ABSTRACT

MARWAN, SAMIHA ABDELRAHMAN MOHAMMED. Investigating Best Practices in the Design of Automated Hints and Formative Feedback to Improve Students' Cognitive and Affective Outcomes. (Under the direction of Thomas W. Price).

Timely support is essential for students to learn and improve their performance. However, in large programming classrooms, it is hard for instructors to provide real-time support (such as hints) for every student. While researchers have put tremendous effort into developing algorithms to generate automated programming support, few controlled studies have directly evaluated its impact on students' performance, learning and affective outcomes. Additionally, while some studies show that automated support can improve students' learning, it is unclear what specific *design choices* make them more or less effective. Furthermore, few, if any, prior studies have investigated how well these results can be *replicated* in multiple learning contexts. Inspired by educational theories and effective human feedback, my dissertation has the goal of designing and evaluating different design choices of automated support, specifically next-step hints and formative feedback, to improve students' cognitive and affective outcomes in programming classrooms.

In this thesis I present five studies that attempt to overcome limitations in existing forms of automated support to improve students' outcomes, specifically hints and formative feedback. Hints may be ineffective when they: 1) are hard to interpret, 2) fail to engage students to reason critically about the hint, and 3) fail to guide students to effectively seek help. In Study 1, I addressed the first two challenges by evaluating the impact of adding textual explanations to hints (i.e. explaining what the hint was suggesting), as well as adding self-explanation prompts to hints (i.e. asking students to reflect on how to use the hint). I found that hints with these two design features together increased learners' learning as evidenced by the increase in their performance on future isomorphic programming tasks (without hints available). In Study 2, I tackled the third challenge in two phases. First, I created a preliminary taxonomy of unproductive help-seeking behaviors during programming. Then, using this taxonomy, I designed and evaluated a novel user interface for requesting hints that subtly encourages students to seek help with the right frequency, estimated with a data-driven algorithm. This led to an improvement in students' help-seeking behavior. In Study 3, I replicated my first two studies in an authentic classroom setting, across several weeks, with a different population, to investigate the consistency and generalizability of my results. I found that hints with textual explanations and self-explanation prompts improved students' programming performance, and increased students' programming efficiency in homework tasks, but the effectiveness of hints was not uniform across problems.

Formative feedback is effective when it is immediate, specific, corrective and positive. Learning theories, and empirical human tutoring studies show that such elements of feedback can improve both students' cognitive and affective outcomes. While many automated feedback systems have some of these feedback elements, few have them all (such as providing only corrective feedback but not encouraging positive feedback), and those were only evaluated on a small set of short programming tasks. In Study 4, I tackled this gap in research by developing an adaptive immediate feedback (AIF) system, using expert-authored rules, that provides students with immediate positive and corrective feedback on their progress while programming. I found that the AIF system improves students' performance, engagement in programming, and intentions to persist in computer science. Lastly, in Study 5 I developed a hybrid data-driven algorithm to generate feedback that can be easily scaled across different programming tasks, with high accuracy and low expert effort. I then used this algorithm to design an improved version of the AIF system (i.e. AIF 3.0), with a more granular feedback level. In Study 5, I deployed and evaluated the AIF 3.0 system in an authentic CS0 classroom study over several weeks. I found that the AIF 3.0 system improved students' performance and the proportion of students who fully completed the programming tasks, indicating increased persistence. Studies 1, 2, and 4 are laboratory studies, while Studies 3 and 5 are classroom studies, all conducted with iSnap, a block-based programming environment.

The contributions of this thesis include: 1) the discovery of effective design choices for automated hints, 2) the design of adaptive immediate feedback systems, leveraging an expert-authored and hybrid data-driven models, 3) an empirical evaluation of the impact of automated hints and formative feedback on learners' cognitive, and affective outcomes, and lastly 4) replication evaluations of hints and feedback in authentic classroom settings, suggesting consistent effects across different populations and learning contexts. These contributions inform researchers' knowledge of challenges of automated support designs using either data-driven or expert-authored models, as well as challenges in classroom studies for open-ended programming tasks; and how they can affect students' outcomes, which overall can guide future research directions in computing education and human-computer interaction areas.

Investigating Best Practices in the Design of Automated Hints and Formative Feedback to Improve Students' Cognitive and Affective Outcomes

by
Samiha Abdelrahman Mohammed Marwan

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2021

APPROVED BY:

Tiffany Barnes	James Lester
Eric Wiebe	Thomas W. Price Chair of Advisory Committee

BIOGRAPHY

Samiha's full name is "Samiha Abdelrahman Mohammed Marwan". Samiha was born in Cairo, the capital city of Egypt, and her first language is Arabic. Samiha is the daughter of Kamilia Ragheb, the woman who loved teaching and invested all her life to educate not only her kids, but any person who cannot afford to be well educated, without expecting anything in return. Like her mom, Samiha grew up to be very passionate about teaching and dreaming to engage people with science and make the world a better place for everyone.

Samiha joined the faculty of Computer and Information Sciences (CIS) in Ain-shams University, one of the most reputable public universities in Egypt. After being admitted into CIS, Samiha joined the Bioinformatics department which was brand-new then in 2007. Although it was a lot of hard work, and her fellow bioinformatics students were fiercely competitive, Samiha learned a crucial fact: *It's not impossible to succeed, it's remaining successful and keeping the drive that's so much harder.* Samiha was granted her bachelor's degree with excellence and honors and ranked second overall in her class. Samiha then earned her MSc degree in 2016, from the British University in Egypt, one of the top private universities in Egypt. Samiha's research during her MSc degree was in DNA steganography, under the supervision of Dr. Ahmed Shawish and Prof. Khaled Nagaty.

In 2017, Samiha was awarded a PhD scholarship at North Carolina State University. In her first year, Samiha started doing research in cryptography as this area was related to her MSc research. At the end of the first year, she started to feel more into teaching students programming and helping them program better, where she then decided to spend her Summer doing voluntary research in the Game2Learn lab, supervised by Dr. Tiffany Barnes. In Fall 2018, Samiha was offered to be a research assistant under the supervision of Dr. Thomas Price, and from then her passion in the computing education area grew more and more. Samiha's research work focused on designing and developing automated feedback features and conducting research studies to evaluate their impact on students' learning and affective outcomes. In addition to her research work, Samiha volunteered in SPARCS (an outreach program at NCSU), where she assisted in teaching middle-school students some CS topics.

Samiha's research work continues to investigate ways that can make programming easier for students. One day she hopes to contribute in making students use their computational skills to make the world a better place.

ACKNOWLEDGEMENTS

All praise to ALLAH, the most gracious and merciful, for helping me accomplish all this work.

I want to start by thanking my husband Ahmed Elmorli for believing in me and pushing me forward all the time. Ahmed did not only support me emotionally, but also taught me how to be a better coder. I am just so fortunate to have a kindhearted software engineer husband, the father of the most supportive daughter ever; Lily Elmorli.

While she is not here, her prayers are always following me. Mom, I am always grateful to have had a mom like you, and I would have never reached this phase without you raising me to be a useful person who is eager to spread knowledge everywhere. To you and Dad, I acknowledge that this accomplishment is the result of your sacrifices.

I want to express my sincerest thanks to my supervisor Dr. Thomas Price who taught me a lot, and shared all the advice and resources he could to make me a successful researcher. I also want to acknowledge that all my PhD work is the extension to his PhD work, and without any doubt, I wouldn't have completed this work without his guidance and passion about it.

I also want to express my gratitude to Dr. Tiffany Barnes, who gave me the opportunity to collaborate in her research lab, and provided me with hours and hours of mentorship. I appreciate the help of the rest of my committee members as well, Dr. James Lester and Dr. Eric Wiebe for their thoughtful insights on my research, and fruitful suggestions that made my research work better.

All thanks to Dr. Veronica Catete for making me brainstorm ideas with her, and for her emotional support whenever I am sad. Thanks to my PhD friends Rui, Yihuan, Alex, Emma, Yang, and Sasha for reviewing my work and giving me advice on how to make my research studies and paper writing better.

To my brothers, Mohammed and Amr, thank you for being my backbone despite the hundreds of miles between us. You have always been the first place to go whenever I feel I can't keep going any more. You are the reason why I pursued my PhD. I can never be thankful enough!

Last but not least, I want to thank my aunties, my in-laws and all my friends in the US and in Egypt for their prayers and continuous emotional support through all my PhD journey. I am lucky to be surrounded with all these kindhearted caring people.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Programming is a vital skill in many disciplines, as evidenced by the large number of introductory programming courses offered for non majors [Sim19]. However, in 2019, the average failure rate of computer science (CS) courses ranges from 25% to 28% [Sim19; Ben19], and still many CS programs struggle to increase students' retention in the major [Gor20]. This is especially true in early programming stages; as concluded in a 2019-survey of CS failure rates: "It appears that introducing students to computing is still one of computing education's grand challenges [Ben19]." One powerful way to improve students' outcomes is to provide them with automated support, such as hints on what to do next, or formative feedback, that can be positive or corrective, showing students whether their current solution is correct or incorrect [Shu08; Riv17a]. Such automated support in programming can impact student thinking and behavior and has been shown to improve learners' knowledge [Shu08; Thu13; Mar20c]. However, due to the rapidly growing programming classrooms, it is hard for computing instructors to provide real-time support for every student. To address this, there is a growing body of research investigating how to automatically generate high-quality support for students. This automated support can be in the form of compiler messages [Bec16], error flagging [Cor01], or next-step hints [Riv17b; Van06]. However, there are conflicting findings on the impact of automated support on students' cognitive outcomes [Riv17a; Mar19b; Gus18]. For example Gusukuma et al. found that misconception-driven feedback – which used a set of expert-authored rules – improved students' learning [Gus18], while Price, Marwan et al. found that data-driven next-step hints (i.e. a hint suggesting a correct next step from prior correct students data) sometimes improved students' performance [Mar19b] and sometimes had no effect [Pri20], depending on the learning context. The inconsistency of these results is in large part due to the variety of ways that automated support can be generated, which may yield to different outcomes. In addition, little work has investigated the *trade offs* between the impact of different automated support design choices (such as human-authored versus data-driven support, or text-based versus visual support) on students' outcomes and perceptions. We therefore lack an understanding on *what* features can make automated support effective, or how to improve it. As Cohen notes: "feedback is one of the more instructionally powerful and least understood features in instructional design" (p. 33) [Coh85].

1.2 Research Approach

In this thesis, I explore how to effectively design automated support, in particular data-driven next-step hints and formative feedback, and I present evaluations of their impact on student outcomes across different learning contexts. Data-driven next-step hints are a type of automated support generated from historical solutions (whether from students, or experts) that can bring current student code closer to a correct solution, in response to their incorrect or incomplete code. The primary purpose of next-step hints is to allow students progress when they are stuck [Pri17e; Riv17b]. Data-driven next-step hints are particularly powerful since it is scalable, can support students during program construction, and can be easily generated with a variety of approaches in different forms [Pri17e; Riv17a; Mar19b]. However, only a few classroom studies show that they can actually *help*, particularly in terms of students' performance, and learning outcomes. This lack of effect for next-step hints in particular might be for a number of reasons: First, next-step hints are "bottom-out" hints [Pri17e], telling students *what* to do, not *why*, and therefore, students may not fully understand the hint [Mar19a]. Second, these hints do not engage students in sense-making (i.e. reflecting on the hint to understand its meaning), which is an essential component of learning [Koe13]. Third, since these hints are usually on-demand hints (i.e. upon a student's request), students may keep requesting and following them blindly to reach the correct solution, indicating an unproductive help-seeking behavior, which correlates negatively with learning [Ale06]. These reasons clarify how and why next-step hints *may* just improve students' performance (i.e. allowing them to complete the programming tasks), but not learning.

In K-12 and CS0 classrooms novice students *not only* need support when they are stuck, but also need corrective and positive feedback when they are uncertain or do not know whether they are on the right track [Win96]. This uncertainty may lead some students to lose motivation, give up early [Pal05; Gor20], or even submit incorrect code without realizing it, leading to lower course performance, and missed opportunities to debug their errors [Sha20]. Feedback that can address this struggle is called formative feedback; which is the second focus in this thesis. Shute defined formative feedback as a type of task-level feedback that provides *specific, timely* information to a student in response to a particular problem or task, based on the student's current ability [Shu08]. Formative feedback has been shown to improve students' learning and motivation to learn [Shu08; Mar20a; Boy08; Eps02]. In a review on effective formative feedback, Scheeler et al. show that formative feedback is effective when it is specific, corrective, positive and immediate [Sch04]. However, prior work often developed a *subset* of such forms of formative feedback, not all of them, in computer learning environments. This might be because it is easier to develop systems that can recognize errors than recognizing every possible correct step a student can make in open-ended programming tasks [And95; Fos15], and therefore, less feedback systems can provide specific positive feedback proactively similar to human tutors [Fos15; Mit13]. Also, most, if not all, of prior work developed formative feedback on *short* programming tasks, using expert-authored rules [Gus17; Gus18; Mit13]. These feedback systems require extensive expert effort and might be hard to scale for large or more complex programming tasks. In addition, to my knowledge, there is only prior work that evaluated *human tutoring*, not automated, formative feedback, on students' affective outcomes (e.g. self-efficacy and motivation) [Boy08; DE09]. This is important to investigate because prior literature showed that the increase in students' self-efficacy and academic self-esteem (i.e. affective outcomes) can predict students' grades, and other indicators of academic success [Ban77; Dwe86; Sch84]. Lastly and most importantly, few, if any, formative feedback systems have been deployed and evaluated in authentic classroom settings.

Given these limitations in prior literature, in this thesis I focus primarily on developing and evaluating different design choices to improve the effectiveness of next-step hints and automated formative feedback on students' cognitive and affective outcomes. Furthermore, while prior work often evaluates the impact of automated support on one population in one learning context, in this thesis I have also studied the impact of my developed automated support features on different populations, and in different learning contexts. This yields more robust evidence for the potential and limitations of automated hints and formative feedback.

Overall, my dissertation informs the computer education research (CER) community about what design choices can make automated next-step hints effective in real classrooms, and, specifically, how and when data-driven next-step hints improve students' performance and learning. In addition, it informs both the CER and human-computer interaction (HCI) communities on how to design formative feedback that is scalable and can be deployed in real classrooms and what forms of formative feedback that can improve both students' cognitive and affective outcomes. In particular, these are the contributions of my work:

1. Identifying design choices to improve the effectiveness of automated next-step hints.
2. Introducing a taxonomy of students' unproductive help-seeking behavior in programming.
3. Developing two adaptive immediate feedback systems (AIF, and AIF 3.0), one using expert-authored rules, and one using a hybrid data-driven algorithm. Both systems provide formative feedback that is immediate, corrective, positive and specific.
4. Empirically evaluating the impact of hints and AIF feedback on students' outcomes.
5. Evaluating how my findings generalize (i.e. replicated) across different populations, and multiple learning contexts.

1.3 Research Questions

The main goal of my dissertation is to design effective automated support that improves students' cognitive and affective outcomes. To reach my goal, my dissertation seeks to answer these four high-level research questions:

1. **RQ1:** How do specific design choices (i.e. textual explanations, self-explanations, and hint displays) impact automated next-step hints to improve students' performance, learning, and perceptions?

2. **RQ2:** What is the impact of enhanced next-step hints during block-based programming, on students' performance, and learning?

3. **RQ3:** What is the impact of the formative adaptive immediate feedback (AIF) systems on students' affective and cognitive outcomes?

4. **RQ4:** How consistent are the impacts of enhanced next-step hints / formative feedback across different populations and learning contexts?

My first two studies (shown in Chapters 3, and 4) address my first research question (RQ1) and part of RQ4. The third study (as shown in Chapter 5) addresses RQ2 and part of RQ4. The fourth and fifth studies (as shown in Chapters 6, and 7), addresses RQ3 and and part of RQ4. Each of my presented five studies has a subset of research questions that clarifies the outcomes of the study. Table 1.1 shows sub research questions for each study in my dissertation.

Table 1.1: Thesis' Sub-Research Questions.

Study no.	**Research Question**
Study 1	1- What is the effect of hints with and without self-explanation prompts on: a) learners' perspectives on iSnap's helpfulness with and without receiving hints? b) learners' performance? c) learning, as measured by performance on future tasks without hints?
Study 2	1- What specific forms of unproductive help-seeking do novices engage in when programming? 2- How can the design of a help interface improve students' help-seeking behavior?
Study 3	1- In classroom settings, what is the impact of enhanced next-step hints on students' programming performance, and learning? [Replication study] 2- How does problem difficulty and hint quality mediate the effect of enhanced next-step hints on students' outcomes?
Study 4	1- What impact does adaptive immediate positive feedback have on students' performance, and learning? 2- What impact does adaptive immediate positive feedback have on students' perceptions, engagement, and intentions to persist in CS?
Study 5	1- In classroom settings, what is the impact of a data-driven adaptive immediate feedback system on students' (a) performance, (b) rate of task completion, and (c) learning? 2- How is an adaptive immediate feedback system perceived by students?

CHAPTER

2 BACKGROUND & RELATED WORK

The goal of this dissertation is to explore how to improve the effectiveness of automated support, in particular data-driven next-step hints and formative feedback. Next-step hints support students by telling them a next step (e.g. one code edit) that can bring their code closer to the correct solution to get students unstuck during programming [Riv17a; Pri17e; Van06]; while formative feedback is a type of information communicated to the learner that is intended to modify their thinking or behavior to improve their learning, such as corrective and positive feedback [Shu08]. As such, in this section I will present a literature review on automated support generation approaches, with a focus on both expert-authored and data-driven ones and empirical evaluations on each. I will then present literature on effective human feedback features, and how these features were designed in computer-based tutors in different domains, showing how my research will be built on this prior work.

2.1 Automated Feedback Generation Methods

A growing body of work in computing education, and artificial intelligence has developed and evaluated various computer-based tutors that provide automated programming feedback [Pri17e; Cor01; Riv17b; Mit07; Fos15; Har10]. This feedback is developed to support struggling students while solving their assignments, particularly at times when it is hard to receive instructors' assistance. There are various methods to generate this feedback such as expert-authored approaches using constraint or rule-based models [Mit99; Ger16], or program generation techniques from software engineering, such as test-driven synthesis [Per14], or automated program repair [Yi17]. In my dissertation, my work is focused on the most common feedback generation approaches: *expert-authored* approaches and *data-driven* approaches.

2.1.1 Expert-authored Feedback Generation Methods

Expert-authored approaches for feedback generation can include any approach that relies on expert-authored effort: such as expert-authored rules that are used to develop test cases and autograders [Bal18; Mar20a; Wan20], or expert-authored constraints used in constraint-based models [Mit99; Mit02; Mit13; Le09]. For example, constraint-based tutors generate their feedback by diagnosing students' solutions against expert-authored constraints (e.g. SQL tutor in database language [Mit13]). Ask-Elle is another expert model based intelligent tutor that uses a model-tracing technique to provide automated textual hints for students in the Haskell programming language [Ger16]. In Ask-Elle, an expert (e.g. a teacher) defines a set of model solutions and a set of constraints to a given problem and annotates their source code with feedback messages. When a student requests a hint, the Ask-Elle tutor provides them with one of these feedback messages. While these tutors have been proven to generate useful feedback, they often require extensive expert effort to create rules. To address this authoring burden, example-tracing tutoring systems infer tutoring rules based on examples of potential student behaviors. This still requires an author with some domain expertise, but it allows rules to be constructed by non-programmers who have domain expertise [Ale16a]. An expert can create different example solutions to capture different solution strategies; and augment them with hints or feedback. Example-tracing tutors have been developed in multiple non-programming domains like genetics [Cor10], mathematics [Ale09], and applied machine learning, and they have been shown to improve the problem-solving process and student learning [Ale16a].

Despite the accuracy of expert models in providing feedback on test cases or correct features, they require extensive human expert effort in defining models or test cases [Tol20; Mar20b; Gus17]. Furthermore, as Rivers [Riv17a] and Price et al. [Pri17e] argue, it is hard to scale these approaches to new programming tasks, particularly those with open-ended nature[1]. To address this challenge, researchers developed data-driven approaches to generate automated support (such as hints) using previous students' correct solutions, which make these approaches independent of test cases, or expert-model solutions, allowing them to scale to a variety of programming problems. Next, I discuss data-driven next-step hint and formative feedback approaches, though the implications of my work can be applied to different forms of automated support as well.

2.1.2 Data-Driven Support

Data-driven approaches refers to systematically collecting and analyzing various types of educational data, to guide a range of decisions to help improve the success of students [Cus18; Rom20]. Data-driven models largely avoid the need for expert authoring altogether by using prior students' correct solutions, instead of expert rules or instructor solutions, to learn patterns of correct solutions. This enables automated assessment feedback on student code [Fos15]. Next, I focus on data-driven hints and formative feedback generation methods.

Data-driven Hint Generation Methods

Data-driven hint generation systems are inspired by the Hint Factory, a hint-generation algorithm used in a logic tutor [Sta08]. The Hint Factory uses previous solution strategies of students who successfully completed a problem, and generates hints for new students by offering a hint based on the actions of previous students who had similar solution strategy. Data-driven approaches have been successfully applied in programming by providing students with personalized hints automatically, in different contexts, by looking through either previous students' or experts' solutions [Hic14; Riv17b; Laz17; Pri17a].

Hints generated from data-driven approaches are usually in the form of *edit-based, next-step* hints, which suggest a specific edit to the student's code to bring them closer to a correct solution [Riv17c; Har10; Pri17e; Wat12]. Figure 2.1 shows an example of how a next-step hint (referred to as a code hint) is displayed in iSnap, a block-based programming environment [Pri17e]. Data-driven next-step hints are promising since they provide adaptive feedback,

[1]Open-ended programming exercises are those that have multiple correct solutions and strategies.

specific to the student's code without requiring instructors to author hints or test cases manually [Pri17a; Riv17c]. However, because they are not authored by instructors, these hints are generally "bottom-out" hints, limited to suggesting to students *what* to do next, without an explanation of *why* the hint is relevant to students' code [Pri17b; Riv17a].

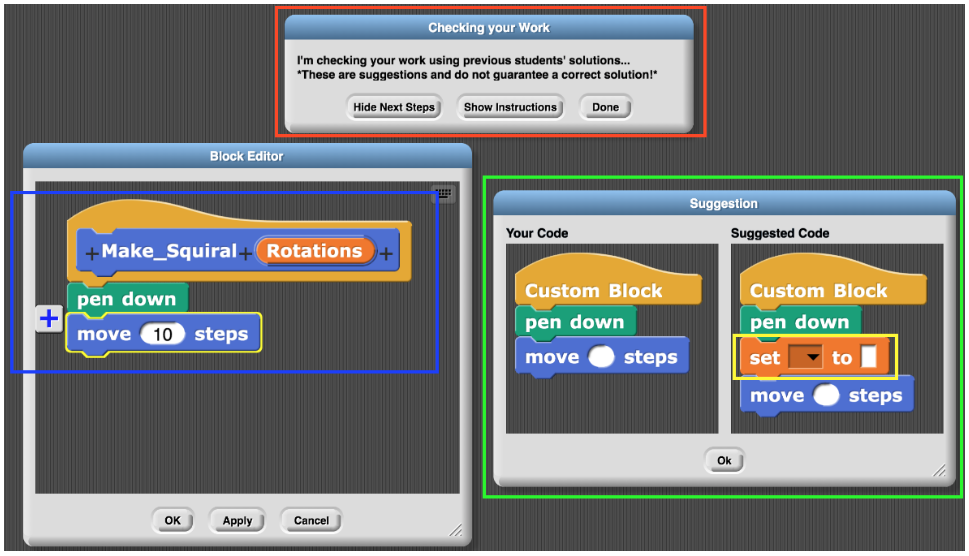


Figure 2.1: A next-step hint (referred to as code hint) in iSnap. The student's code is shown in the bottom-left (blue box), with a suggested hint in the bottom-right (green box), and a dialog that appears when students request a hint in the top (red box).

