected for the first time by the system. Students were observed to take wide ranges of time before an objective was detected [Table 2]. We plotted average active and idle time against correct objective detection ratios and observed that students with higher correct objective detection ratio have shorter phase A in terms of active time [Figure 10a]. For 1 of the students, this phase did not occur because an objective was never detected in the student's code. In this phase, only a few cases were found when the students had idle time. 16 out of 27 students did not

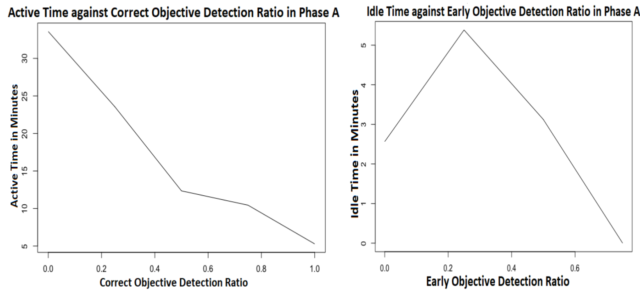

Figure 10: a) Active Time in Phase A against Correct Objective Detection Ratio; b) Idle Time in Phase A against Early Objective Detection Ratio

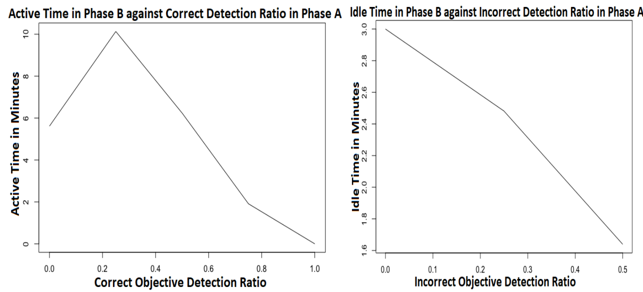

Figure 11: a) Active Time in Phase B against Correct Objective Detection Ratio in Phase A; b) Idle Time in Phase B against Early Objective Detection Ratio in Phase A

have any idle time at all. 7 students had idle time ranging from 3-5 minutes. The rest of the 4 students had idle time ranging from 10 - 24 minutes. We observed that a higher early detection rate (over 25%) has a decreasing trend in average idle time [Figure 10b]. This may potentially indicate that positive feedback can be motivating to students, even if it is provided early.

Active and Idle Time Observed in Phase B: For 11 students, phase B did not occur at all, due to either the fact that an objective was never detected, or the students submitted their code after all of the objectives were detected for the first time in phase A. 10 students spent >0 - <10 minutes, and 6 students spent >10 - 35 minutes in phase B. 4 of the 6 students who spent a higher amount of time in this phase B had a high early detection ratio at phase A (50-75%), and 1 student had a high incorrect detection ratio (50%). These students did not have correct solutions, even if some or all of the objectives got detected in phase A. In this phase B, 21 out of 27 students had no idle time. 6 students had idle time ranging from 3 to 24 minutes. As we plotted average active and idle time in phase B against the correct and incorrect objective detection ratio in phase A, we observed a higher correct detection rate (>25%) in phase A seemed to decrease the active time spent in phase B [Figure 11a]. This means that correct detections helped students complete their programs more quickly. Correct objective detections in phase A pushed the students towards the end of their attempt. However, incorrect objective detection in phase A decreased idle time in phase B and caused the students to continue actively working [Figure 11b].

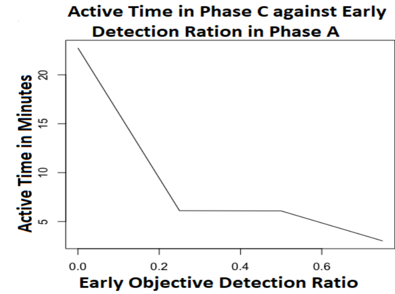

Figure 12: Active Time in Phase C against Early Objective Detection Ratio in Phase A

Active and Idle Time Observed in Phase C: This phase C indicates the time when no change was detected in any of the objectives. In this phase C, one of the following scenarios occur : 1) Most of the objectives were detected in earlier phases; the student had a working solution and was making minor modifications without impacting the objectives; or 2) One, more than one, or all of the objectives were undetected and the student was working on the assignment but submitted the attempt without another objective being detected. We observed that when the first scenario occurs, students spent only a few minutes in phase C and submitted their code. 18 out of 27 students spent only 0.1-8 minutes in the system after scenario 1 occurred: even if the objective detection was wrong, the student relied on the system, and submitted their code. We observed a higher early detection ratio in phase A led to decreased average active time in phase C [Figure 7]. The remaining 9 students spent 12-56 minutes in this phase. Scenario 2 played out for 7 of these 9 students, who all submitted the program with some incomplete objectives.

7.1 Findings

The results of our analysis showed that the active and idle time spent in Phase B and Phase C are associated with the quality of detection in Phase A in some cases. We observed that correct objective detection in Phase A that led to a working solution pushed students to finish their attempt, making Phases B and C shorter. Whereas, incorrect objective detection in Phase A that led to a non-working solution decreased the idle time observed in Phase B and students worked more. However, the active time in such cases varied from student to student, and depended on the extent to which objective detection went wrong. Our case studies presented in Section 6 demonstrated indication of students' significant reliance on the feedback. This reliance on the system interacted differently depending on whether the feedback was correct or incorrect, and whether or not the student code output appeared correct, and these differences are reflected in students' response or effort in terms of active and idle time.