According to the ACT-R cognitive theory, learning problem solving requires two types of interrelated knowledge: declarative knowledge, which can be represented verbally (knowing *that*, e.g. "what is a variable"), and procedural knowledge, which encodes how to solve a specific problem step (knowing *how*, e.g., "when and how do I declare a variable") [And96]. Principle-based hints that explain a domain concept can therefore be seen as helping students to acquire declarative knowledge and contextualize their procedural knowledge accordingly [Ale13]. In addition, the Knowledge-Learning-Interaction (KLI) framework [Koe13] suggests that hints can prompt this acquisition of verbal (declarative) knowledge by engaging the student in sense-making, or the process of reasoning and constructing self-explanations. This idea is also supported by Aleven and Koedinger who argue that the self-explanation process is an essential component of learning from hints, but that it is unlikely to occur spontaneously [Ale16b]. As such, one of my dissertation goals is to improve the design of next-step hints relying on these theoretical perspectives by including both textual explanations (similar to principle-based hints), and prompts to encourage students to self-explain the hints, to improve students' performance and learning.

Data-driven Formative Feedback Generation Methods

While most data-driven methods have been used to generate fine-grained feedback – such as hints in the hint factory [Sta08], little work has used data-driven methods to generate formative feedback. For example, Diana et al. developed a model that searches for meaningful code chunks in student code to generate data-driven rubric criteria to automatically assess students' code [Dia18]. Diana et al show that a data-driven model can have agreement with that of the experts [Dia18]. In the iList tutor, Fossati et al. used a data-driven model to provide feedback on correct steps, where they assess student code edits as *good*, if it improves the student's probability to reach the correct solution, or *uncertain* if a student spent more time than that taken by prior students at this point [Fos15]. However, data-driven models are dependent on the similarity of the current student's approach to prior student submissions, making it difficult to control the quality of their feedback [Mar20b; Sha20; Tol20; Pri17b]. In a case study paper, Shabrina et al. discuss the practical implications of data-driven feedback on subgoals, showing that the quality of feedback is important, even positive feedback, since inaccurate feedback can cause students to spend more time on a task even after they were done [Sha20]. Because of such challenges and perhaps other reasons such as the inability for individual instructors to augment autograder feedback, few tools have been built to provide formative feedback to students during their programming tasks [LR18].

2.2 Empirical Evaluations of Automated Support Systems

2.2.1 Empirical Evaluations of Data-Driven Next-Step Hints

The majority of evaluations of data-driven next-step hints have focused on technical aspects of the hint generation process, such as how often a system will be able to successfully generate hints (e.g. [Hic14; Wan17]) or whether the generated hints are able to resolve known errors in students' code (e.g. [Gup17; Laz14; Wan17]), or by asking experts to directly rate the quality of the generated hints (e.g. [Pie15; Pri17a; Pri17d; Pri18]). For example, Hartmann et al. [Har10] evaluated their *HelpMeOut* system by deploying it during a programming workshop, and they manually labeled the hints that the system generated as "Helpful" or

"Not Helpful." They found that 47% of hint requests returned helpful suggestions. However, only little work conducted studies to closely investigate how these data-driven next-step hints impact learners' experiences, perceptions, performance, and learning outcomes.

Other studies evaluated the impact of next-step hints by interviewing students or collecting surveys to measure their perceptions on the helpfulness of these hints. In [Pri17b], authors studied students as they completed two programming assignments, first aided by a human tutor and then by automated hints. The authors found that automated code hints were perceived as quick and easy to use, but they were also perceived as less perceptive and interpretable than human help. Moreover, Kelly Rivers' PhD thesis describes a user study conducted with the ITAP tutoring system, which offers data-driven hints for Python programming [Riv17a]. Rivers found that less experienced programmers wanted more detail and more content in their hints, compared with more experienced programmers, and that most users wanted hints to give as little information as possible, while allowing the option to see more detail if necessary.

A few studies do point to next-step hints' potential to impact learning; however, none of these studies investigated students completing whole programming tasks. Corbett and Anderson [Cor01] found that on-demand hints in their structured ACT Programming tutor, which led students through a programming problem one step at a time, improved students' performance on a post test. Fossati et al. [Fos15] found that versions of their iList linked list tutor that offered feedback, including on-demand hints, produced improved student learning. Additionally, Choudhury et al. [Cho16] found that students with access to their code style hints produced significantly better-quality solutions than students who did not. Only Rivers [Riv17a] compared students with and without access to hints during programming assignments in a classroom context. However, she found no significant difference between the conditions. These results were inconclusive due to low rates of hint-usage [Riv17a], which might be due to the hints' lack of explanation and interpretability, leading to a decrease in how often students request and follow hints, and eroding their trust in the system [Pri17e; Pri17b; Pri17d]. In addition, to my knowledge, there is no study that evaluated the generalizability of next-step hints' effect across different programming languages.

2.2.2 Empirical Evaluations of Formative Feedback

In a review on formative feedback, Scheeler et al. stated that formative feedback is effective when it is: immediate, specific, positive, and corrective [Sch04]. However, most tutoring systems in computing education are only capable of providing formative feedback with a subset of Scheeler et al's desired qualities. For example, assessment feedback systems, such as autograders, can provide corrective feedback when student code passes or fails a test case; however, this feedback lacks positivity, and it is not immediate since it is offered only when the student submits their code [Bal18; Wan20]. Other tutoring systems provide feedback in the form of text-based hints to help students reach the correct solution [Riv17c]. While these hints can be specific and corrective; they do not confirm students' correct steps. An exception is the work of Gusukuma et al., who developed a specification language that allows human experts to author immediate formative feedback to novice programmers in a block-based programming environment [Gus17], supporting various forms of formative feedback, e.g. immediate corrective or elaborative feedback. However, their specification language was tested on only one programming task, and its impact on students' outcomes was not evaluated [Gus17].

Other tutoring systems have been evaluated in prior studies to investigate the benefits of formative feedback on student outcomes; however such studies have some limitations and also left some unanswered questions [Fos15; Mit13]. For example, the most recent version of the iList tutor (iList-5), a tutoring system for linked lists, provide several forms of formative feedback, that is immediate, positive or negative, and also proactive; such that the system tries to anticipate students' future moves [Fos15]. Fossati et al. found that iList-5 is as effective as human tutors in improving students' learning [Fos15]. However, this quasi-experimental study was conducted over several semesters across different institutions, where several contextual factors might have confounded its results. The SQL tutor, a constraint-based tutor that supports students in a database course, provides students with different levels of adaptive feedback. Mitrovic et al. found that when students use SQL tutor that provides positive and negative feedback they complete problems much faster, (i.e. achieved mastery in learning), than students who received only negative feedback [Mit13]. However, the SQL tutor feedback is submission-based feedback; where the student has to submit their answer first to receive feedback, lacking immediacy.

In block-based programming environments, Gusukuma et al. developed and evaluated misconception feedback using a learner model developed by instructors' analysis of students' prior work, and found that it improved students' performance in a classroom study [Gus18]. Wang et al. created expert-authored rules to provide adaptive feedback that is positive and corrective, finding *qualitatively* that such feedback engaged students in solving short programming tasks faster [Wan20]. While both Gusukuma et al and Wang et al's methods of generating such feedback are common in literature, these methods are hard to scale across several tasks, and require extensive expert effort to be developed for complex programming tasks.

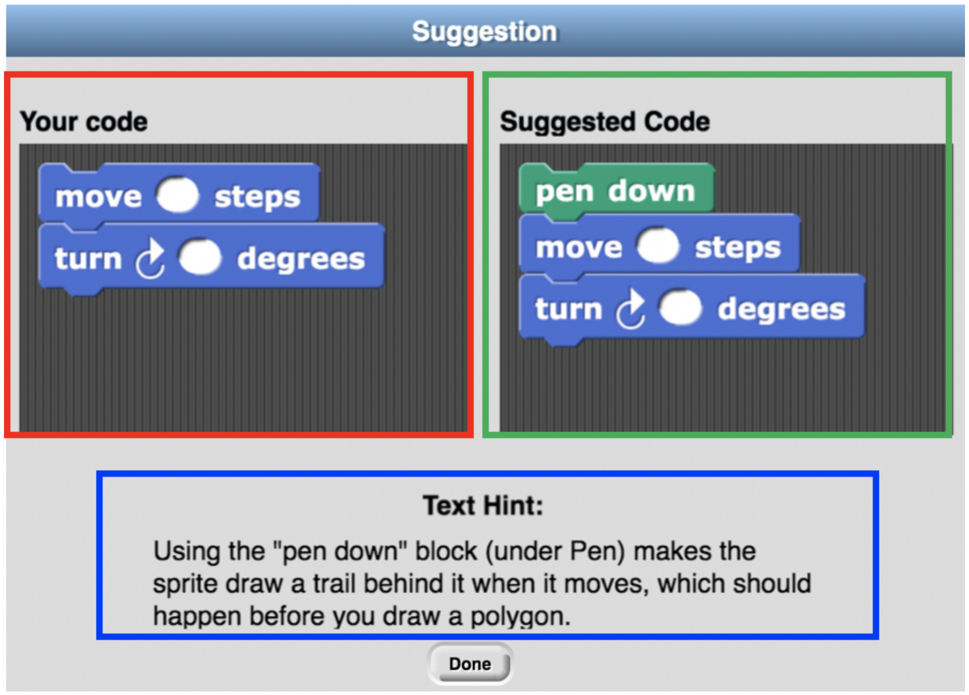
2.3 Effective Feedback Design Choices

While there are a variety of ways to generate and design automated support, it is still unclear how we can make this automated support effective. Inspired by effective human tutoring strategies, in this section, I will present related work on feedback design choices to show their strengths and weaknesses, and how they can be used to improve students' outcomes.

2.3.1 Instructional Explanations

Several computer-based tutors using expert-authored models provide automated feedback in the form of informative (or textual) hints for students [Ger16; Le09; Mit02; Mit07; Gus18]. For example, in the Ask-Elle tutor, feedback and hints are derived automatically from teacher-specified feedback messages annotated on model solutions of a given programming problem. However, data-driven hint approaches provide only single edits to students to make, without providing any textual explanations to these suggested edits. In many ways, next-step hints are similar to worked-out example problems, but rather than presenting a whole solution, they present only a small piece of it. Previous work has shown that the presence of instructional explanations can help students understand why and when certain procedures are appropriate in example problems [Chi89]. We might therefore expect that attaching textual explanations to next-step hints would similarly help students to understand them, and improve students' outcomes.

In my prior work –not included in my proposal– I conducted a study to evaluate the impact of augmenting textual explanations to next-step hints generated in iSnap programming environment, as shown in Figure 2.2 [Mar19a]. As suggested by prior work [Mit13; Pri17a], we designed textual explanations to provide three types of information to the student: (1) where they can find the suggested block in the given next-step hint; (2) the functionality of this block; and (3) why this block should be used in the correct solution. In this study, we found no difference in performance between learners who received textual explanations in addition to next-step hints (Experimental group) and those who received only next-step hints (Control group). However, we found that learners in the experimental group perceive iSnap's support as significantly more useful, relevant and interpretable and had a better understanding of the hints provided than learners who received only next-step hints.



Using the "pen down" block (under Pen) makes the sprite draw a trail behind it when it moves, which should happen before you draw a polygon.

Figure 2.2: A next-step hint with textual explanation in iSnap. The student's code is shown in the top-left (red box), with suggested next-step hint in the top-right (green box), annotated with a textual explanation (blue box).

2.3.2 Self-Explanation Prompts

Another way to help students make sense of a next-step hint could be to prompt them to reflect, by generating their own "self- explanations" about what a next-step hint means [Ren02], or why that step makes sense [Chi94]. Having students self-explain a sentence, or what a solution step means, is a constructive activity that engages students in active learning, while allowing them to monitor their understanding [Chi09]. In domains other than programming, several studies suggest that prompting students to generate self-explanations can benefit learning, such as helping them identify principles when learning from mathematics problems [Wil16] or in learning physics [Con00]. However, there is also evidence that learners may be distracted or frustrated when they struggle to come up with appropriate responses to self-explanation prompts [Shi18].

In programming, there has been comparatively less work exploring the impact of self-explanations on students' learning [Vih15; Vie17; Mar17a]. Vihavainen et al. compared the effect of self-explanation prompts with supporting multiple-choice questions versus the same self-explanation questions but without additional supporting exercises in the context of worked examples in an introduction to computing course [Vih15]. They found that students who received self-explanations with supporting multiple-choice questions performed better on a programming exam. In addition, in a year to year quasi-experiment, Vihavainen et al. found that the combined groups of students who received self-explanations performed significantly better in explanation-related exam questions than students in a previous year who did not receive self-explanations. Other studies explored different designs of self-explanation prompts. For example, Viera et al., studied the effect of asking students to write comments on MATLAB and Python code as a self-explanation strategy, referred to as in-code commenting [Vie17]. Viera et al. investigated students' perceptions about the in-code commenting activities and found that these activities helped students to better understand the code examples, to practice algorithm design, and to get used to the programming language syntax. However, authors did not investigate how in-code commenting impacted students' learning. Margulieux et al. studied the effect of prompting students to self-explain subgoals of a programming exercise in a block-based programming environment. Authors found that students who were asked to self-explain subgoals on their own performed better than students who were either asked to self-explain subgoals with expert-constructed labels, or not given self-explanation prompts at all [Mar17a].

These studies suggest the potential of self-explanation prompts to improve students' performance and learning in the domain of programming; however, more studies are needed to support these findings.

2.3.3 Encouraging Appropriate Help-Seeking

Aleven et al., noted that students generally lack the ability to seek help effectively, which obstructs the ability of learning environments with feedback to improve students' learning [Ale06]. Therefore, in this subsection we review models of how students seek help, different designs of help display in learning environments, and impact of these designs on students' help-seeking and learning. I will also describe how I can use this knowledge to improve students' help-seeking, which can therefore improve the effectiveness of automated support on students' outcomes.

Help Seeking

Help-seeking is defined as a meta-cognitive skill represented in the ability of students to solicit help from a teacher, peer, or other sources [Ale06]. Meta-cognitive skills are essential for learning [Cou00], and have been strongly linked to students' performance in CS [Ber05; Bir14]. Many help-seeking models are based on Nelson-Le Gall's help-seeking theory that states steps needed for a student to seek only the amount of help needed to complete a task, known as instrumental help-seeking [NLG81; NLG86]. Reflecting Nelson-Le Gall's and Newman's help-seeking theories [NLG81; New94], Aleven et al., presented the first help seeking model (HSM) for a computer-based learning environment (Geometry Cognitive Tutor) [Ale06]. However, the HSM depends on estimates of students' prior knowledge which is hard to assess for *novices* in programming. Also, it is unclear how to apply HSM in programming as the model features are specific to their Geometry tutor features. In programming, Vaessen's et al., developed a Discrete Markov Model (DMM) to detect students' help-seeking strategies in a functional programming tutor. Their model predicted these strategies based on students' self-reported achievement goals, which make their model context-specific [Vae14]. In a qualitative study, Price et al. identified *factors* affecting students' help seeking in programming problems, providing a baseline to consider when designing models for help-seeking [Pri17c]. Based on prior work, it is unclear what are the forms of unproductive help-seeking behavior during programming. In Section **??**, I introduce a preliminary taxonomy of novices' unproductive help seeking behavior from students' step-by-step interactions during programming in an authentic classroom setting.

Help Displays in Computer-Based Tutors

Several classroom studies, adopting Nelson-Le Gall and Newman help-seeking theories [NLG81; New94], found that students who need help are the least likely to ask for it [Puu98; Kar09; Woo99]. In computer-based environments, these results motivate researchers to explore the effect of different help displays on students' help-seeking and learning. Most learning environments provide on-demand help, i.e. providing help upon students' request, like Andes (physics tutor) [Van05], Assisstments [Raz10], and ITAP [Riv17b]. Prior work shows that on-demand help can improve students' learning [Ale06; Ren02] because it is better to give students control over requesting help [Raz10]. However, they do not prevent students' help abuse. For example, students may keep requesting hints, i.e. gaming the system, to finish tasks faster, without understanding the reason behind the hint. In an algebra tutor, Baker et al. found that help abusers learn only 2/3 as much as other students

[Bak04]. To address help abuse, Murray et al., studied delaying help when students request it, as cognitive tutors suggest that delaying help reduces help requests and improves students' learning [Mur05]. Murray et al. found that delaying help eliminated help abuse; but does not affect learning. They also found the number of hints received was negatively correlated with students' post-test scores, similar to other studies in different domains [Mar19b; Miw13].

In addition to abusing on-demand help, avoiding help has been shown as a second serious unproductive help-seeking behavior that negatively correlates with learning [Ale16b; Ale06]. For example, Baker et al., using students' log data in a Genetics Cognitive Tutor, found that help avoidance is negatively associated with students' performance on a transfer test [Bak11]. To address help avoidance, researchers allowed tutors to provide help proactively, e.g. automatically after a student makes a number of errors. Several studies, in the math domain, compared the effect of on-demand help versus proactive help on students' learning. Murray et al. found that proactive help can be more effective for some students because it can save time when a student is floundering, and provides valuable information at a time when the student is motivated to learn it [Mur06]. In the CS domain, Corbett et al., found that students who received immediate feedback (similar to proactive feedback) in the ACT programming tutor finished exercises faster than those who received on-demand one, particularly in harder exercises [Cor01]. Conversely, Razzaq et al. found that students learned significantly more from on-demand hints over proactive ones in a math tutor [Raz10].

Despite the amount of work done to evaluate students' help-seeking behavior in several domains, far less work has been done in programming [Cor01]. Research in programming tutoring environments mainly focuses on how to develop automated feedback; rather than how to display it and how students ask for it. Perhaps this is the reason why there are only a few studies that found an impact of programming feedback on students' learning [Fos15; Mar19b]. For example, Rivers in her Ph.D. thesis on ITAP, a Python tutor, found that the percentage of students requesting hints ranged from 26.9% - 53%, and therefore the impact of ITAP's help might not be clear due to the large number of students who avoided using help [Riv17b]. In this thesis, I developed a novel data-driven display for hints combining advantages of both proactive and on-demand help display to improve students' help-seeking behavior, and it can also be deployed in different programming environments as described in Chapter 4.

2.3.4 Formative Feedback Desired Qualities

In the previous sections I focused on features that can improve the impact of next-step hints, like adding more text to explain it, or prompting students to self-explain it. In this section, I will review characteristics of *effective* formative feedback, and how they can improve students' outcomes. As a reminder, research on formative feedback suggests that it is effective when it is: corrective, specific, immediate, and positive [Thu13; Shu08; Sch04], as described in more detail below.

Corrective feedback tells students not only whether their answer is right or wrong, but also provides information to help them achieve a more correct response (e.g. a hint about the solution or a corrective action) [Shu08]. While many automated feedback systems for programming highlight incorrect behavior (e.g. failed test cases [Bal18], syntax errors [Fos15]), few provide clear, actionable information to help a student address these issues, perhaps because such feedback is difficult to design [Wat11]. From a theoretical perspective, such corrective feedback would be more effective because it makes learners aware of their errors and guides them to provide the right answer [Thu13]. In practice, Gusukuma et al. conducted an empirical study showing that adding corrective information on top of detecting students' programming misconceptions (i.e. misconception feedback), improved students' performance in a post test [Gus18].

Feedback is **specific** when it provides information about *how* and *where* a student's work does or does not meet assignment goals, such as a feedback that tells the student *what* is missing in their code, not just that their code is incomplete [Ash00; Shu08]. From a theoretical perspective, Shute and Thurlings et al. argue that formative feedback is effective when it is specific and clear [Shu08; Thu13]. This is because feedback lacking specificity can increase students' uncertainty and the cognitive load needed to understand the feedback or respond to it, which can lead to decreased learning [Shu08; Wil05]. Several tutoring systems for programming have been shown to improve student performance with specific feedback, such as error highlights and detection of failed test cases [Gus18; Bal18]. However, there is still a lack of research on how to generate automated specific feedback in programming due to the multiple approaches students can take to reach a correct solution.

Feedback is **immediate** when it is provided right after a student has responded to an item (or made an action) [Shu08]. While studies have shown benefits for both delayed and immediate feedback [Shu08], some suggest that immediate feedback is more helpful for students with less prior knowledge or less motivation [Kle15; Shu08]. From a theoretical perspective, immediate, specific feedback can help students to focus their attention on the error at the time that the error occurs, where students can actively correct it before moving on and, therefore, errors do not compound and the salience of the feedback is increased [Thu13]. This is reflected by empirical studies that found an increase in novices' programming performance and learning when they receive immediate feedback [Mar19b; Gus18; Cor01].

Positive feedback *praises* students when they achieve a step (or a task) appropriately (e.g. by saying: 'Good Move!') [DE09; Fos15]. Studies in human tutoring dialogs and cognitive learning theories (CLT) show that positive feedback can increase students' confidence in their abilities, decrease their uncertainty about their answer steps, and motivate them to learn [Mit13; Lep93; Boy08; Mar21b]. Such positive feedback can also improve students' affective outcomes [Mar20a]. However, positive feedback is not always present in intelligent tutoring systems because such systems were primarily designed to intervene when they detect *incorrect* steps or solutions [Mit13; Fos15; And95]. As a result, few feedback systems can provide positive feedback proactively in a manner similar to human tutors [Fos15; Mit13].

While there are several other potential characteristics of effective feedback [Thu13], we focus on corrective, specific, immediate, and positive feedback because they are not usually available in programming environments. More importantly, evaluations of these systems have been limited only to *cognitive* outcomes. However, prior work suggests that the feedback should also impact students' affective outcomes such as students' engagement, motivation and persistence. For example, a review of the impact of feedback on persistence finds that positive feedback "increases motivation when people infer they have greater ability to pursue the goal or associate the positive experience with increased goal value" [Fis12]. This suggests not only the need for the design of feedback that embraces these best practices, but also evaluation of its impact on cognitive and affective outcomes.

CHAPTER

3

STUDY 1: AN EVALUATION OF THE IMPACT OF AUTOMATED PROGRAMMING HINTS ON PERFORMANCE AND LEARNING

This chapter was adapted from: Marwan, Samiha et al. “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 (pp. 61-70). The original text has been modified as follows: The Related Work section has been incorporated into Chapter 2.

3.1 Abstract

A growing body of work has explored how to automatically generate hints for novice programmers, and many programming environments now employ these hints. However, few studies have investigated the efficacy of automated programming hints for improving performance and learning, how and when novices find these hints beneficial, and the tradeoffs that exist between different types of hints. In this work, we explored the efficacy of next-step code hints with 2 complementary features: textual explanations and self-explanation prompts. We conducted two studies in which novices completed two programming tasks in a block-based programming environment with automated hints. In Study 1, 10 undergraduate students completed 2 programming tasks with a variety of hint types, and we interviewed them to understand their perceptions of the affordances of each hint type. For Study 2, we recruited a convenience sample of participants without programming experience from Amazon Mechanical Turk. We conducted a randomized experiment comparing the effects of hints' types on learners' performance and performance on a subsequent task without hints. We found that code hints with textual explanations significantly improved immediate programming performance. However, these hints only improved performance in a subsequent post-test task with similar objectives, when they were combined with self-explanation prompts. These results provide design insights into how automatically generated code hints can be improved with textual explanations and prompts to self-explain, and provide evidence about when and how these hints can improve programming performance and learning.

3.2 Introduction

Computer Science has the highest dropout rate of any STEM B.S. degree in the U.S. [Che13]. Half of that dropout occurs in students' first year, with multiple studies estimating the rate of students failing their first CS course at 33% [Ben07; Wat14]. Many studies have highlighted the positive influence that feedback can have on students' learning and motivation in computer science [Wat12; Mit04; Esp10]. Thus, researchers have developed a number of tools to provide automated feedback to support novice programmers, including enhanced compiler messages [Bec16], positive feedback [Fos15], and on-demand hints [Riv17c; Pri17e].

On-demand hints are particularly promising, since they can be generated automatically (e.g. using student data [Pri17a; Riv17c]), allowing them to scale to new problems and contexts. These automated hints are frequently *edit-based, next-step* hints, which suggest an edit that the student can make to bring their code closer to a correct solution. This can be conveyed through textual instructions (e.g. [Riv17c]), or by showing a "diff," contrasting the student's code with the suggested code (e.g. [Har10; Pri17e; Wat12]). In this work, we refer to these as *code hints*. Figure 3.1 shows an example of how a code hint is displayed in one block-based programming environment, iSnap [Pri17e].

However, a great deal is not known about how code hints impact learners. For example, how do learners perceive code hints? How do code hints impact learning? How can we improve the usability and impact of code hints? A number of small-scale studies have examined user logs in real classrooms, which provide some indication that code hints can help students when they get stuck during programming [Pri17e; Riv17a]. However, previous work also suggests that students sometimes find it difficult to interpret why code hints are relevant without having additional explanations [Mar19a]. Other research argues these hints may not lead to learning as they give away part of the correct solution [Ale16b; Paa18]. The challenge is that few studies have systematically compared novices doing programming tasks with and without code hints, such as by conducting randomized experiments. In addition, classroom studies typically do not gather extensive data about users' experiences through surveys and semi-structured interviews [Bec08].

This paper aims to evaluate the impact of code hints, and investigate how they can be improved, in the context of block-based programming. Specifically, we investigated whether code hints can be improved with two complementary features: textual explanations, and prompts for students to self-explain the hint in their own words [Shi08a; Sch06]. We investigated students' subjective experiences through interviews (Study 1), and conducted a randomized, controlled experiment to measure the effects of hints on immediate performance, as well as learning, as measured by success on a similar programming task without hints (Study 2).

Study 1 was a pilot study with 10 undergraduate, novice programmers, who received hints with different combinations of additional features (textual explanations and self-explanation prompts), and afterwards, we interviewed them about their experiences. The qualitative data suggested that students appreciated code hints, as these showed concrete next steps and guidance about the right direction. Students reported additional benefits of receiving textual explanations, such as elaborating the logic behind a code hint. On the other hand, students expressed a variety of opinions on self-explanation prompts, with some saying that they were confusing, and others appreciating that the prompts guided them to think deeper about the hint and how their code worked.

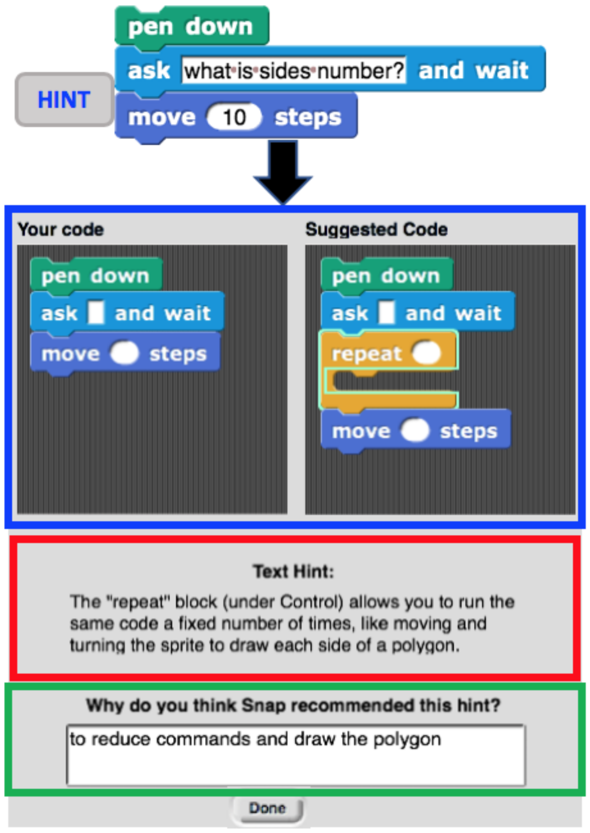
To further investigate the impact of hints, Study 2 was a randomized, controlled experiment that compared: (1) No hints; (2) Code hints with textual explanations; (3) Code hints with textual explanations *and* prompts to self-explain the hints. The study was conducted as an online laboratory-style study, rather than a classroom study. We achieved a larger samples and more experimental control by recruiting a sample of 250 Mechanical Turk workers who were novices in programming. While this population differs from students in many ways, they have been suggested as an appropriate alternative to university participants for lab studies [Beh11; Kit08] and even have been found to behave similarly to online education learners in MOOCs in some situations [Dav18]. Using this population allowed us to conduct a randomized, controlled experiment with a large number of people, which would have been harder to justify in a classroom setting with high stakes outcomes [Bec08]. Study 2 suggested that code hints with textual explanations (with or without self-explanation prompts) improved immediate programming performance, relative to no hints. In addition, we found that learning – as measured by performance on a second task where hints were not provided – was only increased when self-explanation prompts accompanied code hints and textual explanations. This improvement was also limited to objectives on the second task that closely resembled objectives from the first task.

In summary, this paper's primary contributions are: 1) Insight into students' perspectives on the value of next-step code hints, accompanying textual explanations, and self-explanation prompts. 2) Results from a randomized experiment with a convenience sample of online learners suggesting that code hints with textual explanations (with and without self-explanations) improve immediate performance. 3) Results from a randomized experiment suggesting that prompts to self-explain are necessary in order for code hints to lead to learning (better performance on a related second task).

3.3 Design of Hint Support

In this work, we built on an existing system called iSnap [Pri17e], a block-based novice programming environment that supports students with code hints. The hints are generated by a data-driven algorithm [Pri17a], which uses a database of correct solutions for a given problem to auto-generate hints. The algorithm identifies a solution that closely matches the structure of the student's current code and suggests an edit to the student's code that will bring it closer to that solution. When the system has a hint available, it annotates the student's code with a "HINT" button, as shown in Figure 3.1 (top). When clicked, the system displays a hint dialog, as shown in Figure 3.1 (blue box) with the suggested edit. This is communicated through a visual "code hint," which contrasts the student's current code with suggested code. In this work, we evaluated 2 additional features of the system's hints:

textual explanations and self-explanation prompts. Both of these features were designed to overcome limitations of code hints, identified in prior work, as explained below.



The "repeat" block (under Control) allows you to run the same code a fixed number of times, like moving and turning the sprite to draw each side of a polygon.

Why do you think Snap recommended this hint?

to reduce commands and draw the polygon

Figure 3.1: iSnap displays hint button (top). When clicked, it shows a code hint (blue box), textual explanation (red box) and self-explanation prompt (green box).

First, as Gusukuma et al. note, code hints suggest only *what* the student should do, not *why* [Gus18], and prior work suggests that this can make hints difficult for students to interpret [Pri17b]. Further, if a primary role of hints is to help students to contextualize their actions with domain knowledge, as Aleven suggests [Ale13], then code hints alone may not facilitate learning. In our previous work we addressed this limitation by adding textual explanations to code hints [Mar19a][1]. An example is shown in Figure 3.1 under

[1]In our current implementation, these explanations are written manually and selected automatically [Mar19a]. The hints evaluated in this work are therefore technically "semi-automated." However, our goal in this work was to explore and evaluate possible ways to improve code hints, and we leave questions of generation for future work.

the label “Text Hint.” These explanations were designed to complement a given code hint by conveying: 1) where to find the suggested block, 2) what the block does, and 3) how it is useful for the given assignment. In this prior work [Mar19a], we found that students appreciated the textual explanations, and that they led to improved ability to explain the purpose of hints, but they did not have any significant *additional* impact on programming performance compared to code hints alone.

A second limitation of automated code hints is that they are effectively “bottom-out” hints, telling the student exactly what to do, without requiring them to reason about the information. Prior work suggests that while such bottom out hints are necessary to help students who are stuck, they likely only lead to learning when students spontaneously self-explain the hint, which is rare [Ale16b; Shi08a]. To address this, we designed self-explanation prompts, as shown at the bottom of Figure 3.1. These prompts randomly show one of a variety of messages, such as “Why do you think the system recommended this hint?” and “What is this hint trying to help you to understand or do?” that encourage the student to think critically about the hint itself and its relation to their code. The self-explanation prompt is open-ended, and users can write anything in the response field, with at least 20 characters in order to close the hints dialog. In domains other than programming, several studies suggest that self-explanations can benefit learning (e.g. in mathematics [Wil16]), and such self-explanation prompts were particularly useful for low prior knowledge students in a biology class [McN17]. However, there is also evidence that learners may be distracted or frustrated by such prompts [Shi18], and there is comparatively less work exploring the generation of self-explanations in programming [Mor15; Vih15].

3.4 Study 1: Perceptions of Hints

Our goal with this study was to understand students’ subjective perceptions of code hints, textual explanations and self-explanation prompts, specifically when and how they are helpful, and how they can be improved. We ran a pilot study with undergraduate students at a research university where students programmed with different combinations of hint support, and we conducted interviews with students afterwards. We identified key themes from the interviews, which informed our subsequent study and offer design implications for how and when to use different kinds of hint support.

3.4.1 Methods

Population: We recruited 10 undergraduate students from an introductory engineering course at a large research university, to participate in our study by announcing the study to all students by email. As we were specifically interested in hints' impact on novice programmers, we required that participants have no prior programming courses or experience. To encourage students to participate regardless of their interest in programming, we compensated participants with a $20 gift card. To facilitate scheduling, we recruited the first 10 participants who met our eligibility criteria and were able to sign up for a study timeslot. As a result, we were unable to ensure a demographically representative population. Our participants were all males (ages 18-20), all of them were first-year students in engineering fields (7), life sciences (1), exploratory studies (1) and human biology (1). All participants reported that they had not used any block-based programming language before.

Programming Environment: All programming in this study took place inside of iSnap[Pri17e]. The system automatically offered a hint every 2 minutes by annotating the student's code with a hint button (Figure 3.1, top). This hint automatically updated as the student edited their code. This student was free to click on the hint immediately, or wait. Hints accumulated over time, such that if a student did not ask for any hints for 4 minutes, the student could then request 2 hints in a row. This setup was intentionally designed to address challenges evaluating hints in prior work, where many students either avoided hints [Pri17d; Riv17a; Ale03], or abused them by repeatedly requesting them [Ale01]. By proactively showing hint buttons every 2 minutes, the system encouraged frequent help use without forcing it, and the 2 minute timer prevented overreliance on help. To evaluate all hint types, in this study students saw 4 types of hints: code hint only, code hint with textual explanation, code hint with self-explanation prompts and code hint with both textual explanation and self-explanation prompts. Each time a student requested a hint, they were given a different random hint type, which ensured that each student saw as many different hint types as possible.

Procedure: One researcher conducted the study[2] with each participant individually over a 75 minute period. Prior to arrival, participants filled out a short pre-survey that collected demographic information. The researcher started by asking the student to read through a short tutorial on programming in iSnap for 5-10 minutes. The tutorial covered the user interface of iSnap and explained all programming concepts needed for the later programming tasks (loops, input/output and drawing) using a combination of text and

[2]Study procedures are available at: https://go.ncsu.edu/icer19-study1-procedure

short example animations. Since the goal of our study was to study how students use and learn from help, the tutorial was intentionally short, and students were expected to learn as they programmed. Next, the researcher asked the student to read the instructions of the first programming task and then asked the student to work on Task 1 for 15 minutes. Task 1 asked the student to create a program to draw a polygon with any number of sides (chosen at runtime by the user). After 15 minutes, the student was given the option to take another 5 minutes to complete the task if desired, after which they were asked to stop. Almost 90% of students were able to complete, or nearly complete, this first task. After the student finished Task 1, the researcher conducted a semi-structured interview (Interview 1) about the student's experience with iSnap. This interview lasted 4-6 minutes, during which the researcher asked questions about the specific hints that the student received during Task 1. The researcher showed the student each hint that the student had received, along with the code that the student had written at that point in time. The researcher asked the student about the timing of each hint, what was helpful about it, whether they trusted it, how it can be improved and what motivated them to ask for hints.

Next, the student completed another programming task (Task 2), working for another 15 minutes (plus 5 optional minutes). Task 2 was very similar to Task 1, using the same programming concepts, but it was more difficult. We created two versions of Task 2: an easier version in which students had to draw a strip of triangles, and a harder version in which students had to draw a design made of rotated circles. The two solutions differed by only a few blocks. Since our goal was to understand how students used hints, we wanted to ensure that they were challenged on Task 2. Any student who finished Task 1 in less than 10 minutes was given the harder assignment on Task 2. As in Task 1, students could ask for hints in both tasks, since the goal of this study was to evaluate perspectives on hints across different tasks. During all programming tasks (lasting for 15-20 minutes), students were free to request hints from iSnap, but restricted to regular intervals, as explained in detail above. In Task 2, 7 of the 10 students were able to complete it. After the student finished Task 2, the researcher conducted a second semi-structured interview (Interview 2), which lasted 9-12 minutes. As in Interview 1, the researcher asked about each requested hint. The researcher then asked some questions on each hint type[3]. Questions in the second interview helped us to gain insights about students' perspectives about each hint type, why it was helpful or less helpful, and what other types of help they were expecting to see other than the given help. This qualitative data will have useful implications on future designs of

[3]We found that 9 out of 10 students have received all kinds of hints

support in block-based programming environments.

3.4.2 Results: Interview Analysis

To gain qualitative insights into students perspectives on each hint type, we examined responses to several open-ended questions on each hint type students have received, in both Tasks. One of the authors reviewed all 10 students responses to identify both positive and negative themes that emerged for each type of hint and how they can be improved. To identify themes, the researcher grouped student responses by the type of hint being discussed (e.g. code hint), and the valence of the comment (positive, negative). We report on these themes below.

Code hints:

These helped students to see a clear next action they could take, which they could visually compare to their current work. Students (4 out of 10) declared that code hints were useful because they contrasted students' current code with suggested code: *" it just really helped when evaluating where I am and where I need to get to"* [P2][4]. Since this was the first time students had used the system *"it gives you something that you had not thought about."* [P6]. Some students appreciated the simplicity and visual nature of code hints, which allowed them to progress faster: *"I am a visual learner so the code hint ... makes it a lot quicker"* [P1]. The simple and actionable nature of code hints is in contrast to principle-based hints, which can require domain-specific reading skills to understand, even when well-written [Ale16b]. However, students (3 out of 10) did criticize code hints for saying only what to do and not helping them understand: *"it just told me what to do but I did not know what the problem is..."* [P3]. Another student felt that code hints were confusing because a *"code hint itself cannot provide enough information"* [P9].

Code hints with Textual Explanations:

These provide complementary benefits, where the explanation helps students understand the "how" and "why" of a code hint. Students (7 out of 10) noted that the textual explanation gave useful but different information from a code hint: *"[the code hint] shows which block to use and the text gives an idea of what to use it for."* [P6]. When we asked students whether they prefer code hints alone or code hints with textual explanations, 2 students noted that code hints alone can be enough, and that adding explanations *"can not be not helpful. If you do not need then you do not need."* [P9], and *"I did not find it as helpful as doing it myself."* [P10]. The rest of students preferred having both because

[4]P2 indicates that this quotation was from Participant 2.

“the text hints helped me to understand the visual hints [code hints] on a deeper level"[P7]. These results suggest that students appreciate explanations with code hints, and there is little cost to the student, as they are easy enough to ignore.

Self-explanation Prompts with Hints:

These can help students stop and think more deeply about the hint. Some students (4 out of 10) saw the value in pausing to think and self-explain: *“it made me think and take a step back about the whole process."* [P7] and *“it adds a value... it just makes you go on and look at it again."* [P10]. Other students (2 out of 10) mentioned that the self-explanation prompt enhanced code hints as *“it helped to interpret what does the picture mean.*[P2] and *“the question forced you to figure how this helps you so you really understand rather than just looking at it and dismissing it"* [P8].

Self-explanation prompts were criticized for being frustrating and confusing. A few students (3 out of 10) had vocally negative comments on adding self-explanation prompts, as confusing: *“I was confused about the question... because I did not know what the hint was giving to me."* [P3] and not giving any support *“it is not giving me anything back, it is just asking me if I understood it."* [P5]. We observed that most of the students who were more critical of the prompts were also not able to complete both programming tasks. This agrees with prior work suggesting that students with lower prior knowledge may have difficulty with open-ended self-explanation prompts, since they lack the domain knowledge to construct meaningful explanations [Roy05].

Other Insights that Emerged From Interviews:

To better understand why students actually needed help, we asked them, "what motivated or encouraged you to ask for help?" Most students said they needed help when they are lost: *“when I did not know what to do next."* [P9] and *“when I have used all the possible options, the hints help to move to the next step."* [P5]. Other students thought that getting a hint at any time would be helpful: *“whether I was doing something right or wrong it just put me in the right direction."* [P2]. Most students preferred to get hints at the beginning of the task: *“just starting out at first that really helps... I want to at least know how to start it."* [P1]. However, a few students preferred hints *“probably a little later"* [P10], or after playing around for a while in order to *“see what I can get on my own and go deeper... then once I felt lost, I pressed the hint button."* [P7]. Students’ answers clarify that there is no specific *time* where all students agree to have hints at. This suggests that it is preferable to keep hints available to students all the time, but in such a way that prevents hints abuse.

When students were asked *how code hints can be improved*, all of them appreciated them as they were. As for improving textual explanations when added to code hints, one student suggested to put *“the text first and then the visuals"* because when the explanation comes after the code hint *“people will skip it to get this over"* [P5]. Other students suggested making explanations more helpful by *“add[ing] examples on what to do"* [P6]. For self-explanations, a few students suggested to have them *“after the assignment, like a feedback, and then you can reflect upon the whole process."* [P5].

Finally, the researcher asked students what other hint types they prefer to receive. Most students suggested that the current ones are enough, some suggested that in addition to providing specific hints related to students’ code, *“adding general hints... would be really helpful."* [P6]. Another student suggested having hints that indicates their progress such that *“the program can pop a hint when I am doing something wrong... and if I am doing something right it says like ‘good work’ "* [P5].

Conclusions: Our results suggest that, overall, students see the benefits in all three types of hint support, which offer complementary benefits. However, some students did find self-explanation prompts to be confusing or irritating, so it is important to investigate how they impact student outcomes.

3.5 Study 2: Learning and Performance

Having positive insights on hints types from students’ interviews, we were encouraged to investigate the effect of hints on performance and learning transfer on a large-scale study. To do so, we recruited crowd workers through Amazon’s Mechanical Turk (MTurk) platform who have attested to have little to no prior knowledge in programming. Previous work has shown that recruiting crowd workers can be an effective form of conducting large-scale user studies in lieu of using university participants [Beh11; Kit08], and this has been previously employed in computing education research to assess the efficacy of online learning approaches [Lee15]. This experiment aims to answer the following research questions: What is the effect of hints with and without self-explanation prompts on: 1) learners’ perspectives on iSnap’s helpfulness with and without receiving hints? 2) learners’ performance?, and 3) learning, as measured by performance on future tasks without hints?

3.5.1 Methods

Population: In this study, we recruited 250 total crowd workers through Amazon’s Mechanical Turk (MTurk) platform. As in Study 1, we recruited only participants who attested to having no programming experience[5], and we compensated them for participation ($4.50). To ensure that our data included only participants who made an honest effort at the programming tasks, we excluded from our analysis those participants who did not attempt Task 1 (no edits). We also excluded those who attempted a task multiple times by resetting the environment, or reported having prior programming experience (despite claiming to meet the eligibility requirements), leaving 201 total participants. The recruited learners included 118 males, 80 females and 2 learners did not specify their gender. The median age was 25-34, and 84.5% reported their education level as either a BS degree or some college credit. While this population is different in important ways from college learners, it is also demographically similar in age and education level.

Procedure: This study used a similar procedure to Study 1, though it was conducted online rather than in person. Learners completed the same tutorial and the same 2 programming tasks in the same programming environment, and received the same automated hints. However, since our goal with Study 2 was to measure the impact of hints on performance and learning, there were 4 differences in the procedure. First, each learner was randomly assigned to one of three conditions that determined what type of hint (if any) our system provided in the first task: 1) no hint (control condition), 2) **C**ode hints with **T**extual explanations (CT), and 3) **C**ode hints with **T**extual explanations and self-**E**xplanation prompts (CTE). Unlike in Study 1, we intentionally chose to always provide textual explanation with code hints, since our results from Study 1, as well as our prior work [Mar19a; Pri19] suggested that textual explanations may improve the likelihood of learners benefiting from hints with few drawbacks. However, students' mixed reactions to self-explanation prompts in Study 1 encouraged us to create a separate condition for hints with explanation prompts (CTE). Our population included 63 learners in the no hint condition, 79 in the CT condition and 59 in the CTE condition[6]. As in Study 1, the programming environment provided learners with the opportunity to request a hint every two minutes, but in Study 2 the type of hints was always the same, dictated by their condition (e.g. a code hint with textual explanations), not random. Our timed approach ensured that, in a 15-min task, every learner can have from 0 to 7 hints maximum.

Second, to measure if hints can improve learners' performance in future tasks, we used Task 2 as a post-test, which did not offer hints to learners in *any* condition. We chose to

[5]While participants could have lied about their level of programming experience, the data suggest that the majority of participants were novice programmers, and those with experience would be distributed randomly across conditions.