8. DISCUSSION

Our case studies and analysis demonstrated that, although our feedback system could guide students to complete a programming assignment, it could mislead students to do extra work, submit a partial correct solution, or to end up at a non-working solution in the event it provides inaccurate

feedback. However, the visual feedback of objectives going green [even for an incorrect or too-early detection] reduced idle time, giving us an indication that the feedback was motivating for students. All these observed impacts could be the result of students' high reliance on the feedback system. To prevent the negative impacts generated in the event of incorrect detections, mechanisms to prevent incorrect detections must be explored. We observed our system to fail in the event of new student behaviours. Since we built interventions that use data from prior students and new students behave in new ways, the system has not had time to learn and adapt based on their behavior. Thus, any system like ours must have an iterative process for integrating new behaviors that may arise from diversity in students and instructors.

9. CONCLUSION AND FUTURE WORK

This paper presents case studies to provide important insights into the impacts of positive feedback on novice programmers from multiple perspectives. We present case studies that shed light on how the feedback system helped two students to complete a programming task who failed to complete the task on their first attempts without feedback. While these scenarios highlighted the usefulness of positive feedback, our case studies surrounding the event when the system could not provide accurate feedback provided insights on what impact feedback failures can have on students' responses. These insights can be highly useful to decide on measures to mitigate an adverse impact or to formulate adaptations to handle unexpected behaviors. This may involve expressing a confidence level for detectors, or inviting students to self-explain how and why their solutions are correct. The primary contributions of this work are: 1) Case Studies demonstrating how the positive feedback system can induce a working solutions for a programming task; 2) Case studies and code trace-based analyses that gave important insights on how a data-driven positive feedback system impacts students' behaviour when the system goes wrong. Our results show interesting relationships between correctness of the provided feedback and the time students spent on the task or in the system. In our future work, we plan to explore these impacts in larger controlled studies and on other programming tasks, and we also plan to explore how we can adapt our system to balance students' understanding of their own code with reliance on feedback, to promote learning.

10. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grants 1623470.

11. REFERENCES

[1] A. C. Bart, J. Tibau, E. Tilevich, C. A. Shaffer, and D. Kafura. Blockpy: An open access data-science environment for introductory programmers. *Computer*, 50(5):18–26, 2017.

[2] W. Dann, D. Cosgrove, D. Slater, D. Culyba, and S. Cooper. Mediated transfer: Alice 3 to java. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 141–146, 2012.

[3] D. Garcia, B. Harvey, and T. Barnes. The beauty and joy of computing. *ACM Inroads*, 6(4):71–79, 2015.

[4] L. Gusukuma, A. C. Bart, D. Kafura, and J. Ernst. Misconception-Driven Feedback: Results from an Experimental Study. *Proceedings of the 2018 ACM Conference on International Computing Education Research - ICER '18*, (1):160–168, 2018.

[5] S. Marwan, G. Gao, S. Fisk, T. W. Price, and T. Barnes. Adaptive immediate feedback can improve novice programming engagement and intention to persist in computer science. In *Proceedings of the International Computing Education Research Conference (forthcoming)*, 2020.

[6] S. Marwan, J. Jay Williams, and T. Price. An evaluation of the impact of automated programming hints on performance and learning. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, pages 61–70, 2019.

[7] T. Murray. Authoring intelligent tutoring systems: An analysis of the state of the art. 1999.

[8] T. W. Price and T. Barnes. Comparing textual and block interfaces in a novice programming environment. In *Proceedings of the eleventh annual international conference on international computing education research*, pages 91–99, 2015.

[9] T. W. Price, Y. Dong, and T. Barnes. Generating data-driven hints for open-ended programming. *International Educational Data Mining Society*, 2016.

[10] T. W. Price, Y. Dong, and D. Lipovac. isnap: towards intelligent tutoring in novice programming environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 483–488, 2017.

[11] T. W. Price, R. Zhi, and T. Barnes. Evaluation of a Data-driven Feedback Algorithm for Open-ended Programming. In *Proceedings of the International Conference on Educational Data Mining*, 2017.

[12] T. W. Price, R. Zhi, and T. Barnes. Hint generation under uncertainty: The effect of hint quality on help-seeking behavior. In *International Conference on Artificial Intelligence in Education*, pages 311–322. Springer, 2017.

[13] R. Zhi, S. Marwan, Y. Dong, N. Lytle, T. W. Price, and T. Barnes. Toward data-driven example feedback for novice programming.

[14] R. Zhi, T. W. Price, N. Lytle, Y. Dong, and T. Barnes. Reducing the state space of programming problems through data-driven feature detection. In *EDM Workshop*, 2018.

[15] R. Zhi, T. W. Price, S. Marwan, A. Milliken, T. Barnes, and M. Chi. Exploring the impact of worked examples in a novice programming environment. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 98–104, 2019.