[6]An error in our random assignment process caused there to be more participants in the CT condition. We carefully verified that was due only to random assignment, not disproportionate dropout.

use Task 2 to assess learning, rather than a traditional post-test, as we were interested in measuring learners' ability to perform a similar task without help. We were also able to use fine-grained analysis of learners' Task 2 programming logs to compare how learners performed across time in different programming objectives. Unlike in Study 1, we had only one version of Task 2, where learners had to draw a strip of triangles (the easier version).

Third, since this study was conducted online, we could not interview participants. Instead, we provided post-survey after both tasks, as explained below. Fourth, all participants had exactly 15 minutes to complete each programming task, with no extra time.

Measures: We analyzed 2 primary sources of data from learners: Post Task 1 Survey and log data. We also collected a Post Task 2 survey, which is not analyzed here. Post Task 1 Survey asked learners to rate the overall helpfulness of the programming environment on a rating scale from 1 to 10. It asked users to elaborate on their judgments and to explain in what situations this action was most useful, though we do not analyze this data in this work. In addition to the survey, we also collected log data of learners' work in our system, including complete code traces. For both programming tasks, we gave learners the same amount of time (15 minutes). We chose tasks that would take most learners at least 15 minutes, so we used the number of objectives completed in this time as our measure of programming performance. We defined 4 objectives (e.g. "draw a shape" or "correctly get and use input from the user"), such that each objective was independent, and completing all 4 indicated successful completion of the whole task. We developed an automatic grader to determine the number of objectives completed by each participant, and we manually verified the auto-grader's accuracy on 100 submissions of each task.

3.5.2 Results

The Impact of Hints on Learners' Programming Performance and Learning:

To measure the effect of hint condition on learners' immediate performance, we compared the number of objectives that learners completed during Task 1 in the control group (M[7]=2; SD=1.40), CT (M=2.53; SD=1.24) and CTE (M=2.8; SD=1.31). A Kruskal-Wallis test[8] shows a significant difference among hint conditions for learners' performance ($\chi^2(3) = 12.84, p = 0.001$). Afterwards, we used post-hoc non-parametric Dunn's test with Benjamini-Hochberg correction for multiple comparisons to determine pairwise significant differences across hint conditions [Ben95; Dun64]. Dunn's test shows a significant difference

[7]Though the data was not normally distributed, we report averages with SD, rather than medians, since the number of objectives quite small.

[8]We used non-parametric tests, as our data were not normally distributed.

between control group and CT learners ($z = 2.16$; $p = 0.045$), a significant difference between control group and CTE learners (z=3.56; $p = 0.001$) and non-significant difference between CT and CTE learners ($z = 1.62$; $p = 0.104$). This shows that both conditions with code hints completed significantly more objectives than the control condition. As a result, more learners completed all of Task 1 in the CT condition (27.8%) and CTE condition (45.8%) than the control condition (22.2%). Figure 7.5 (left) plots the mean number of objectives that had been completed by learners in each condition at different times throughout the 15 minute task. It shows that the difference between the three groups became more pronounced over time. **These results suggest that code hints with textual explanations, and code hints with both textual explanations and self-explanations prompts significantly improve performance.**

Task 2 served as our measure of learning, since learners had no hints on this task. It consists of 4 objectives. The first 2 objectives were identical to the first 2 objectives in Task 1 ("draw something" and "ask user for input X and repeat X times"), and they measured how well learners learned to repeat these steps in a new context. The last 2 objectives measured learners' ability to apply the same programming constructs (loops and drawing) in a new way ("repeatedly draw a triangle" and "draw a strip of triangles"). For example, one solution for the third objective required nested loops.

We first compared the *total* number of objectives completed by learners in the control group (M=2.09; SD=1.43), CT condition (M=1.9; SD=1.47), the CTE condition (M=2.47; SD=1.40). A Kruskal-Wallis test showed that this difference was not significant ($\chi^2(3) = 5.58$; $p = 0.06$). These results were inconclusive, and we hypothesized that hints might have only impacted future performance for isomorphic objectives. We therefore compared performance on *only the first two objectives* on Task 2, which were identical to the first two objectives in Task 1. A Kruskal-Wallis test shows a significant difference across groups in their performance of the first 2 objectives ($\chi^2(3) = 8.54, p = 0.013$). A post-hoc Dunn's test with Benjamini-Hochberg correction shows a significant difference between CTE learners and both the control group and CT learners (($z = 2.73$; $p = 0.01$), ($z = 2.35$; 0.028)) respectively, however, no significant difference between CT learners and the control group ($z = 0.53$; p=0.59). We found that 41.2%, 49.3%, 67.7% of control group, CT and CTE learners, respectively, were able to finish the first 2 objectives of Task 2. **These result suggests that only code hints that have self-explanation prompts improved learners' performance on future tasks without hints, specifically on objectives that learners saw before in Task 1.**

Hints Request Rate, Follow rate and Processing Time in Task 1:

While our system offered hints automatically, starting at 2 minutes, learners were free to

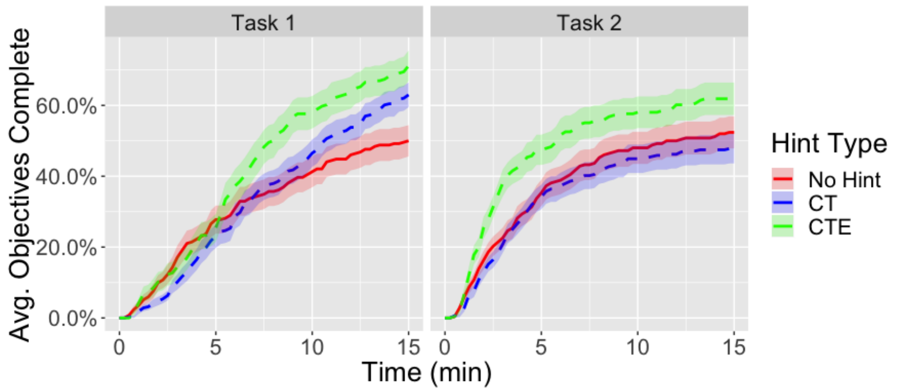


Figure 3.2: Learners' average completion progress (with shading indicating standard error) in each condition, measured over time, for Task 1 (left) and in Task 2 (right).

open these hints or ignore them. We found nearly all learners in both conditions requested at least 1 hint (98.7% in the CT group, and 96.6% in the CTE group). However, we found that the number of hints requested by learners in the CT condition (Med=5; IQR=4) was greater than that in the CTE condition (Med=4; IQR=4), and a Mann-Whitney U-test showed that this difference was significant ($U = 1649$, $p < 0.01$, Cohen's $d = 0.477$). Furthermore, we found no significant spearman correlation between the number of hints requested and the number of objectives completed on Task 1 in both the CT group ($r = -0.101$, $p = 0.37$) and CTE group ($r = -0.135$, $p = 0.31$). We did find a weak but significant, negative spearman correlation between the number of hints requested on Task 1 hints and Task 2 performance in the CT group ($r = -0.243$, $p = 0.03$) but not in the CTE group ($r = -0.102$, $p = 0.40$). This suggests that the *number* of hints requested does not strongly predict performance on current or future tasks.

Learners could also choose whether or not to follow hints that they received. We defined a learner's *follow rate* as the percentage of requested hints that a learner followed, meaning they used the block suggested by the hint within 120 seconds of seeing a hint. We found the average number of hints followed by each learner in CTE group (Med=0.8; IQR=0.5) was greater than that by CT group (Med=0.66; IQR=0.42), and a Mann-Whitney U-test showed this difference was significant ($U = 2686$, $p = 0.036$, Cohen's $d = 0.33$). We also measured how long learners kept the hint dialog open before closing it, as a rough measure of how long they took to process the hint. We found the average time taken by each learner to dismiss the hint dialog in CTE group (Med=27.4; IQR=17.3) was greater than that by CT group (Med=16.3; IQR=9.96) and Mann-Whitney U-test shows that CTE learners have spent significantly more time processing hints than CT learners ($U = 3353$, $p < 0.01$, Cohen's $d = 0.72$). **Our results suggest that self-explanation prompts encourage learners to take longer to view hints, request fewer hints and follow more of the hints they requested.**

Users' Ratings in Post Task 1:

We compared ratings on iSnap's usefulness between learners collected in Post Task 1 Survey. We found helpfulness ratings of learners in the control group ($Med = 5$, $IQR = 4$) was much less than both the CT learners ($Med = 7$, $IQR = 3.4$) and CTE learners ($Med = 8$, $IQR = 2$). A Kruskal-Wallis test showed a significant difference in learners' rating across conditions ($\chi^2(3) = 29.72, p < 0.01$). Afterwards, Dunn's test showed a significant difference between control group and both CT learners ($z = 3.91$; $p < 0.01$) and CTE learners ($z = 5.34$; $p < 0.01$), but no significant difference between CT and CTE learners ($z = 1.78$; $p = 0.07$). **This results suggested that iSnap was perceived as significantly more useful when providing hints.**

3.6 Discussion and Limitations

In this section we discuss our primary results from Study 2, and how our interpretation of them can be informed by our qualitative results from Study 1.

Code hints with textual explanation improve learners' immediate programming performance. In Study 2, learners in the CT and CTE conditions were able to complete 25% and 40% more of Task 1, respectively, than the control group without hints. Helping students to progress when stuck to complete a problem is one of the primary purposes of next-step hints [Ale16b], but it was not obvious that they would accomplish this. We found that students were in fact much more likely to complete all four of Task 1's objectives with hints (over twice as likely in the CTE group). Our findings from Study 1 suggest that students are aware of code hints' ability to improve immediate performance, and appreciate the hints' ability to help put them *"in the right direction."* [P2]. This is a similar theme in prior work on students' perceptions of code hints [Pri17b; Pri19].

Self-explanation prompts changed the way that students interacted with code hints. In Study 1, our participants frequently noted that self-explanation prompts caused them to *"... think and take a step back about the whole process."* [P7]. Study 2 helps us to better understand quantitatively what impact the prompts had on learners' use of hints. We found that the median student with hints *and* self-explanation prompts spent 64% more time viewing each hint than students without the prompts, which agrees with our qualitative findings. This suggests that these prompts may have *"forced you to figure how this [hint] helps you, so you really understand rather than just looking at it and dismissing it"* [P8]. Further, we find that learners in this group asked for only 67% as many hints, but were 25% more likely to follow them. This may be evidence that learners are getting more out of the hints that they read, as students told us in Study 1 *"it helps me contemplate my thoughts process."* [P7]. One might expect that asking for fewer hints would have a negative impact on learners' immediate performance, since they see fewer pieces of a correct solution. However, we found that learners in the CTE condition did no worse than their CT counterparts, and may have even performed a bit better, with 64% more students finishing all of Task 1's objectives.

Code hints only improved learning when accompanied by self-explanation prompts. We found that learners in CTE group on Task 1 performed 23% better overall on Task 2 than the control condition. However, learners with only code hints, but not prompts (CT group), performed no better than the control condition. Prior work suggested that students learn best from hints when they *spontaneously* self-explain their meaning [Ale16b; Shi08a], and our results suggest that this can also be encouraged with prompting them to self-explain. This students' need for self-explanation support may explain why Rivers' prior evaluation of automated hints did not find a learning effect [Riv17a]. The students we interviewed in Study 1 seem to be aware of this need, as most (though not all of them) appreciated self-explanation prompts' ability to encourage them *"to interpret what... the picture [hint] mean[s].)"* – or what the KLI framework might call the sense making necessary for learning from hints [Koe13]. We also note that we can make no claim about whether or how textual explanation contributed to learning, since we chose not to investigate this in Study 2.

Code hints with self-explanation prompts improved learning, but only on isomorphic tasks. The primary learning impact of code hints with textual explanations and self-explanation prompts seems to be on learners' ability to perform same programming objectives that they have previously accomplished with hints. We found that the CTE condition performed significantly better on these isomorphic Task 2 objectives than the control group, but not on other objectives that were different from Task 1 and more challenging. We note that this investigation of isomorphic objectives was a post hoc analysis, which we performed after finding inconclusive results about hints' overall impact on learning. Future work should investigate the hypothesis that code hints with both textual explanations and self-explanation prompts may be more effective for helping students to repeat things they have already done than completing new tasks.

What can we learn from these results? Our results are an important initial step in understanding the potential benefits and limitations of using automated programming hints in classrooms. Systems that offer automated hints *without* self-explanation prompts are currently in use in classrooms (e.g. [Pri17e; Ger16]), and our results suggest that their designers might consider adding this feature. We acknowledge that our studies focused on short (15 minute) programming assignments, and our quantitative results relied on a convenience sample of crowdworkers, so *we should be cautious in generalizing these results* to other contexts. However, we argue that given the lack of existing empirical results on the efficacy of programming hints, these initial, positive results still provide important insight. It is difficult to create large-scale, controlled classroom studies, but our results justify the need for such studies, and for identifying important hypotheses to test in these studies (e.g. the importance of self-explanation prompts). For example, if hints create a learning impact over just 15 minutes of programming, it is possible that the effect may be much larger over a whole semester, but this can only be verified empirically.

3.6.1 Limitations

In Study 2, our population consisted of paid crowd workers with no prior programming experience. Their motivations, and prior knowledge may differ from those of other populations of learners when programming hints are used, and we emphasize that this limits the generalizability of our results. However, such an approach is not unprecedented in computing research [Lee15], and we argue that we can still gain valuable insight from this population, as suggested by prior work [Beh11; Kit08]. Working with this population allowed us to collect a large amount of data, randomly assign participants to conditions, and collect a more gender-balanced dataset – all of which can be quite difficult in a classroom setting. In addition, our quantitative results from crowdworkers also strongly parallel our findings from novice students in Study 1

In Study 1, our participants all identified as male. Since prior work suggests that gender plays an important role in how students seek and use help [Ale03; But98], this limits the generalizability of our results. Additionally, in both studies, we only studied users during two simple, 15-minute programming tasks, and we have begun further work to investigate if our results generalize to longer or more complex tasks in classrooms. We argue that this short duration likely made it *more* difficult to detect an effect of hints on learning. However, it likely limited the diversity of help-seeking scenarios that users encountered in Study 1.

3.7 Conclusion

This paper presented two studies to assess and investigate next-step programming hints' impact on learners' performance, learning and perspectives. We have attempted to provide useful insights to the education community on when and how these hints are useful and how to improve automated support in programming environments. This paper's primary contributions are: 1) Insight into students' perspectives on the value of next-step programming hints and accompanying textual explanations and self-explanation prompts; 2) A large-scale evaluation of the impact of programming hints on performance and learning with a convenience sample of online learners; and 3) Insight into the conditions (e.g. code hints *with* self-explanation prompt) under which hints can contribute to learning, and when they may hinder learning. Specifically, we found that code hints with textual explanations improved students' performance, and they improved learning on isomorphic objectives in future tasks when accompanied by self-explanation prompts. Our results motivate future work to investigate whether our results generalize to a classroom context, and to further explore the specific contexts under which programming hints can lead to learning.

CHAPTER

4

STUDY 2: UNPRODUCTIVE HELP-SEEKING IN PROGRAMMING: WHAT IT IS AND HOW TO ADDRESS IT

This chapter was adapted from: Marwan, Samiha et al. “Unproductive Help-seeking in Programming: what it is and how to address it”. In Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (pp. 54-60). The original text has been modified as follows: The Related Work section has been incorporated into Chapter 2.

4.1 Abstract

While programming, novices often lack the ability to effectively seek help, such as when to ask for a hint or feedback. Students may avoid help when they need it, or abuse help to avoid putting in effort, and both behaviors can impede learning. In this paper we present two main contributions. First, we investigated log data from students working in a programming

environment that offers automated hints, and we propose a taxonomy of unproductive help-seeking behaviors in programming. Second, we used these findings to design a novel user interface for hints that subtly encourages students to seek help with the right frequency, estimated with a data-driven algorithm. We conducted a pilot study to evaluate our data-driven (DD) hint display, compared to a traditional interface, where students request hints on-demand as desired. We found students with the DD display were less than half as likely to engage in unproductive help-seeking, and we found suggestive evidence that this may improve their learning.

4.2 Introduction

Providing students with feedback serves as an essential element of learning theories [Luc07; Thu13]. Traditionally, feedback comes from an instructor, but with the growing enrollment in Computer Science (CS) classes [Ass17], researchers put extensive effort in developing programming environments to provide automated feedback to students for the same purpose. Several studies showed that this feedback, like simple test-case feedback [Bal18; Joh16], or hints suggesting a single edit to proceed [Riv17b; Fos15] can increase students' performance and learning [Cor01; Mar19b; Gus18]. However, this positive impact requires the student to use help effectively, which is a difficult metacognitive skill [Ale06]. Despite the prevalence of automated help in programming classrooms [Cor01; Riv17a; Mit13], little prior work has investigated how to encourage students to seek and use this help more effectively.

A large body of work in mathematics tutoring systems suggests that students' unproductive use of help features is a serious challenge [Ale03; Ale16b; Raz10]. For example, Aleven et al. showed that 72% of students' actions in a Geometry tutoring system represented unproductive help-seeking behavior, such as avoiding help when they needed it or abusing help to speed through the assignment [Ale03]. Further, unproductive help-seeking predicts poor learning outcomes [Ale06; Bak04], suggesting that students need additional guidance on how and when to use help. A naive approach might be to limit the amount of help provided (e.g. give no more than 3 hints). However, previous studies show that students with different prior knowledge, need different amounts of help: " the lower the prior knowledge, the higher the need for assistance" [Kar09; Woo99]. Therefore, not only is it important to encourage students to effectively seek help, but it is also important to tailor the amount of help to their performance [Raz07]. In programming environments, to design such guidance, we first need to understand how students engage in help-seeking in computing classrooms, which is unclear from prior work. To our knowledge, prior work has been limited to analyzing survey data [Vae14], or characterizing students' motivations for seeking and avoiding help [Pri17c]. More work in the programming domain is needed to identify the specific *ways* that students unproductively seek help, to guide instructors and inform the design of feedback interfaces to address these behaviors.

In this work, we investigate the following research questions: RQ1) What specific forms of unproductive help-seeking do novices engage in when programming? RQ2) How can the design of a help interface improve students' help-seeking behavior? To address RQ1, we investigated log data, capturing real students' use of hints during independent programming homework and we propose a preliminary taxonomy of novices' unproductive help seeking behavior. Similar to Aleven et al. [Ale06], we found the majority of students engaged in some form of unproductive help-seeking, avoiding or abusing it. To address RQ2, we designed a novel system for displaying adaptive hints. It uses a data-driven (DD) algorithm to encourage students to ask for help, based on their progress compared to prior students. We conducted a small pilot study to compare students' help-seeking behavior when working in a programming environment with our DD display, compared to a traditional on-demand hints display (i.e. hints provided upon students' request). We found students in the DD display condition never abused help and were less likely to unproductively avoid it, compared to students in the on-demand condition. We also found suggestive evidence that this improved behavior may translate into learning, but further work is needed to verify this. In addition, despite the DD display limiting students' ability to use help, students perceived it to be at least as helpful as the on-demand display, and they noted its intelligence in providing hints only when they need it.

4.3 Study 1: Defining Unproductive Help-Seeking in Programming

The goal of this study is to address our first research question: What specific forms of unproductive help-seeking do novices engage in when programming? From this study, we propose a preliminary taxonomy of unproductive-help seeking in programming, which informs the design of a new hint display, presented in Section 4.4.1.

4.3.1 Method

To explore *how* students in authentic classroom settings unproductively seek help, we looked at how all students (n = 50) used hints during two homework assignments in an introductory computing course for non-majors in a public university in the United States[1]. In the first part of this course, students were studying block-based programming by solving exercises from the BJC curriculum [Gar15]. The first homework is called "Squiral", and the other one is called "Guessing Game 2". In Squiral, students write a procedure that takes three parameters (R, L, and T) to draw a square-like spiral where the spiral has *R* rotations, of side length *L* and side thickness *T*. In Guessing Game 2, the program should generate a random secret number within a provided range and ask the user to guess this secret number, keeping track of how many guesses the user has made. The two exercises together required the use of loops, variables, conditionals, procedures and lists.

Students programmed their homeworks in iSnap [Pri17e], which is an extension to the block-based Snap*!* programming environment that provides on-demand hints. These hints suggest an edit a student can make to bring their code closer to the correct solution. iSnap also logs all students' interactions during programming (e.g. creating, or deleting a block) as a *trace*, allowing researchers to replay all of a student's steps in this trace. Students produced an average of 351 and 740 edits, on Squiral and Guessing Game 2, respectively. To capture students' authentic help-seeking behavior, we focused on homework assignments, where students work *independently* and typically use automated help, rather than instructor help. By studying students' log data, we can observe their behavior without influencing it (an instructor's presence might deter some types of help abuse). While log data has some limitations, its analysis has led to substantial research insights in computing education [Ale06; Riv17b; Don19].

We applied a mixed-methods approach to identify and categorize types of unproductive help-seeking in our log data. Two researchers, one of whom had extensive experience analyzing students' log data in iSnap, started off by reviewing prior work on taxonomizing help-seeking behavior [Ale06]. Afterwards, the two researchers analyzed students' log data through a 3-phase method, adapted from Dong et al. [Don19]: (1) Phase 1: *Initial Definition*: the two researchers independently, manually inspected half of the Squiral log data, initially focusing on students' hint requests. They noted the time taken to view each hint, whether the student followed the suggestion in the hint, and the relationship between hint requests. The researchers used this data to identify initial categories of unproductive help-seeking,

[1]We did not have access to students' demographic information.

and discussed these categories to produce initial definitions. (2) Phase 2: *Consensus*: the two researchers divided the second half of the first homework into two sets and each tagged students' data with the initial definitions. Finally, both researchers resolved their disagreements and discussed their findings with two other Snap*!* instructors who have previous research experience with help-seeking behavior, and then they refined their definitions accordingly. (3) Phase 3: *Verification*: the two researchers divided the second homework into two sets, and each tagged a set by the resulted categories they defined in the previous phase. Neither researcher identified any new behaviors beyond those defined in Phase 2.

4.3.2 Results and Discussion

We organized our results into a preliminary taxonomy of novices' unproductive help seeking in programming. As in prior work, we found that help abuse and help avoidance are the two main forms of unproductive help-seeking behavior; however, we further refined these into subcategories, which reflect distinct, but not mutually exclusive, cases of these behaviors. We compare our results to Aleven et al.'s Help Seeking Model (HSM), discussed in Section 2.3.3.

Help abuse: the behavior of asking for too much unnecessary help. Based on our data we categorized it into three subcategories:

1- Immediate Help: Students who exhibit this behavior request hints *before* beginning to program a task or objective on their own. For example, some students started their homework by clicking on the help button, and this often predicted high reliance on hints throughout the assignment. This suggests students are deliberately "gaming the system" [Bak04], which will likely obstruct learning. When investigating help seeking in the Geometry tutor, Aleven et al. argue that an *initial* help request can be legitimate for students who have no idea where to start. However, Ko et al. suggest that for novice *programmers*, these "design barriers" are best addressed with more comprehensive help, such as examples, which is rarely provided by automated help [Ko04]. Therefore, we argue that attempting to use low-level feedback in this case constitutes help abuse.

2- High Frequency: In this behavior, students request and consistently follow a high number of hints in a very short time (e.g. five hints in one minute). While this may accompany Immediate hint requests, it can happen at any time during the assignment.

High Frequency behavior is very similar to the help abuse form "clicking through hints" in the HSM, where a student moves to the next hint before spending enough time with the current one [Ale01]. However, in novice programming "clicking through hints" might not always be hint abuse. Because programming problems have vast solution spaces [Riv17b], there are many relevant hints a system could provide, and novices may need to view many of them before finding a relevant one [Mar19a]. Therefore, our definition and tagging distinguished between students who viewed many hints and *followed them*, versus those who simply viewed many hints and followed one.

3- Unneeded Help: In this behavior, students ask for repeated hints, despite having demonstrated an ability to progress independently. For example, some students completed half of the solution successfully with little-to-no hint usage, and then switched to relying almost exclusively on hints, sometimes even destroying all their original code. Unlike the first two categories of abuse, whether a hint request constitutes Unneeded Help depends on the student and their current progress.

This behavior is similar to the help abuse form “Ask hint when skilled enough to try a step” in the HSM; however, the HSM defined it based on estimating students’ prior knowledge [Ale06], which is hard to measure for novices, especially in programming. We assumed students are skilled if they were making adequate progress but then decided to rely on hints without trying.

Help avoidance: this is when students fail to ask for hints when struggling and a hint could help them move forward. While our trace data allowed us to identify instances of help avoidance in an authentic homework setting, it did not give us clear insight into students’ motivations for avoiding help. Therefore, we discuss two broader categories of help avoidance and discuss them in the light of prior work that investigated *factors* affecting students’ help-seeking in a laboratory study [Pri17c].

1- No Use: Some students spent a long time with unproductive work with no help requests. For example, one student programmed Squiral iteratively, i.e. as a series of "move" and "turns" without using any variables or loops, making a total of 40 blocks. This solution is inefficient and does not match the grading rubric since the student is not using parameters. However, the student did not request a single hint. Prior work suggested that this behavior might be because students wanted to be independent, or they were not aware of the help button[Pri17c], which suggests the need for a hint display interface that reminds students to ask for a hint.

2- Stopped Use: In a given homework, some students requested hints at the beginning, and then stopped doing so, even when they struggled. Some students even submitted their code incorrectly without asking for help again. Prior work shows that some students may have found their initial hints to be uninterpretable [Mar19a] or untrustworthy [Pri17c], and therefore avoided requesting more help.

Table 4.1: Percent of students tagged with each help-seeking category from the Taxonomy in Study 1: HW1 and HW2, and Study 2: data-driven (DD) vs. on-demand (OD) display.

		HW1	HW2	DD	OD
Abuse	Immediate help	8%	15.21%	0%	0%
	High-freq.	20%	21.73%	0%	0%
	Unneeded help	24%	32.60%	0%	20%
Avoidance	Stopped Use	16%	4.34%	26.66%	13.33%
	No Use	14%	23.91%	0%	33.33%
	Productive	42%	26.08%	73.33%	33.33%

Because this taxonomy reflects specific unproductive help-seeking behavior inferred from students' log data, it makes it applicable to design programming environments' interfaces that can deter this behavior. While we describe these behaviors in terms of hints in our data, we argue they generalize to other forms of help (e.g. enhanced compiler messages [Bec16], and misconception feedback [Gus18]). Table 4.1 shows the percentages of each unproductive help-seeking category detected from students' data. From Table 4.1, we found 58% and 74% of students' help-seeking behavior is unproductive, in Squiral (HW 1) and Guessing game 2 (HW 2), respectively. These alarming results motivated us to design an interface for displaying hints to moderate students' help-seeking based on our taxonomy.

While this taxonomy reflects the same broad categories of help abuse and avoidance in prior work [Ale06], it also defines more specific subcategories, where *each* inform classroom instruction and the design of help interfaces. For example, while High Frequency abuse can be addressed by limiting the frequency of hint requests, this will not address many forms of Unneeded Help.

Our results have two important limitations. First, they are based on only two assignments in a single environment with hints, though our findings' close alignment with prior work suggests they will likely generalize to other classrooms and forms of help. Second, while our use of log data captured authentic help use, it limited our ability to infer students' motivations, especially for help avoidance. However, our results represent a novel contribution to the computing education literature, as our taxonomy is the first to define unproductive help seeking behaviors, grounded in real data, focusing explicitly on programming classrooms. Also, our results can directly inform the design of novel help systems, as discussed in Section 4.4.1.

4.4 Study 2: Improving Help-seeking

This study addressed our second research question: How can the design of a help interface improve students' help-seeking behavior? To provide a solution to this question, we designed a novel Data-Driven (DD) display for hints (described in Section 4.4.1) to address the unproductive help-seeking behavior defined in Study 1. In addition, we conducted a pilot study to evaluate the impact of our display on students' a) help-seeking behavior, b) learning, and c) perceptions of the helpfulness of the programming environment.

4.4.1 Design of Data-Driven (DD) Display

We designed a novel hint display in iSnap to address each category in the two high level problems we identified, help abuse and help avoidance. A primary finding of Study 1 is that both help abuse and help avoidance can be defined – and addressed – with respect to the student's progress, rather than simply based on the frequency of help requests. Higher levels of help may be appropriate for some students and not others (e.g. "the lower the prior knowledge, the higher the need for assistance" [Rol14; Kar09]). Thus, a key feature of our hint display is a way to measure a students' progress over time, and whether it represents a need for help. We do so using a data-driven (DD) approach, that measures the average time taken by prior students to complete each objective (e.g. "draw a shape") of the same task. We then measure a students' progress in real time and compare it to the average of prior students – if they have taken longer than expected to complete a given objective, we assume they may need help. Our DD approach does not rely on expert judgements of how long each objective should take, which can be biased by the expert blindspot [Nat01]. We use an autograder, as is common in many programming environments [Bal18; Joh16; Wan20], to assess objective completion. We use this DD progress measure to adapt when and how the hint display offers help, addressing both help abuse and avoidance.

For help-abuse, our primary hypothesis is that we can address all its three categories using our DD approach by restricting students to request hints only when they are taking longer than expected on a given objective. This addresses "Immediate Help," and partially addresses "Unneeded Help," since students cannot ask for hints until they have attempted an objective for some time. To address "High Frequency" hints abuse, we also added a 2-minute "cooldown" on hints, deriving the following rule: *At any time that a student is taking longer than expected on their current objective (based on the DD mechanism), they accumulate one hint every two minutes, which they can request whenever they want.* For

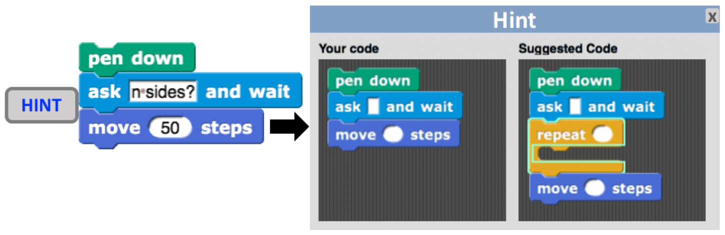


Figure 4.1: Example of a next-step hint given by iSnap. When the HINT button (left) is clicked, a hint is shown (right).

example, if the system detects student Alice is not progressing on Objective 1, then she will be able to ask for a hint. If she still does not progress, she will get another hint two minutes later. If she does progress, then she won't get another one until the system detects she has fallen behind on Objective 2. Note that we chose the 2-minute threshold based on our review of log data in Study 1, but we make no claim that is generalizable. It is preferable to be adjusted based on the instructor's choice.

For help avoidance, to address the first category "No Use", we wanted to make the hints more salient, and remind students to use them without taking away students' sense of control [Raz10]. To do so, whenever a student accumulates a hint, meaning they have fallen behind and may need help, the system pops up a *flashing* hint button (as shown in Figure 4.1), embedded in students' code, as a subtle reminder. The student can still choose when and whether to click on the button and view the hint, combining the advantages of proactive and on-demand hints displays [Raz10; Mur06]. For the second help avoidance category "Stopped Use", we did not focus in this work on improving hint quality. However, one possible way to address this behavior is to provide hints only when students need them, using the DD mechanism. As Murray et al. suggested, providing hints proactively (in our case it is just proactive reminders), may give students information at times they really need it [Mur06], perhaps motivating them to open it.

4.4.2 Method

Population: Because it is hard to assign students into different learning conditions in a real classroom (i.e. issues of fairness, different students seeing different interfaces), we recruited undergraduate students from an introductory engineering course at the same university for our study. As in our classroom population, these students had not taken any prior programming course and therefore, we assumed they had minimal experience. We also allowed students to take the study independently, online, as in a homework setting, through a web interface that led them through each procedure step. We compensated participants with a $10 gift card to encourage broader participation. Our population included 30 students (16 males; 14 females). All students were 18-20 years old and in their first year. Their intended majors were Engineering (n = 21), Biology (n = 5), CS (n = 2), and undecided (n = 2).

Programming Tasks: In this study, students were asked to solve two tasks, the first one with the hints and the second without the hints. Because we wanted to explore help-seeking under varying levels of difficulty, we designed three different tasks (A, B and C), each using similar programming concepts but having increasing difficulty. We assigned half of the students to take task A, then task B, and the other half took task B, then task C. Task A asked students to take an input number *n* and draw a polygon with *n* sides. In Task B, students were asked to take an input *n* and draw a strip of *n* triangles. Task C asked students to draw a series of *n* "daisy squares," a geometric design made of overlapping squares. Our goal with varying the difficulty was to create scenarios where students would need more or less help, allowing us to test the adaptive levels of help provided by the DD display. However, we did not control for tasks' difficulty in our analysis. We used data from a prior experiment using tasks A and B to train the timings for the DD display.

Procedure: Students first took a short presurvey and read through a tutorial about the block-based programming environment (iSnap) that focused on the programming concepts needed for the study tasks (e.g. loops, and drawing), using both text and short animations. We randomly assigned half of the students (n = 15) to have access to on-demand hints and the other half to use our new DD hint display (n = 15)[2]. Based on student's condition, the tutorial described how iSnap can offer help to them when needed. Students were then given up to 15 minutes to complete programming Task 1, and iSnap provided them with hints based on their condition. Afterwards, they had 15 minutes to complete Task 2, this time without hints, which we use as a measure of learning transfer. After each programming task, students took a post-task survey, where they were asked some questions on their experience and the hints provided (on Task 1).

4.4.3 Results and Discussion

RQ2a: *What is the impact of the DD display on students' help-seeking behavior?* To understand whether the DD display reduced unproductive help-seeking, two researchers, who

[2]We assigned the same number of students in each condition to each pair of tasks and so the tasks difficulty was equivalent for both conditions.

were blind to students' conditions, tagged each student with any form of help abuse or avoidance defined in the Taxonomy in Section 4.3.2. We report the full results in Table 4.2. In the on-demand condition, researchers tagged 7 (46.66%) as help avoiders, and 3 (20%) as help abusers, while in the DD display condition, researchers tagged 4 (26.66%) as help avoiders and 0% as help abusers. This suggests that the DD display cuts incidence of help avoidance in almost half and prevents help abuse altogether. For comparison, we combined these behaviors into a single attribute indicating unproductive (1) or productive (0) help use. A Fisher's exact test shows that the difference in unproductive help use across the two conditions is not significant ($p = 0.14$)[3] but had a moderate effect size (odds ratio = 3.92). The effect of our DD display may have been reduced by the tasks being too easy to necessitate much hint use: we found that 46.6% of students who asked for little or no hint still finished Task 1 successfully. The effect size suggests the DD display had a meaningful impact on students' help-seeking behavior, but our small sample size makes it unclear how well these results would generalize.

We then investigated how well our DD display encouraged students to ask for an appropriate number of hints, as defined by *its own algorithm*. We compared the number of hints suggested by the algorithm and the number of hints opened by students (for the on-demand condition, we simulated the DD algorithm on students' data, since they did not have the DD display). We found a much stronger correlation between the number of hints recommended by the algorithm and the number of hints opened by students in the DD condition ($r = 0.79$), than that in the on-demand condition ($r = 0.31$). This confirms that students' natural help-seeking behavior (in the on-demand condition) does not align with the algorithm's estimate of their help needs, but that the DD design succeeded in shaping students' help use. However, some students in the DD condition did still avoid help (27%). Manual investigation shows that these students were offered hints but chose not to click the hint button, sometimes after seeing a potentially confusing hint.

RQ2b: *How does the DD display impact students' learning?* We chose to measure students' performance on each task using the total time taken to complete the task, since 50% of students in both conditions successfully finished both tasks. If a student did not finish a given task, we recorded their time as 15 minutes (the maximum allowed for each task). We found students in the DD display condition took longer to finish Task 1 ($M = 10.99$; $Med = 12.33$) than those in the on-demand condition ($M = 8.98$; $Med = 9.42$), but a

[3]We report statistical tests, along with effect sizes, to help the reader better interpret our results. However, as this was pilot study with a small sample, these tests are only likely to detect large effects and should be interpreted cautiously.

Mann-Whitney U test[4] shows the difference was not significant, with a medium effect size ($p = 0.24$; Cohen's $d = -0.47$). Interestingly, we found that time taken to finish Task 2 (without hints) showed the reverse: students in the on-demand condition spent longer ($M = 10.55$; $Med = 12.23$) than those in the DD display condition ($M = 8.94$; $Med = 7.38$), but the difference is not significant ($p = 0.41$; Cohen's $d = 0.33$).

Based on these findings, we calculated how much students' performance *improved* from Task 1 (with hints) to Task 2 (without hints): Task 2 time - Task 1 time. We found that students in the DD display condition decreased their time from Task 1 to Task 2 ($M = -2.03$; $Med = -2.57$), while those in the on-demand condition increased their time ($M = 1.57$; $Med = 2.16$), and A Mann-Whitney U test shows the difference was significant, with a large effect size ($p = 0.02$; Cohen's $d = 0.85$). Together, these results suggest that on-demand display helps students more on the problem where they are available (short-term performance), but the DD display seems more likely to improve later performance without hints (i.e. learning). However, we are cautious in interpreting this result, since, as stated above, the overall impact of the DD display on students' Task 2 performance was positive but not significant.

One possible explanation for this difference would be that students in the on-demand condition simply asked for *more* hints on Task 1, improving their performance on Task 1 but not Task 2. However, we found that the students in the DD condition actually opened *more* hints ($M = 3.06$; $Med = 3$) than the on-demand condition ($M = 1.8$; $Med = 1$), and the difference was significant ($p = 0.04$; Cohen's $d = 0.59$). We also found that students *interacted* with hints differently in the two conditions. Students in DD display kept hints open for significantly more time ($M = 21.10$; $Med = 19$) than students in the on-demand condition ($M = 17.70$; $Med = 10$). A Mann-Whitney U test shows this difference is significant ($p < 0.01$; Cohen's $d = 0.23$). These results suggest that providing hints *only* at times when students are assumed to need them, allows them to process hints better [Mur06], which affected their performance in Task 2. In addition, these results confirmed our hypothesis that timing the hints on students' needs can eliminate help abuse categories. Similar to prior work [Mar19b], we found a significant *negative* correlation between the number of requested hints and students' grade in Task 2 in the on-demand condition ($r = -0.74$, $p < 0.01$), but not in the DD display condition ($r = 0.07$, $p = 0.78$), suggesting that DD display may mediate the relationship between help use and learning. Also, the ability of the DD display to tailor the amount of help based on students' needs confirms Wood's theory

[4]We used non-parametric tests because the data was not normally distributed.

Table 4.2: Percent of help-seeking categories in Study 2.

		On-Demand	DD
Abuse	Immediate help	0%	0%
	High frequency	0%	0%
	Unneeded help	20%	0%
Avoidance	Stopped Use	0%	13.33%
	No Use	46.66%	13.33%
	Productive help-seeking	33.33%	73.33%

of contingent tutoring, which emphasized that the tutor's help should be contingent upon the learner's needs to improve their learning [Woo99].

RQ2c: *What are students' perceptions of the helpfulness of both hint displays?* As Kardan et al., noted "Success of any adaptive support mechanism highly depends on the users' perception of its quality" [Kar15], and measuring students' perceptions is lacking in most help-seeking literature. It seems possible that students would not perceive the DD display to be as helpful as the on-demand one, since it does not give them as much control over when they receive hints. To measure this, we asked students in the Post Task 1 survey "On a scale from 1 to 10, how helpful was iSnap overall?". We found that students in the DD group rated iSnap at least as helpful ($M = 7.36$; $Med = 8$) as students in the on-demand condition ($Mean = 6.23$; $Med = 7.5$). This suggests no evidence that deploying DD design in the programming environment was perceived as less helpful than that with traditional on-demand help.

Moreover, when we asked students in the DD group to elaborate on why they gave this rating, students noted that hints just popped up at the right time "every time I got lost there was a hint." [P1]. Other students felt the hints were adaptive to their code "It seemed to let me try to figure things out for a minute then tell me when I couldn't get it." [P7]. In Post Task 2 survey, when we asked students if hints they received in Task 1 helped in Task 2, 92% of the students noted that the hints helped them to progress in Task 2 because "the code [in Task 2] required me to use similar blocks that I had learned in the past task." [P5], and others mentioned specific concepts they had learned from the hints, e.g. "when the hints from task 1 explained how I have to rotate my pen at certain times, it helped me do the same during task 2. [P4]". Also, we did not find any comment that showed that students needed more help, which suggests that using the DD display, students did not notice their indirect lack of control on requesting help at any time.

4.4.4 Limitations and Broader Implications

We note three limitations to our findings. First, some of our results were inconclusive due to the small sample size. Also, our tasks may have been too easy, since 26.6% of students completed Task 1 in 8 minutes (although they claimed having no programming experience). However, the goal of this work was to identify unproductive help-seeking behavior in programming and explore how the DD display can improve that behavior, and our results suggest potential for it to do so. Second, while the varying difficulty of the programming tasks might have affected the results, we found no significant difference in help-seeking or performance between students with different tasks. Lastly, we had no pre-test, and some of our results could be explained by our two conditions having different levels of prior knowledge, despite being randomly assigned.

Overall, in this paper, we have three main contributions: 1) a preliminary taxonomy of novices' unproductive programming help-seeking behavior, which inspired us to (2) develop a data-driven (DD) adaptive display for hints to improve students' help-seeking, and (3) we conducted a controlled pilot study to evaluate the impact of the DD display on students' help-seeking behavior. Our results show promise for how the DD display could improve students' help-seeking behavior and possibly their learning, without reducing the perceived usefulness of the system. Also, our DD display for hints can be deployed in other programming environments that log students data and provide any type of feedback. We do note that our DD design does require an autograder for testing student's code in real time to monitor their progress through the assignment, but these are commonly available in programming environments [Bal18; Joh16; Wan20]. Our system has implications for any classroom, whether or not automated help is available, since a similar system could also remind students to ask for help from instructors, peers, or message boards. In future work, we plan to replicate Study 2 on a large-scale population to verify the effect of DD display on performance and learning, and further explore the relation between help-seeking and different feedback types.

CHAPTER

5

STUDY 3: REPLICATION STUDY - THE IMPACT OF ENHANCED NEXT-STEP HINTS DURING BLOCK-BASED PROGRAMMING IN A CLASSROOM CONTEXT

5.1 Introduction

In my previous work I developed three design features to improve next-step hints: 1) textual explanations: to help students interpret the meaning of the suggested next-step hint, 2) self-explanation (SE) prompts: to engage students in active learning by prompting them to reflect on the hint, and 3) a help-display interface to improve students' help-seeking behavior. For simplicity, I call the next-step hints with the previously mentioned design choices "**enhanced next-step hints**". While my prior studies show evidence of how enhanced next-step hints can improve students' performance and learning [Mar20c; Mar19a], these studies have two important limitations. First, they were all laboratory studies, conducted on a small set of *short* exercises (5-7 lines of code). Some of these studies were also conducted on participants recruited from the Amazon Mechanical Turk platform, which is a different population from real students (as discussed in Chapter 3). Therefore, my findings may lack generalizability and may not accurately capture the real effects of enhanced next-step hints on students in an authentic classroom setting. For example, will enhanced next step hints be as effective for students in an authentic classroom as MTurkers on programming tasks? For short versus more complicated programming tasks? In a short-term laboratory context versus a long-term classroom context with many tasks? We do not know yet the answer, and it is therefore important to establish whether a given intervention works only in some settings, or across multiple, different settings.

This can be achieved through replication studies. In 2016, Ahadi et al. published a review on replication in computing education research discussing different reasons why replication studies are needed, such as to "verify earlier work", or to "elucidate which other factors may be relevant" or "rule out the effects of site-specific factors" [Aha16]. Following Ahadi et al.'s guidance, the goal of this study is to understand how robust my prior findings on data-driven enhanced next-step hints are, and whether they generalize to different populations and learning contexts. Specifically, this study is a *conceptual replication* [Hao19], in which the procedures are different (in this case different population, duration, context, tasks, etc.), but the research questions are the same (what is the impact of enhanced hints on students' outcomes). In addition, my prior work did not consider how the impact of enhanced next step hints may have been mediated by other factors, such as the programming problem difficulty or hint quality; which has been shown that such contextual factors can interact with automated support success at promoting learners' performance and learning [Shu08; Hau13]. This might also explain conflicting findings on the *overall* effectiveness of enhanced next-step hints.

To address this, in this work I conducted a classroom study in a real CS0 classroom setting across several weeks. For multiple programming tasks with varying difficulty, I developed next-step hints enhanced with the hint design features, developed in my prior studies: textual explanations, self-explanation (SE) prompts, and data-driven hint display. I evaluated the impact of the enhanced next-step hints on students' performance and learning in a classroom context. I also explored whether the problems' difficulty and hint quality affect the hints' effectiveness on students' outcomes.

This replication study seeks to answer the following research questions. Note that the first research question is parallel to that of Study 1 (Chapter 3), with a focus on an authentic classroom setting.

1. In classroom settings, what is the impact of enhanced next-step hints on students' programming performance, learning, and perceptions?
2. How does problem difficulty and hint quality mediate the effect of enhanced next-step hints on students' outcomes?

Overall, I hypothesize four hypotheses: First, students in the `Hints` group will complete more objectives (i.e. perform better) in the in-class and homework programming assignments than students in the `Control` group [H1 - Performance]. Second, the `Hints` group will finish their programming exercises in less time than those in the `Control` group [H2 - Time on Task]. Third, due to the presence of self-explanation prompts, I hypothesize that students in the `Hints` group will perform better in HW 2 (the assessment task) and in the post-test, showing an improvement in students' learning [H3 - Learning]. Fourth, I expect to find students appreciate having hints with textual explanations, more than having hints with both textual explanations and self-explanation prompts [H4 - Perceptions]. These hypotheses are all based on my findings in Study 1 and Study 2 (see Chapters 3, 4).

5.2 Methods

5.2.1 Population

I conducted this study in a CS0 classroom, in Fall 2020, which includes 74 undergraduate students, 62 of whom consented to our IRB-approved study. Participants of this course are undergraduate students who were novices with minimal programming experience, since they are required to have not taken any prior undergraduate or Advanced Placement (AP) programming course. In this class, 28.8% identified themselves as women, 69.5% as Men, and 0.01% as others. 69.5% identified themselves as White, 13.5% as Asians, 6.7% as Black/African American, 5% as Indians, and 3.4% as Native Americans. 61% of these students are less than 21 years old, 32.3% are between 21 and 24 years old, and 6.7% are older.

Table 5.1: Programming exercises used in this study.

Type	**Exercise name**	**Programming Concepts**					
		Funct-ions	**Loops**	**Variables**	**Conditions**	**Operators**	**User Interac-tion**
Lab 1	Polygon-Maker	x	x			x	
Lab 2	Squiral	x	x	x		x	
HW 1	Daisy Design	x	x	x	x	x	x
Lab 3	Guessing Game 1		x	x	x	x	x
Lab 4	Frogger	x	x	x	x	x	x
HW 2	BrickWall	x	x	x	x	x	

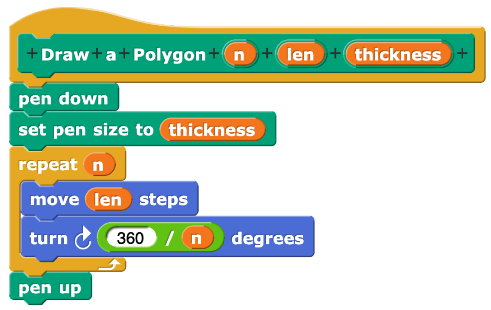
5.2.2 Programming Environment

During the first 4 weeks, students learn block-based programming using the iSnap programming environment [Pri17e], in which they do their classroom and homework exercises. Due to COVID-19, classrooms were held online via Zoom. Table 5.1 shows a description of the 6 programming exercises (4 in the classroom and 2 homeworks) used in this study. For both classroom and homework exercises, iSnap provided students with the enhanced next-step hints. As a part of the class requirements, students engaged in pair programming during *in-class* assignments, where each pair of students worked together on the same assignment attempt using a pair programming feature designed in iSnap. Using this feature, the instructor asked students to swap roles after every major step, to ensure that students were maintaining an appropriate pair programming practice. In Section 5.2.4 I fully discuss the pair programming feature, challenges, and how I dealt with it during data collection and data analysis.

5.2.3 Procedure

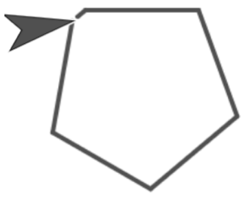
I employed a controlled study for 4 weeks, allowing for between-subjects comparisons [1]. Table 5.2 shows the study procedure. In the first week of the CS0 class, a researcher introduced

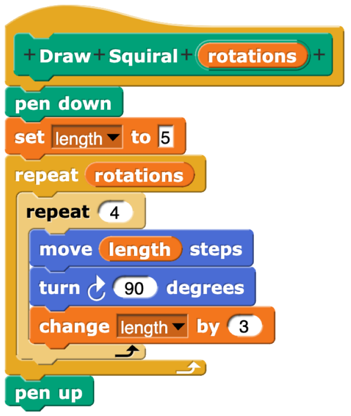
[1]For the remaining weeks I flipped conditions to ensure that all students receive hints at some point; however, I did not include analysis of this data since hints were not designed to be effective for more advanced programming tasks in Week 5-7.



```
+Draw+a+Polygon+(n)+(len)+(thickness)+
pen down
set pen size to (thickness)
repeat (n)
    move (len) steps
    turn ↻ ((360) / (n)) degrees
pen up
```

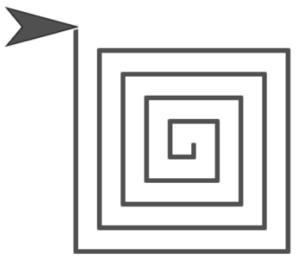
Polygon Maker

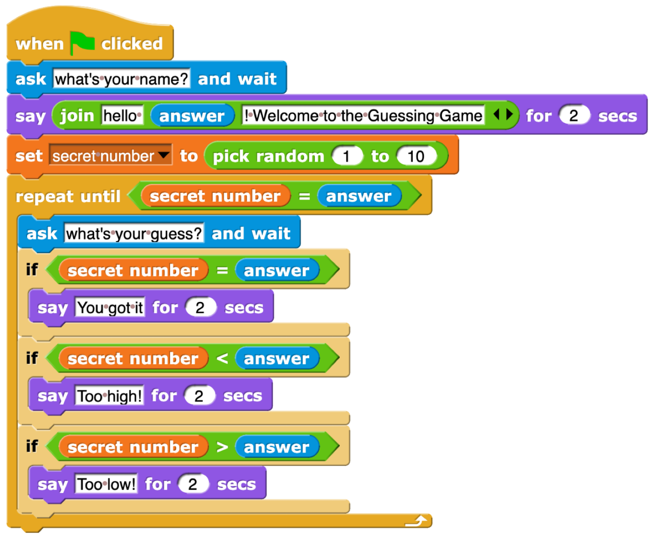




```
+Draw+Squiral+(rotations)+
pen down
set length to 5
repeat (rotations)
    repeat (4)
        move (length) steps
        turn ↻ (90) degrees
        change length by (3)
pen up
```

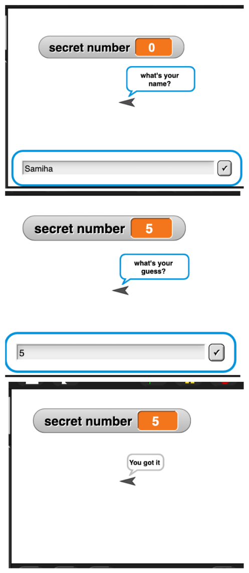
Squiral



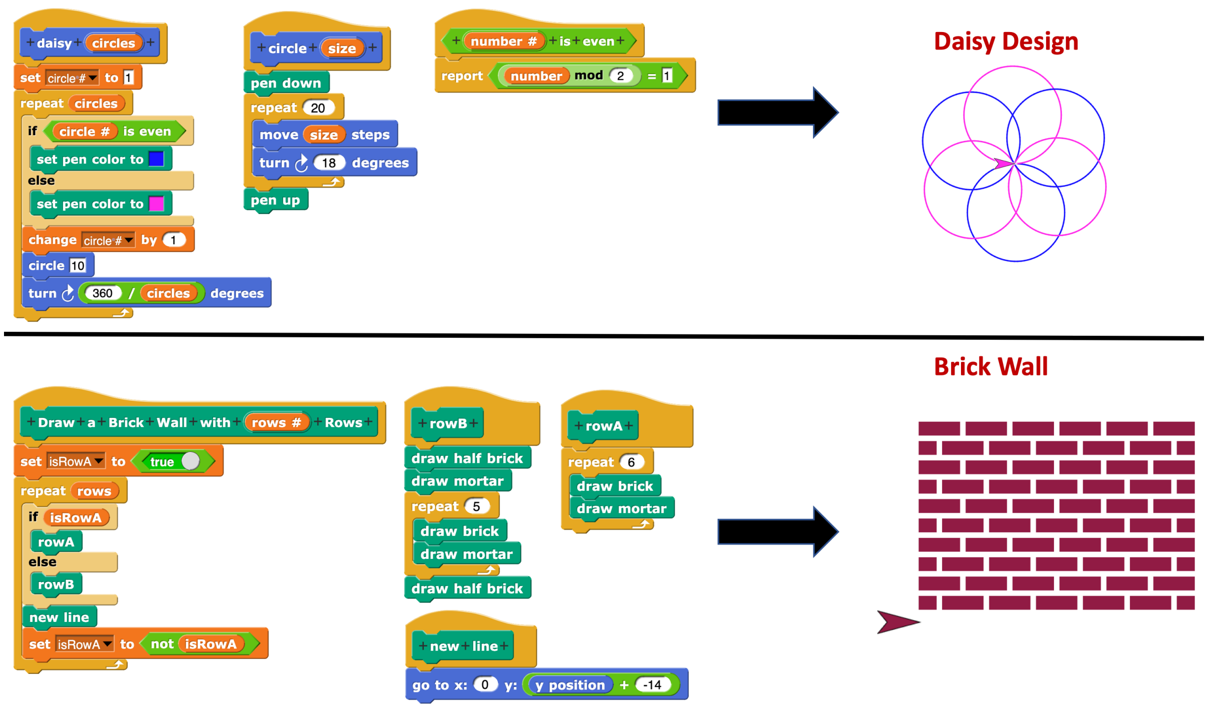


```
when ⚑ clicked
ask what's your name? and wait
say (join hello (answer) ! Welcome to the Guessing Game) for (2) secs
set secret number to (pick random (1) to (10))
repeat until <(secret number) = (answer)>
    ask what's your guess? and wait
    if <(secret number) = (answer)>
        say You got it for (2) secs
    if <(secret number) < (answer)>
        say Too high! for (2) secs
    if <(secret number) > (answer)>
        say Too low! for (2) secs
```

Guessing Game



```
secret number 0
what's your name?
Samiha
secret number 5
what's your guess?
5
secret number 5
You got it
```



```
+daisy+(circles)+
set circle # to 1
repeat (circles)
  if <(circle #) is even>
    set pen color to 🟦
  else
    set pen color to 🟪
  change circle # by 1
  circle 10
  turn ↻ (360 / (circles)) degrees
```

```
+circle+(size)+
pen down
repeat 20
  move (size) steps
  turn ↻ 18 degrees
pen up
```

```
+(number #)+is+even+
report <((number) mod 2) = 1>
```

```
+Draw+a+Brick+Wall+with+(rows #)+Rows+
set isRowA to <true>
repeat (rows)
  if <isRowA>
    rowA
  else
    rowB
  new line
  set isRowA to <not (isRowA)>
```

```
+rowB+
draw half brick
draw mortar
repeat 5
  draw brick
  draw mortar
draw half brick
```

```
+rowA+
repeat 6
  draw brick
  draw mortar
```

```
+new+line+
go to x: 0 y: ((y position) + -14)
```

Figure 5.1: One sample solution for each programming task (on the left), and their corresponding output (on the right).

the study to all students and offered them the opportunity to consent to participate in the research. When students first logged in the programming environment, iSnap randomly assigned students to one of the two conditions: the `Hints` condition where iSnap provided students with enhanced next-step hints, if needed, and the `Control` condition where iSnap provided no hints. Afterwards, to accommodate pair programming, students were randomly paired according to whether they have consented or not, and their condition, such that: for students who consented, I paired students together with similar conditions, and similarly for those who did not consent. However, if for any reason two students with different conditions were paired together, I chose to give them access to the `Hint` condition, and I discuss the potential implications of this in Section 5.5.

Figure 5.1 shows a solution example for each programming task with its corresponding expected output. In **week 1** students had their first *in-class* programming task called `PolygonMaker`, which asks students to create a procedure with 3 parameters: `n`, `len`, `thick`, to draw a polygon with *n* sides, each with with length *len*, and thickness *thick*. In **week 2**, students had three programming tasks: `Squiral` (in-lab), `Guessing Game` (in lab), and `Daisy` (homework). `Squiral` asks students to create a procedure that takes user

Table 5.2: Study Procedure.

Week no.		**Conditions**	
		Control	**Hints**
1		Consent form Survey 1	
	PolygonMaker	In lab 1 (no hints)	In lab 1 (hints)
2	**Squiral**	In lab 2 (no hints)	In lab 2 (hints)
	GG1	In lab 3 (no hints)	In lab 3 (hints)
	Daisy	HW 1 (no hints)	HW 1 (hints)
3	**Frogger**	In lab 4,5 (no hints)	In lab 4,5 (hints)
	BrickWall	HW 2 (no hints)	HW 2 (no hints)
4	Survey 2		
	Post-test 1		

input for `‘r’` and draws a spiral square with `‘r’` rotations. `Guessing Game` asks students to create an interactive game where the player tries to guess a randomly selected secret number by the computer. The game repetitively asks the player to guess the secret number until correct, and after each guess it tells the player if their guess was too high or too low, or congratulate them if the guess was correct. `Daisy` asks students to create a procedure that draws a daisy with a user-specified number `‘n’` of overlapping circular petals with alternate colors.

In **week 3**, students had two programming tasks, `Frogger` (in-lab), and `BrickWall` (homework). Frogger is an interactive game with several running Sprites: frogs, cars, lakes, and lily pads on the screen, where the player is required to get 1 frog on each lily pad without running into a car or sinking in the lake. For technical issues, the log data for `Frogger` was not retrieved correctly, and therefore, I excluded its analysis from this work. `BrickWall` asked students to draw a wall of bricks, with alternate rows of bricks using nested procedures, variables, conditions, and loops. At the end of **week 4**, the instructor gave students a post-test in the form of a multiple-choice quiz, adapted from [Wei15], followed by a post-survey to collect their perceptions about the usefulness of the hint features.

5.2.4 Analysis & Measures

In this study, I collected two sources of data: log data and surveys. This data was used to measure the following:

Performance

To measure performance, I used the existing teaching assistants' rubrics, such that each rubric item corresponds to an objective of a correct solution, where a successful completion to all objectives of a given exercise is equivalent to completing that exercise. Because teaching assistants (TAs) split grading students' submissions and deduct grades due to late submissions leading to inaccurate grades, I graded students' submissions, blind to condition, for all programming tasks, where a student score is the total number of completed objectives.

Analytical Approach: To analyze whether the enhanced next-step hints had an impact on students' performance in the in-class programming assignments, I used linear mixed effects (LME) models which "...are an extension of simple linear models to allow both fixed and random effects, and are particularly used when there is non-independence in the data, such as arises from a hierarchical structure" [Bru11]. Using LMEs is appropriate for this data due its repeated measures design in which observations (i.e. student scores in each programming task) were nested within students, since each student attempted more than one in-class programming assignment. Across both conditions in the in-class assignments, the mean programming performance is 88.4% (min = 0% ; SD = 20.83%; max = 100%). For further analysis of students' performance in any *individual* task, I used a Mann-Whitney U test to measure the statistical significance of differences between both conditions. A Mann-Whitney U test is appropriate in this case because this data does not satisfy the assumption of normality. I also report statistical tests, along with effect sizes, to provide more insights of the results; however, these results should be considered with caution since the sample size for each condition is less than 30.

Time on Task

For each assignment, I use students' log data to calculate the time students take until they submit the assignment to measure the effect of enhanced next-step hints on students' active time on task. Student *active time* is the total time from when they began to program to the time when they submitted their code. If a student was idle (i.e. making no code edits) for more than 5 minutes, I consider this as *idle* time and subtract it from their active time.

Analytical Approach: Similar to the performance analysis, I used linear mixed effects (LME) model to predict the effect of the enhanced next-step hints on students' overall active time taken to complete the in-class programming assignments. Across both conditions in the in-class assignments, the average student active time (in minutes) is 20 (min = 2 ; SD

= 14.52 ; max = 81). In addition, for any individual task, I used a Mann-Whitney U test to measure the statistical significance of differences in the active time across conditions.

Learning

I measured students' learning in two ways. First, I compared students' performance in the `BrickWall` homework assignment, where both conditions have no access to hints. The `BrickWall` assignment acts as an appropriate measure of assessment since it shares similar programming concepts with the previous programming assignments (e.g. loops, custom blocks, and variables), and therefore, one would expect that if students learned well during the previous programming assignments they can perform well on this task. Our second measure of learning is by comparing students' scores in a post-test which consists of 14 multiple-choice questions, where each question is graded as 1 if it is correct, and 0 otherwise.

Analytical Approach: For each of our learning measures (i.e. the assessment task, and the post-test), I used a Mann-Whitney U test to measure the statistical significance of differences between students' non-normal scores across the `Control` and the `Hints` groups. Across both conditions, the mean performance in `BrickWall` is 95.83% (min = 50% ; SD = 11.91%; max = 100%), and the mean score in the post-test is 87.23 (min = 61.54; SD = 11.13; max = 100).

Pair Programming Analysis

In this section, I describe how I analyzed students' log data during pair-programming. As a reminder, recall that pair programming was part of the classroom instruction. The instructor of the CS0 classroom developed a feature in the iSnap block-based programming environment that enabled pair-programming during in-class assignments, even during an online class. When two students work together, one of them acts as the "driver" and the other as the "navigator". Students can switch roles by pressing the "swap role" button in the programming environment. Clicking on this button will save the current code of the "driver", and load it on the "navigator's" screen. The iSnap logging feature logs this data, showing when students switch roles.

When analyzing students' log data while they were pair programming, there were three primary pair behaviors: First, some students worked separately on two different Snap projects, submitting two different projects at the end. Second, some students worked together throughout the programming assignment, submitting the same project at the end. Third, some students worked in pairs for some time but then finished separately, submitting two different projects at different end times, with some shared code and log data (e.g. students have the same first 60 edits, but one has an additional 30 edits, and the other has a *different* additional 10 edits)). This variance was expected, and occurred, for example, because some pairs finished their lab in-class, while others had to finish it (separately) for homework. Because of these discrepancies, when analyzing students' log data, I treated each student as a separate instance, where each student had a final submitted attempt, which I used to calculate students' performance, and this attempt has a single code trace (logged by iSnap), which I used to measure time on task. For students who worked in pairs, these submissions and traces were identical. For students who worked only part of the time together, their submissions were different, and their traces overlapped for some period of time.

It is worth emphasizing that for students who share part of their work together in a given programming assignment, this is how I calculated their time on task: Consider two students A and B who were working together on a given programming task. When Student A and Student B work at the same time, this time is added to Student A's and Student B's code trace. When either of them is working separately, that time is added only to the trace of the working student, and not their partner. Simply, Student A and Student B will have two different code traces that combined the time they worked together, in addition to the time they worked separately.

Hints Usage

To understand *how* hints may have impacted performance and learning, it is important to analyze how students used hints in the first place. To do so, I measured *hint usage*, as a measure of hint quality, in three ways: (1) how many hints students' have *opened* in each programming task. (2) Of these hints, how many were followed, i.e. whether students applied the hint's suggestion within a maximum of 10 code edits[2]. (3) how students answered SE prompts. To grade students' answers to open-ended SE prompts, I used a simplified rubric compared to prior work [Mar19a; Mar17a]. In particular, I graded SEs responses as either **informative** or **uninformative**. Informative SE prompt answer is when a student gave an informative *detailed* answer (such as: "The hint is showing me an easy way to get the proper degree change without any simple mathematical error"), or a *simple* answer without details

[2]I used a threshold of 10 edits after inspecting hundreds of students' data on how they used hints, since sometimes students need to apply several edits (i.e. deleting, or snapping blocks) before they apply a hint

(such as: “It helps pick the correct blocks to use.”). Uninformative SE prompt answer is when the student’s answer was nonsense (such as: “dfzdfgdzfg”) or unmeaningful (such as “Okk").

Students’ Perceptions

Using collected post surveys, I performed quantitative analysis to evaluate students’ ratings in Likert-scale questions about the usefulness of iSnap enhanced next-step hints, and which design choice they prefer. For open-ended questions, I reviewed all students’ responses to gain qualitative insights about students’ perspectives on hints during an authentic classroom experience. I identified positive and negative themes that indicated reasons why enhanced next step hints, or why specific design choices (e.g. SE prompts), were helpful or unhelpful, respectively.

5.3 Results

I analyzed students’ log data and survey data to investigate the impact of hints on students’ performance, time taken to complete tasks, and learning, as well as their perceptions of the hints. I analyzed in-class programming assignments together, since I am interested in the overall effect of hints on performance, time and learning (as discussed in Section 5.2.4 on the in-class tasks: `PolygonMaker`, `Squiral`, `Guessing Game`. I chose to analyze the `Daisy` homework task separately since this was a homework task and it lacks two factors of support: pair programming, and TA help during the in-class programming tasks, which may have mediated the effectiveness of hints. Table 5.3 shows descriptive statistics of students’ performance and time in all the programming tasks.

5.3.1 H1 - Performance

To evaluate the impact of hints on performance for the 3 in-lab assignments, I used a linear mixed-effects model (Model A), with *performance* as the dependent variable, *condition* and *task* as independent variables (fixed effects), and *student* as a random effect. As shown in Table 5.4, `condition` takes on a value of ‘1’ for the `Hints` group and ‘0’ for the `Control` group. ‘Task’ represents the categorical order of the programming assignments as 2 dummy variables (also known as indicator variables) to represent distinct categories [Skr09], where ‘0, 1’ used to represent `PolygonMaker`, ‘1, 0’ represents `Squiral`, and ‘0, 0’ represents `Guessing Game`. The model has a total of 160 observations which is the total number of

Table 5.3: The mean/medians of students' performance scores (in %) and time (in minutes) for each programming task across the *Hints* and *Control* group. The bold numbers indicate which group has a higher value.

	Performance		**Time**	
	Hints	Control	Hints	Control
PolygonMaker	**92 / 100**	76.85 / 100	10.11 / 10.17	**11.11 / 10**
Squiral	**92.24 / 100**	84.26 / 100	**27.83 / 24.87**	21.85 / 16
Guessing Game	91.85 / 100	**93.6 / 100**	23.67 / 19	**24.16 / 23.00**
Daisy	**90.38 / 100**	86.54 / 100	42.65 / 28.5	**62.12 / 44**
BrickWall	**97.83 / 100**	94 / 100	57.87 / 57	**61 / 54**

Table 5.4: Model A: Estimated coefficients (Standard Error) of linear mixed models with repeated measures predicting students' performance (i.e. grades) on programming tasks.

	Coeff (Std.Err)	**p - value**
Intercept	83.64 (2.66)	<0.001
Condition	7.76 (3.8)	0.043
Polygon	6.35 (3.34)	0.0601
Squiral	-0.45 (3.24)	0.88
Observations	160	

students' submissions in the three tasks. As shown in Table 5.4, Model A shows that only being in the `Hint` condition significantly improves students' performance ($p = 0.04$), such that students who had access to hints performed on average 7.76% points higher than that of the `Control` students across all the 3 in-class programming assignments. These results provide support for "H1 - performance", where having hints increased students' programming performance across in-class programming tasks.

For the `Daisy` homework task I found that students in the `Hints` condition performed better (M = 90.38; Med = 100; SD = 18.81) than students in the `Control` group (M = 86.5; Med = 100; SD = 22.62). However, a Mann-Whitney U test showed that this difference was not significant ($p = 0.64$; Cohen's $d = 0.2$), possibly due to a ceiling effect (the median score was 100).

5.3.2 H2 - Time on Task

To measure the time students spent on solving in-class programming tasks, I combined students' active time (as discussed in Section 5.2.4) on the in-class tasks: `PolygonMaker`, `Squiral`, `Guessing Game`. Similar to model A, I used a linear mixed-effects model con-

Table 5.5: Model B: Estimated coefficients (Standard Error) of linear mixed models with repeated measures predicting students' active time spent on programming tasks.

	Coeff (Std.Err)	**p - value**
Intercept	19.94 (1.77)	<0.001
Hint Condition	1.61 (2.5)	0.52
Polygon	-6.22 (2.14)	<0.01
Squiral	9.87 (2.21)	<0.001
Observations	160	

trolling for condition and task type (i.e. the independent variables) in Model B to predict students' active time (i.e. the dependent variable). As shown in Table 5.5, having access to hints (i.e. `Hint` condition) does not impact the time spent on in-class programming tasks ($p = 0.52$). However, we do see a significant effect of Task on time, meaning some assignments took longer to complete than others, but this is to be expected because of the varying difficulty of the programming assignments.

For the `Daisy` homework task I found that students in the `Hints` condition spent less time (M = 42.65; Med = 28.5; SD = 45.9) than that spent by students in the `Control` group (M = 62.11 ; Med = 44; SD = 46.22), and a Mann-Whitney U test showed that this difference is significant ($p = 0.01$; Cohen's $d = -0.42$). As a post-hoc analysis, I measured students' programming efficiency to investigate whether this decrease in time affected students' performance on the programming homework. I measured programming efficiency by calculating: students' performance score (in %) / active time spent (in minutes). I found that the `Hints` group programming efficiency is higher (M = 4.28; Med = 3.35; SD = 3.66) than that of the `Control` group (M = 2.11 ; Med = 1.92; SD = 1.45), and a Man-Whitney U test shows that this difference is significant ($p = 0.01$) with a high effect size (Cohen's $d = 0.82$). This suggests that students with hints achieved correct rubric items at over twice the efficiency of students without hints, on average. Together, these results show that enhanced next-step hints improved students' programming efficiency in a homework assignment, where no pair programming took place and with less help from the instructors and peers. Overall, these results provide partial support for "H2 - Time on Task", where having hints significantly decreases students' active time *only* in a homework task.

5.3.3 H3 - Learning

I first investigated learning by comparing students' performance on `BrickWall` (HW2) where all students did not have access to hints. I found that students in the `Hints` group performed better (M = 97.83%; Med = 100%; SD = 7.20%) than students in the `Control` group (M = 94%; Med = 100% ; SD = 14.93%). A Mann-Whitney U Test does not show a significant difference ($p = 0.42$), possibly due to a ceiling effect (the median score was 100). However, there was a medium effect size (Cohen's $d = 0.32$), with students in the `Hints` group losing less than half as many points on average. In terms of active time, I found that the `Hints` group completed the `BrickWall` task faster (M = 57.87 ; Med = 57 ; SD = 37.24) than the `Control` group (M = 61 ; Med = 54; SD = 29.74). While these results are inconclusive, it suggests that having hints in earlier tasks may improve programming efficiency in later tasks without hints.

I then compared students' scores in the post-test, where a total of 53 students took the post-test; 26 in the `Hints` group, and 27 in the `Control` group. I found little difference between the `Hints` group (M = 86.69%; Med = 88.46%; SD = 12.32%) and the `Control` group (M = 87.75%; Med = 92.31%; SD = 10.07%), and a Man-Whitney U test shows that this difference is not significant ($p = 1$; Cohen's $d = -0.01$). While this result does not show that enhanced next-step hints improve students' learning in a post-test, it does show that hints were not harmful for learning even if it provides students with a small part of the correct solution. Taken together, these results show inconclusive support for "H3 - Learning", such that having hints did not significantly improve students' learning in an assessment task or a post-test.

5.3.4 Hints and Self-Explanation Prompts Usage

Recall that our current version of iSnap programming environment provides students with hints using a data-driven algorithm that calculates whether a student is struggling or not, by comparing current student progress to progress of students in prior semesters [Mar20c]. As a result, if a student is marked as struggling, the system will pop-up a hint button, and it is up to the student if they want to open the hint. Therefore, not all students in the `Hints` group can receive hints, particularly if they are progressing, and the number of hints received by each student varies based on their progress [Mar20c]. To investigate students' hint usage, I first looked at the number of hints opened by students in the `Hints` condition. As shown in Table 5.6, the majority of students received at least one hint button in all tasks; and, 42.3% -

Table 5.6: A summary of students' hints usage and self-explanation prompts.

Task	**Number of students who received Hints**	**Students who opened at least one hint**	**Total hints opened**	**Number of hints Followed**	**Number of Informative SE prompts**
Polygon Maker	24/25	11/24 (45.83%)	26	23/26 (88.46%)	18/26 (69.23%)
Squiral	28/29	15/29 (51.7%)	82	64/82 (78.1%)	79/82 (96.34%)
GG1	26/27	11/26 (42.3%)	62	28/62 (45.16%)	61/62 (98.38%)
Daisy	21/26	10/21 (47.6%)	71	55/71 (77.46%)	62/71 (87.32%)

51.7% of these hints were opened. Compared to previous versions of iSnap, this range of hint usage is higher than what we found in previously published [Mar19a; Mar20c] and unpublished studies. While this result indicates a low probability of hint abuse, it may also indicate students' preference of being independent and not to overly use hints [Pri17c].

To investigate if the opened hints were useful and interpretable to students (i.e. a measure of hint quality), I measured the hints' follow rate (as explained in Section 5.2.4). As shown in Table 5.6, in three programming tasks: `PolygonMaker`, `Squiral`, and `Daisy`, the hint follow rate ranges between 77.46% - 88.46%. These results suggest that next-step hints with both textual explanations and SE prompts make hints interpretable and convincing to be followed. However, looking in the `Guessing Game` task, I found only 45.16% of hints were followed.

Digging further into hints provided in `Guessing Game`, I found 3 reasons for this low hint follow rate. First, as shown in Figure 5.1, the `Guessing Game` correct solution has multiple redundant blocks to be used, like "ask block" or "say block". As a result, often when hints suggest one of these code blocks that are already used in students' code, students may not understand why they need to use them more than once. Second, there are multiple ways to solve the `Guessing Game` assignment, and the hints were not always matching these multiple strategies. For example, sometimes students use "answer" block directly in the code (which is a block used to save user input to a question). However, a student may receive a hint that suggests assigning the "answer" block value into another variable, and then use that variable instead. Both strategies are correct; however, most students ignore this hint in that case. While that shows that hints were unfollowed, it also shows that students think about the hint, and reflect on it, which was clear in students' self-explanation prompts' answers, such as “I don’t think this is necessary because my code works!”. Third, some students attempted to break the `Guessing Game` assignment into several subparts, and then combine them at the end (often called a prototyping behavior – a common tinkering behavior in programming [Don19]). As a result, sometimes they get hints on a subpart that they are not working on at that moment, so they ignore the hint. For example one student said: “I did not reach this part yet". These three reasons indicate design issues in the hint generation system, not only in the iSnap system, but it can also occur in other block-based programming systems, which requires future improvements to such algorithms. Unlike `Guessing Game` assignment, when I investigated the hints opened by students in the other tasks (that have a high hint follow rate), I found most of these hints (>75%) are relevant to students’ code, and captured correct missing code blocks. This indicates that the hint usage is very relevant to the programming assignment it is generated for.

My last measure in hint usage is investigating students’ self-explanation prompts, as a way to understand how such SEs might have impacted students’ understanding to the hints. As explained in Section 5.2.4, I calculated how many SE answers were informative versus uninformative. Table 5.6 shows that 69.23% - 98.38% of students’ answers were informative, indicating students’ consideration to such prompts; whether by providing a *detailed informative* answer or a *simple* one. This is a possible factor of why hints positively impacted students’ performance, and sometimes learning, in this and prior work [Mar19b; Koe12]. Table 5.7 shows an example of SE prompts, and some of students’ informative and informative answers.

5.3.5 H4 - Students’ Perceptions

These results are focused on students in the `Hints` group who took the optional post-survey to present how students perceived hints. In total, 46 students took the survey and 3 of them mentioned they have never clicked on a hint button, leaving a total of 43 responses.

First, when students were asked which hint feature was most useful, I found that the majority of students (21, 48.8%) preferred having next step hints with textual explanations, 17 students (39.5%) preferred having only next-step hints, 3 students (7%) preferred having next-step hints with SE prompts, 2 students (4.6%) preferred having both textual explanations and SE prompts with next-step hints, and one student (2%) preferred having only textual explanations with SE prompts. This aligns with my prior work that shows that students prefered next-step hints with textual explanations [Mar19a; Mar19b], even if they do not read the hint explanations, and that prompting students to self-explain is

Table 5.7: Examples of SE prompts and students' answers to these prompts.

Task	SE Prompt	Informative Answer
PolygonMaker	Why do you think Snap recommended this hint?	i need to tell what thickness the pen needs to be
Squiral	How would you use this hint?	Insert length into the move steps block.
Squiral	Why do you think Snap recommended this hint?	I have been struggling with the correct blocks to use
Guessing Game	How does this hint help you think about how to solve the problem?	This did not help an error still occurs when I use this
Guessing Game	Why do you think Snap recommended this hint?	To understand how to say the total number of guesses to the user!
Daisy	Why do you think Snap recommended this hint?	The block is not present in my code!
Daisy	What is this hint trying to help you to understand or do?	Put set block in before the repeat block.
Daisy	What is this hint trying to help you to understand or do?	Have the number or petals adjust so the conditional function detects the shift

not prefered by most students [Mar19b]. Overall, this analysis supports "H4 - Perceptions", where students prefer having hints with textual explanations more than having hints with both textual explanations and self-explanation prompts.

Second, when students were asked to rate (on a scale from 1 to 5) the overall usefulness of hints, I found that 77% rated 3 or above. This shows that overall students in the `Hints` group found that hints were helpful. Below I report themes on why students' perceived hints as helpful or not, and why they preferred specific design choices.

For enhanced next-step hints, all students' reported that the enhanced next-step hints were overall helpful. Students noted that hints *"guide me [the student] to the correct code, which helped me to get along with my labs [S30]."*, and even more *"hints were the most helpful when dealing with new concepts that we [students] didn't learn in class [S5]."* This aligns well with the quantitative finding that the enhanced next-step hints improved students' overall performance during in-class assignments. In addition, one student noted that the hints were useful only at the early tasks, but not in complex tasks: *"after the first month or so they stopped being useful entirely, as the hints they gave were either too simple or not relevant to whatever more complex code we were learning [S8]."* This might reflect the lower performance I found in the `Guessing Game` assignment, which was mainly due to the lower hint quality as discussed in Section 5.3.4.

Some other students reported why *specific design choices* were helpful or not. For only next-step hints, students noted they were helpful because they are visual learners: *"if I see how something should be then I will remember that next time I encounter a similar problem [S33]"*, and that they are also easy to understand and fix errors: *"The pictures are the quickest way of gathering how a new block works [S17]"*, and they *"definitely helped me quickly fix my mistakes [S20]."* However, no student criticized the next-step hints by itself.

For next-step hints with textual explanations, most students noted that they are *"the easiest to understand because they tell you what to do word for word [S30]"*, and that they were specifically helpful because: *"they exactly told us how to improve our code when we were stuck on how to exactly use a certain block and what to use it for [S25]"*, which reveals the importance of our textual explanation *design* [Mar19a]. However, one student noted that the textual explanations needed to be more detailed: *"If they were more detailed and explanatory, they would definitely be helpful [S39]."*

Similar to prior work, I found conflicting findings about the usefulness of self-explanation prompts. On one hand, students liked SE prompts because *"the question improves my understanding and expression [S7]."*, which is the key intended outcome of the SE prompts. On the other hand, other students do not prefer SE prompts because they did not understand the hint in the first place *"The [SE prompt] isn't helpful, because often the part that compares code is confusing [S11]"*, or because *"the question at the end that was required to be answered got annoying. This is especially true when I needed to look at the same hint multiple times, and had to fill this section out every time [S3]."* I note that the negative themes about SE prompts are somewhat different than prior work because students noted design issues (as noted by S3 response), which we can fix in next deployments, or due to hint quality (as noted by S11). I argue that this opens new directions for improving the design of SE prompts, that can further reinforce their impact on students' outcomes in programming.

5.4 Discussion

5.4.1 RQ1

In classroom settings, what is the impact of the enhanced next-step hints on students' performance, learning and perceptions?

For **performance**, I found that students in the `Hints` group completed on average 7.4% more of in-class programming assignments than those in the `Control` group. These results align with my prior work [Mar19b], and suggest that hints accomplish their primary purpose of helping students progress [Ale16b]. Even though hints required students to take time to read and answer the SE prompts, I found that the `Hints` group performed better and submitted their code faster on the `Daisy` homework. Taking these results together - higher performance on in-Lab assignments, and faster completion on HWs, we might summarize this by saying that hints led to an increase in students' programming *efficiency*.

It is important to note that the enhanced next-step hints I evaluated were accompanied with textual explanations and SE prompts. My results suggest that these design choices are *effective* for creating impactful next-step hints, and my previous work [Mar19b] suggests that they may even be *necessary* to see longer-term impact. It might therefore be worth applying such design choices to other, somewhat similar, automated support (e.g. worked examples) to enhance its impact on students' performance in programming labs and homework tasks. At the end of this section, and in Section 8.1.4, I provide a more detailed discussion about the consistency of my findings across studies.

From a theoretical perspective, I explain how such design features improved the impact of hints on students' performance: First, *textual explanations* explain the hint in plain language, which might have added clarification and easier understanding to hints similar to effective human feedback, as suggested by prior work [Ger16; Van11]. Second, prompting students to *self-explain* hints might have encouraged them to stop and think about the hint, rather than quickly skimming it [Koe12]. This can encourage students to follow the hint, and therefore, bring their code closer to the correct solution. Third, *proactively* providing hints when a student struggles might trigger them to read the hint; since this is *when* they might be looking for help; and therefore, improve the effectiveness of hints [Woo99; Mur06]. In addition to these design choices, the enhanced hints were *immediate*, *specific* and *adaptive* to students' code. These are key effective feedback features that can help students to focus their attention on what they are supposed to, without increasing their cognitive load, and therefore they can easily add or adjust missing code leading to a higher performance [Cor01; Thu13].

For **learning**, I found that the *Hints* group performed better on *BrickWall* (i.e. the assessment task); however, the difference is not significant, suggesting inconclusive findings about whether hints improved learning. I also found no evidence that hints improved students' learning in a post-test. Overall, this result does not support our "H3- Learning" hypothesis that hints with textual explanations and SE prompts can improve students' learning. I suggest three possible reasons for this result. First, students did uniformly well on the assessment task (median grade 100%, and 87.5% of students fully completed the task), perhaps because it came after 3 weeks of programming practice, which might already be enough time for *all* students to succeed on this assignment, leading to a ceiling effect. Second, it might be that the low hints usage (as suggested by their varying quality) in the *Guessing Game* assignment affected students' help-seeking behavior that resulted in less learning outcomes. This interpretation is validated by a number of studies, in different domains, that found positive correlations between students' use of help and their learning outcomes [Woo99; Ale13]. Third, perhaps because this study is conducted in an authentic classroom context, where all students can get support from instructors or peers, and therefore the more assignments they solve the more knowledge they gain. As a result, unlike the short-term prior laboratory studies that found hints can improve learning [Cor01; Mar19b], these reasons might have diminished the difference in learning outcomes across both groups. Regardless, it seems clear that hints did not *impede* learning, which is notable, since prior work shows that hints, which give away part of the answer, can reduce student learning, for example if help abuse occurs [Ale16b; Ale13]. Since *having* hints helped students get unstuck, and complete work better and quicker, while learning at least as much as the *Control* group (for a future task and post-tests), this suggests that hints are still overall quite helpful to students and should be incorporated into introductory computing courses where possible.

For **students' perceptions,** the results confirm my hypothesis [H4-perceptions] that students appreciate having hints with textual explanations, more than having hints with self-explanation prompts. This aligns with prior work suggesting that novices may have difficulty with open-ended self-explanation prompts (e.g. being confusing, disturbing), since they may lack the domain knowledge to construct meaningful explanations [Roy05], or because they do not like doing the extra work [Mar19b]. While one can argue that hints with SE prompts can *not* be helpful if students do not prefer to answer the prompts, I argue that students' preferences do not always reflect what is best for learning. For example, Shih et al. found that students who spontaneously self-explain bottom-out hints come away with better learning results [Shi08b].

Last but not least, in terms of performance, time, and students' perceptions, the current study results are consistent with (i.e. replicated) my prior findings with a population of Mturk learners during an online one-hour study [Mar19b]. This suggests that the effect of hints is somewhat robust across multiple populations and learning contexts. However, I did not find a similar learning effect to what I found in my prior study. This can be due to the first reason discussed above – one-hour study can show an immediate effect on learning, versus 4-weeks study that might have faded this learning effect.

5.4.2 RQ2

How do problem type and hint usage mediate the effect of next-step hints on students' performance?

In terms of the effect of problem type (or difficulty), overall, I found that the hints improved overall student performance on in-lab and homework programming tasks; as shown in Model A in Section 5.3.1.

To further investigate if students' performance varies across programming assignments, I investigated students' performance on each task separately. I found that `Hint` condition performed better in all tasks except `Guessing Game` than the `Control` group, as shown in Table 5.3, but with different magnitudes. This suggests that the programming assignment complexity does not mediate the impact of hints, since I found that having hints improved students' performance in several tasks with varying complexity. However, all of these tasks were still relatively straightforward CS0 assignments, which could be accomplished in a few hours at most, and so my results do not speak to the impact of hints in more advanced programming courses.

I also looked at the `Guessing Game` student log data to investigate the reason for the

Hint group's lower performance. I found that hints provided in Guessing Game programming task were not of high quality, compared to the other tasks, as discussed in Section 5.3.4, which might have led to a decrease in the benefit of hints to students' performance. This shows that the usefulness of hints can be mediated by the *quality* of hints (as revealed by students' hint usage – opening and following hints). This agrees with prior work which suggests that "high-quality initial hints can encourage students to make more use of hints in the future [Pri18]". The variance in hint quality can arise due to many factors, such as the quality of *data* used to generate the hint, or the type of the programming task (i.e. has multiple possible solutions, or just a few ones), or the textual explanation used to describe the hint.

In terms of time, I found a consistent trend, where the Hints group finished faster than the Control group in all tasks except in Squiral, as shown in Table 5.3. Digging further in student log data and hint usage in Squiral, I found no clear reason why the Hints group spent more time in Squiral. This suggests that there might be other factors, unobservable in log data, that affect student programming time during the classroom. This is one of the challenges of running real-world classroom experiments: there are many classroom factors beyond control, such as the effectiveness of pair-programming groups, the distribution of instructor/TA help, etc., which cannot be observed in log data.

5.5 Limitations and Conclusion

This replication study suffers some limitations, mostly due to pair programming. First, some students who were in the Control group got paired with students from the Hints group and therefore got exposed to hints. While there were only a few instances of this occurring, it might have affected the study results. Second, since I evaluated students' data individually, some students who kept working in pairs through a given task might have submitted the same solution attempt, leading to an inevitable duplication in the data. This may have increased our ability to detect an impact of hints, since our unit of analysis was individual students instead of pairs. However, this was a necessary choice for analyzing pair programming data, where students also worked individually. Third, and last, Surveys were optional, and therefore the reported students' perceptions might not be a complete representation of all students. However, the survey results were very similar to my prior qualitative work on students' perceptions of next-step hints design choices [Mar19b].

In summary, in this chapter I presented a replication controlled study to assess the impact of enhanced next-step hints on students' performance, learning and perceptions of hints over several weeks. These are the contributions of this work: (1) a controlled study in an authentic classroom setting that shows that enhanced next-step hints improved students' programming performance in in-class assignments, and students' programming efficiency in homework assignments. (2) findings on how students' usage of hints is affected with the hints' quality which is in relevance with the programming task. (3) classroom challenges, such as the use of pair programming techniques and how researchers can analyze students' log data while pair programming. (4) evidence on how consistent, or different, is the impact of enhanced next-step hints across different populations and learning contexts, which reveals the potential reliability of our contributions and claims.

CHAPTER

6

STUDY 4: ADAPTIVE IMMEDIATE FEEDBACK CAN IMPROVE NOVICE PROGRAMMING ENGAGEMENT AND INTENTION TO PERSIST IN COMPUTER SCIENCE

This chapter was adapted from: Marwan, Samiha et al. "Adaptive Immediate Feedback Can Improve Novice Programming Engagement and Intention to Persist in Computer Science". In Proceedings of the 2020 ACM Conference on International Computing Education Research (pp. 194-203). The original text has been modified as follows: The Related Work section has been incorporated into Chapter 2.

6.1 Abstract

Prior work suggests that novice programmers are greatly impacted by the feedback provided by their programming environments. While some research has examined the impact of feedback on student learning in programming, there is no work (to our knowledge) that examines the impact of adaptive immediate feedback within programming environments on students' desire to persist in computer science (CS). In this paper, we integrate an adaptive immediate feedback (AIF) system into a block-based programming environment. Our AIF system is novel because it provides personalized positive and corrective feedback to students in real time as they work. In a controlled pilot study with novice high-school programmers, we show that our AIF system significantly increased students' intentions to persist in CS, and that students using AIF had greater engagement (measured by their lower idle time) compared to students in the control condition. Further, we found evidence that the AIF system may improve student learning, as measured by student performance in a subsequent task without AIF. In interviews, students found the system fun and helpful, and reported feeling more focused and engaged. We hope this paper spurs more research on adaptive immediate feedback and the impact of programming environments on students' intentions to persist in CS.

6.2 Introduction

Effective feedback is an essential element of student learning [Shu08; Cor89] and motivation [Nar04], especially in the domain of programming [Cor01; Gus18; Mar19b]. When programming, students primarily receive feedback from their programming environment (e.g., compiler error messages). Prior work has primarily focused on how such feedback can be used to improve students' cognitive outcomes, such as performance or learning [Bec18; Gus18; Mar19b]. However, less work has explored how such feedback can improve students' affective outcomes, such as engagement and intention to persist in computer science (CS). These outcomes are equally important because we are facing a shortage of people with computational knowledge and programming skills [Den15], which will not be addressed–no matter how much students learn about computing in introductory courses–unless more students choose to pursue computing education and careers.

It is also important to study feedback in programming environments because prior work shows that it can sometimes be frustrating, confusing, and difficult to interpret [Bec18;

Pri17d; Pri17c]. In particular, there is a need for further research on how programming feedback can be designed to create positive, motivating, and engaging programming experiences for novices, while still promoting performance and learning. Creating these positive experiences (including enjoyment and feelings of ability) are particularly important because they have a profound impact on students' intention to persist in computing [Lew11].

In this paper, we explore the effects of a novel adaptive immediate feedback (AIF) system on novice programming students. We designed the AIF system to augment a block-based programming environment with feedback aligned with Scheeler et al's guidance that feedback should be immediate, specific, positive, and corrective [Sch04]. Thus, our AIF provides real-time feedback adapted to each individual student's accomplishments on their performance on a specific open-ended programming task. Since our AIF system is built on data from previous student solutions to the same task, it allows students to approach problem solving in their own way. Given the beneficial impact of feedback on learning [Thu13], we hypothesize that our AIF system will improve student performance and learning. We also hypothesize that our AIF system will improve the coding experience of novice programmers, making it more likely that they will want to persist in CS. This is especially important given the aforementioned dearth of workers with computing skills, and the fact that many students with sufficient CS ability choose not to major in CS [Kat06].

We performed a controlled pilot study with 25 high school students, during 2 summer camps, to investigate our primary research question: What impact does adaptive immediate feedback (AIF) have on students' perceptions, engagement, performance, learning, and intentions to persist in CS? In interviews, students found AIF features to be engaging, stating that it was fun, encouraging, and motivating. Our quantitative results show that in comparison to the control group, our AIF system increased students' intentions to persist in CS, and that students' who received the AIF were significantly more engaged with the programming environment (as measured by reduced idle time during programming). Our results also suggest that the AIF system improved student performance by reducing idle time, and that the AIF system may increase novice students' learning, as measured by AIF students' performance in a future task with no AIF. In sum, the **contributions** of this work are: (1) a novel adaptive immediate feedback system, and (2) a controlled study that suggests that programming environments with adaptive immediate feedback can increase student engagement and intention to persist in CS.

6.3 Adaptive Immediate Feedback (AIF) System

Our adaptive immediate feedback (AIF) system was designed to improve the feedback students receive as they are learning to code in open-ended programming tasks (e.g. PolygonMaker and DaisyDesign, described in more detail in Section 7.4.2). Our AIF system continuously and adaptively confirms when students complete (or break) meaningful objectives that comprise a larger programming task. Importantly, a student can complete AIF objectives without having fully functional or complete code. This allows us to offer positive and corrective feedback that is immediate and specific. In addition, our AIF system includes pop-up messages tailored to our student population, since personalization is key in effective human tutoring dialogs [Boy08; DE09] and has been shown to improve novices' learning [Lee11; Mor04].

More specifically, our AIF system was developed to support exercises in *iSnap* block-based programming environment [Pri17e]. Our AIF system consists of 3 main components to achieve real-time adaptive feedback: objective detectors, a progress panel, and pop-up messages. The *objective detectors* are a set of continuous autograders focused on positive feedback, that continuously check student code in real time to determine which objective students are working on and whether they have correctly achieved it or not. The *progress panel* is updated by the continuous objective detectors to color each task objective according to whether they are complete (green), not started (grey), or broken (red), since prior research suggests that students who were uncertain often delete their correct code [Don19]. The *pop-up messages* leverage the objective detectors to provide achievement pop-ups when objectives are completed, and motivational pop-ups when a student has not achieved any objectives within the last few minutes. These pop-ups promote confidence by praising both accomplishment and perseverance, which may increase students' persistence [Cec11]. We strove to make the AIF system engaging and joyful, which may increase students' motivation and persistence [Jen01].

To develop our AIF system, we first developed task-specific objective detectors, which can be thought of as continuous real-time autograders, for each programming task used in our study. Unlike common autograders which are based on instructors' test cases, our objective detectors are hand-authored to encompass a large variety of previous students' correct solutions matching various students' mindsets. To do so, two researchers, who are experts in block-based programming languages and one of them is a block-based programming instructor, divided each task into a set of 4-5 objectives that described features of a correct solution, similar to the process used by Zhi et al. [Zhi18]. Then, for each task,

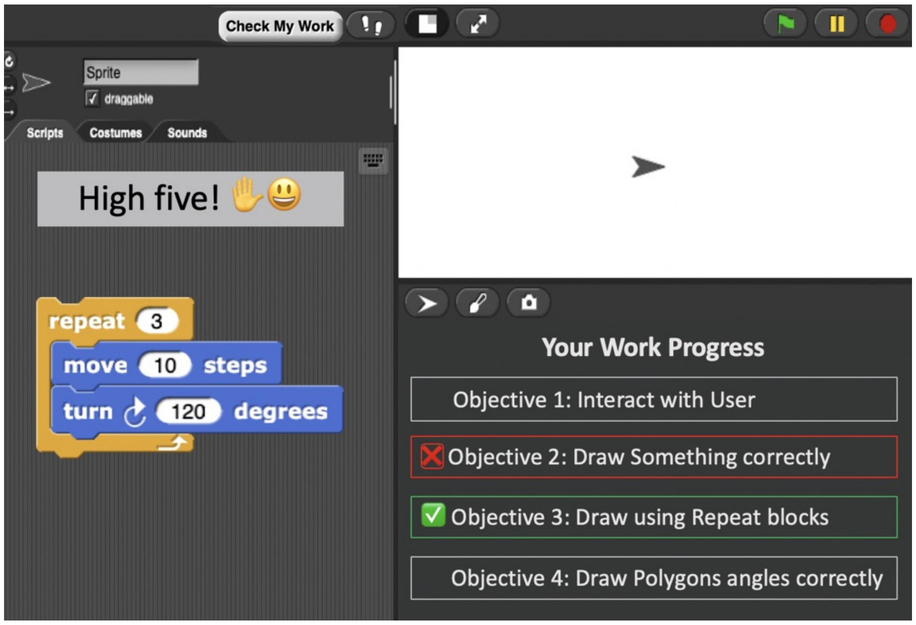

Check My Work
Sprite
draggable
Scripts
Costumes
Sounds
High five! ✋😃
repeat 3
move 10 steps
turn 120 degrees
Your Work Progress
Objective 1: Interact with User
Objective 2: Draw Something correctly
Objective 3: Draw using Repeat blocks
Objective 4: Draw Polygons angles correctly

Figure 6.1: Adaptive Immediate Feedback (AIF) system with pop-up message (top) and progress dialog (bottom right).

we transferred prior students' solutions into abstract syntax trees (AST). Using these ASTs, we detected different patterns that resemble a complete correct objective, and accordingly, we developed objective detectors to detect the completion for each objective. Finally, we tested and enhanced the accuracy of the objective detectors by manually verifying their performance and refining them until they correctly identified objective completion on 100 programs written by prior students.

Based on the objective detectors, we designed the progress panel to show the list of objectives with colored progress indicators, as shown in the bottom right of Figure 6.1. Initially, all the objectives are deactivated and grey. Then, while students are programming, the progress panel adaptively changes its objectives' colors based on students' progress detected by our objective detectors. Once an objective is completed, it becomes green, but if it is broken, it changes to red.

We then designed personalized AIF pop-up messages. We asked several high school students to collaboratively construct messages for a friend to (1) praise achievement when they finish an objective, or (2) provide motivation when they are struggling or lose progress. The final messages shown in Table 6.1 include emojis added to increase positive affect [Der08; Rio17]. In real time, our AIF system selects a contextualized pop-up message based on students' code and actions, detected by our objective detectors. The pop-up messages

Table 6.1: Examples of Pop-up Messages in the AIF System.

State	message
< 1/2 objectives complete	- You are legit amazing!! 🤩
> 1/2 objectives complete	- You're on fire! 🔥🔥🤩
All objectives are done	- High FIVE!! 🙌, you DID IT!!
Fixed broken objective	- Yay!!!! 🎉it's fixed 💪😎!!
Struggle/Idle , <half done	- Yeat it till you beat it!! 👊🤓
Struggle/Idle, >half done	- You are doing great so far!! ✌✌

provided immediate adaptive feedback, such as "Woo, one more to go!!" for a student with just one objective left, or "Good job, you FIXED it!! ;)" when a student corrected a broken objective. To praise perseverance, AIF pop-ups are also shown based on time, either after some idle time, or if a student takes longer than usual on an objective based on previous student data. It may be especially important to provide affective support to students who may be struggling. For example, if a student stopped editing for more than 2 minutes[1] and they are half-way through the task, one motivational pop-up message is "Keep up the great work !!".

Together, our novel AIF system is the first such system to include continuous real-time autograding through our objective detectors, and is the first to use such detectors to show students a progress panel for task completion in open-ended programming tasks. Further, our AIF pop-up message system is the first such system that provides both immediate, achievement-based feedback as well as adaptive encouragement for students to persist.

6.3.1 Jo's Experience with the AIF System

To illustrate a student's experience with the AIF system and make its features more concrete, we describe the observed experience of Jo, a high school student who participated in our study, as described in detail in Section 7.4. On their first task to create a program to draw a polygon (PoygonMaker), Jo spent 11 minutes, requested help from the teacher, and received

[1]We used a threshold of 2 minutes based on instructors' feedback on students' programming behavior in camps.

3 motivational and 2 achievement pop-up messages. As is common with novices new to the block-based programming environment, Jo initially spent 3 minutes interacting with irrelevant blocks. Jo then received a positive pop-up message, “Yeet it till you beat it 👊🤓”. Over the next few minutes, Jo added 3 correct blocks and received a few similar encouraging pop-up messages. Jo achieved the first objective, where AIF marked it as complete and showed the achievement pop-up “You are on fire! 🔥 🔥.” Jo was clearly engaged, achieving 2 more objectives in the next minute. Over the next 3 minutes, Jo seemed to be confused or lost, repetitively running the code with different inputs. After receiving the motivational pop-up “You’re killing it! 👊😎,” Jo reacted out loud by saying “that’s cool”. One minute later, Jo completed the 4th objective and echoed the pop-up “Your skills are outta this world 😎!!, you DID it!! 🎉,”’ saying, “Yay, I did it.” Jo’s positive reactions, especially to the pop-up messages, and repeated re-engagement with the task, indicate that AIF helped this student stay engaged and motivated. This evidence of engagement aligns with prior work that measures students’ engagement with a programming interface by collecting learners’ emotions during programming [Mar17b].

Jo’s next task is to draw a DaisyDesign, which is more complex, and our adaptive immediate feedback seemed to help Jo overcome difficulty and maintain focus. AIF helped Jo stay on task by providing a motivational pop-up after 3 minutes of unproductive work. In the next minute, Jo completed the first two objectives. AIF updated Jo’s progress and gave an achievement pop-up, “You’re the G.O.A.T[2] ✓🤩”. Jo then did another edit, and AIF marked a previously-completed objective in red – demonstrating its immediate corrective feedback. Jo immediately asked for help and fixed the broken objective. After receiving the achievement pop-up “Yeet, gottem!! 👏🏼,” Jo echoed it out loud. Jo spent the next 13 minutes working on the 3rd objective with help from the teacher and peers, and 3 motivational AIF pop-ups. While working on the 4th and final objective over the next 5 minutes, Jo broke the other three objectives many times, but noticed the progress panel and restored them immediately. Finally, Jo finished the DaisyDesign task, saying “the pop-up messages are the best.” This example from a real student’s experience illustrates how we accomplished our goals to improve engagement (e.g. maintaining focus on important objectives), student perceptions (e.g. stating “that’s cool”), performance (e.g. understanding when objectives were completed), and programming behaviors (e.g. correcting broken objectives).

[2]G.O.A.T., an expression suggested by teenagers, stands for Greatest of All Time.

6.4 Methods

We conducted a controlled, pilot study during two introductory CS summer camps for high school students. Our primary **research question** is: What impact does our adaptive immediate feedback (AIF) system have on novice programmers? Specifically, we hypothesized that the AIF system would be positively perceived by students (H1-qual) and that the AIF system would increase: students' intentions to persist in CS (H2-persist), student' engagement (H3-idle), programming performance (H4-perf), and learning (H5-learning). We investigated these hypotheses using data gleaned from interviews, system logs, and surveys.

6.4.1 Participants

Participants were recruited from two introductory CS summer camps for high school students. This constituted an ideal population for our study, as these students had little to no prior programming experience or CS courses, allowing us to test the impact of the AIF system on students who were still learning the fundamentals of coding and who had not yet chosen a college major. Both camps took place on the same day at a research university in the United States.

We combined the camp populations for analysis, since the camps used the same curriculum for the same age range. Study procedures were identical across camps. The two camps consisted of one camp with 14 participants, (7 female, 6 male, and 1 who preferred not to specify their gender) and an all-female camp with 12 participants. Across camps, the mean age was 14, and 14 students identified as White, 7 as Black or African American, 1 as Native American or American Indian, 2 as Asian, and 1 as Other. None of the students had completed any prior, formal CS classes. Our analyses only include data from the twenty-five participants who assented–and whose parents consented–to this IRB-approved study.

6.4.2 Procedure

We used an experimental, controlled pre-post study design, wherein we randomly assigned 12 students to the experimental group *Exp* (who used the AIF system), and 13 to the *Control* group (who used the block-based programming environment without AIF). The pre-post measures include a survey on their attitudes towards their intentions to persist in CS and a multiple choice test to assess basic programming knowledge. The teacher was unaffiliated with this study and did not know any of the hypotheses or study details, including condition assignments for students. The teacher led an introduction to block-based programming, and explained user input, drawing, and loops. Next, all students took the pre-survey and pretest.

In the experimental phase of the study, students were asked to complete 2 consecutive programming tasks (1: PolygonMaker and 2: DaisyDesign). Task1, PolygonMaker, asks students to draw any polygon given its number of sides from the user. Task2, DaisyDesign, asks students to draw a geometric design called “Daisy” which is a sequence of overlapping `n` circles, where `n` is a user input. Both tasks required drawing shapes, using loops, and asking users to enter parameters, but the DaisyDesign task was more challenging. Each task consisted of 4 objectives, for a total of 8 objectives that a student could complete in the experimental phase of the study. Students in the experimental group completed these tasks with the AIF system, while students in the control group completed the same task without the AIF system. All students were allowed to ask for help from the teacher, and could request up to 5 hints from the system[3]. In sum, there were eight objectives (as described in Section 6.3) that students could complete in this phase of the experiment (the experimental phase). We measured a student’s programming performance based on their ability to complete these eight objectives (see Section 7.4.3, below for more details).

After each student reported completing both tasks[4], teachers directed students to take the post-survey and post-test. Two researchers then conducted semi-structured 3-4 minute interviews with each student. During the interviews with *Exp* students, researchers showed students each AIF feature and asked what made it more or less helpful, and whether they trusted it. In addition, the researchers asked students’ opinions on the AIF design and how it could be improved.

Finally, all students were given 45 minutes to do a third, similar, but much more challenging, programming task (DrawFence) with 5 objectives *without* access to hints or AIF. Learning was measured based on a student’s ability to complete these five objectives (see subsection 7.4.3, below for more details).

[3]Our AIF extended the open-source (Blinded for review) system that provides automatic on-demand hints that were available for all students in both conditions.

[4]While all students reported completing both tasks, we found some students did not finish the tasks after looking at their log data.

6.4.3 Measures

Pretest ability - Initial computing ability was measured using an adapted version of Weintrop, et al.'s commutative assessment [Wei15] with 7 multiple-choice questions asking students to predict the outputs for several short programs. Across both conditions, the mean pre-test score was 4.44 (SD = 2.39; min = 0; max = 7).

Engagement - Engagement was measured using the percent of programming time that students' spent idle (i.e. not engaged) on tasks 1 and 2. While surveys are often used to measure learners' engagement, these self-report measures are not always accurate, and our fine-grained programming logs give us more detailed insight into the exact time students were, and were not, engaged with programming. To calculate percent idle time, we defined idle time as a period of 3 or more minutes[5] that a student spent without making edits or interacting with the programming environment, and divided this time by the total time a student spent programming. A student's total programming time was measured from when they began programming to task completion, or the end of the programming session if it was incomplete. While we acknowledge that some "idle" time *may* have been spent productively (e.g., by discussing the assignment with the teacher or peers), we observed this very rarely (despite Jo's frequent help from friends and the teacher). Across both conditions, the mean percent idle time on tasks 1 and 2 was 13.9% (Mean = 0.139; SD = 0.188; min = 0; max = 0.625).

Programming Performance - Programming performance was measured by objective completion during the experimental phase of the study (i.e., the first eight objectives in which the experimental group used the AIF system). Each observed instance of programming performance was binary (e.g., the object was completed [value of '1'] or the object was not completed [value of '0']). This led to a repeated measures design, in which observations (i.e., whether an object was or was not completed) were nested within participants, as each participant attempted 8 objectives (in sum, 200 observations were nested within 25 participants). We explain our analytical approach in more detail below. Across both conditions, the mean programming performance was 0.870 (min = 0 ; max = 1).

Learning - Learning was measured by objective completion during the last phase (the learning phase) of the study, on the last five objectives in which neither group used the AIF system, because we assumed that students who learned more would perform better on this DrawFence task. Each observed instance of programming performance was binary. This led

[5]We choose this 3-minute cutoff based on our analysis of prior student programming log data on the same tasks.

to a repeated measures design, in which observations (e.g., whether an objective was or was not completed) were nested within participants, as each participant attempted 5 objectives. In sum, 125 observations were nested within 25 participants. Across both conditions, the mean learning score was 0.600 (min = 0; max = 1), meaning that, on average, students completed about 3 of the 5 objectives.

Intention to Persist - CS persistence intentions were measured using the pre- and post-surveys using 7-point Likert scales adapted from a survey by Correll at al. [Cor04]. Students were asked to state how likely they were to: 1) take a programming course in the future, 2) Minor in CS, 3) Major in CS, 4) Apply to graduate programs in CS, and 5) Apply for high-paying jobs requiring high levels of Computer Science ability. We averaged these measures together for a CS persistence index with a high alpha (a = 0.85) with a mean of 24.8 (SD = 5.15; min = 13; max = 32).

While we could not measure actual persistence in CS, intentions to persist are a good proxy, given that research finds that they are predictive of actual persistence in STEM fields and "...hundreds of research efforts occurring [since the late 1960s] support the contention that intention is the 'best' predictor of future behavior" [Mer12].

6.4.4 Analytical Approach

Our analytical approach was informed by our small sample size (caused by the low number of students available for study recruitment) in two ways. First, to control for pre-existing differences between students, we control for a student's pretest performance in all of our models, as random assignment in a small sample may not be enough to ensure roughly equal levels of ability in both groups. Second, to maximize our statistical power, we treat each programming performance item (objective complete or incomplete) as our unit of analysis using linear mixed effects models. This means 200 observed objective scores (8 per student) are nested within 25 students. These models, "...are an extension of simple linear models to allow both fixed and random effects, and are particularly used when there is non independence in the data, such as arises from a hierarchical structure" [Bru11]. This is appropriate to use because our data have a nested structure, as observations (e.g., objective completion, CS persistence intentions) are nested within students. Thus, observations are not independent, given that each participant contributed numerous observations. A mixed model allows us to account for the lack of independence between observations while still taking advantage of the statistical power provided by having repeated measures. It also allows us to estimate a random effect for each student, meaning that the model more effectively controls for idiosyncratic participant differences, such as differences in incoming programming ability between participants.

We use a specific type of linear mixed effects model, a linear probability model (LPM) with mixed effects[6] to predict the binary outcomes of programming performance and learning. While logistic models are typically used to predict binary outcomes, we used a LPM with mixed effects because the interpretation of coefficients is more intuitive [VH15a]. This has led many researchers to suggest using LPMs [Ang08], especially because they are typically as good as logistic models at predicting dichotomous variables, and their p-values are highly correlated [Hel09].

6.5 Results

We first use interviews to investigate (H1-qual), that the AIF system would be positively perceived by students, and then use the student surveys to determine the impact of the AIF system on intentions to persist in CS (H2-persist). Next, we analyze student log files to investigate the impact of the AIF system on student engagement (H3-idle), programming performance (H4-perf), and learning (H5-learning).

6.5.1 H1-qual: AIF Perceptions

To investigate student perceptions of the AIF system, we transcribed interviews of all 12 students in the *Exp* group. Afterwards, we followed a 6-step thematic analysis, described in [Mag17; Bra06], to identify positive and negative themes for the AIF pop-ups and progress panel features. Two of the present authors start by (1) getting familiar with the data, then (2) generate initial codes (for each AIF feature), and then start (3) looking for dominating themes. Afterwards, one author (4) reviews the themes, and (5) refines them to focus mainly on the positive and negative aspects of each AIF feature. We then (6) combine these findings in the following summary. The main theme for each feature is whether it is helpful or not. We select quotes that represent typical positive and negative helpfulness comments from student participants labeled with s1-s12.

Pop-up messages: When asked what made pop-up messages helpful or less helpful, 10 out of 12 students agreed that pop-up messages were helpful and elaborated on why. Most students found them "engaging" [s2], "funny" [s1, s2], "encouraging" [s7], and "motivating" [s1, s5, s8]. For example, s1 stated, "it is better than just saying 'correct' because it gets

[6]A LPM is simply a linear model used to predict a binary outcome.

you more into it," and s5 said, "it kept me focused and motivated, I was like 'yea I did it'." One student, s7, noted that messages encouraged perseverance by removing uncertainty: "especially when you don't know what you were doing, it tells that you are doing it, so it let you continue and keep going." Two students noted that pop-ups were helpful by "keeping me on track" [s3, s4]. When asked what made pop-up messages *less* helpful, two students stated that "they were not really helpful" [s4, s3], and one student s7 suggested "a toggle to [turn them] on or off". Overall, the majority of students (83%) found the pop-ups helpful and engaging.

Progress Panel: When we asked students "what about the progress panel makes it helpful or less helpful?", all students found it helpful and requested to have it in future tasks. Students said the progress panel not only helped them to keep track of their progress, but also motivated them:"it told you what you completed so it kinda gives me motivation" [s8]. In addition, students appreciated the change in colors of each objective "because you can see how much you have to do, or if you took off something [a block] and you thought it was wrong and it turns red then that means you were actually right" [s7]. When students were asked what about the progress panel makes it *less* helpful, one student incorrectly noted, "I think there was [only] one way to complete these objectives" [s6]. Students overwhelmingly found the AIF progress panel to be beneficial, especially in understanding when an objective was complete or broken, supporting Scheeler's suggestion that feedback should be specific, immediate and positive or corrective.

This thematic analysis **supports H1-qual**, that students would positively perceive the AIF system. In addition, these results provide insights on how to design personalized adaptive, immediate feedback that can engage and motivate novice high school students in their first programming experiences.

6.5.2 H2-persist: Intentions to Persist

To investigate H2-persist, we analyze students' survey responses to determine the impact of the AIF system on students' intentions to persist in CS. We use a linear mixed-effects model, as each student twice rated their intentions to persist in CS (once in the pre-survey and once in the post-survey). In Table 6.2, Model A, time takes on a value of '0' for the pre-survey and '1' for the post-survey. AIF takes on a value of '0' for all observations at time 0 (as no students had experienced the AIF system at this time), and takes on a value of '1' at time 1 if the student was in the treatment group and received the AIF system. We found that AIF significantly improves CS persistence intentions, as students who used the AIF

Table 6.2: Estimated coefficients (Standard Error) of linear mixed models with repeated measures predicting CS persistence intentions.[7]

	Model A	**Model B**
AIF	2.304 (0.968)*	2.234 (0.968)*
Time	-0.506 (0.682)	-0.472 (0.682)
Pretest	-	0.502 (0.394)
Intercept	24.48 (0.985)***	22.25 (1.995)***
Observations	50	50

Significant codes (p <): + = 0.1, * = 0.05, ** = 0.01, *** = 0.001.

system had persistence intentions that were 2.30 points higher (p = 0.017) than students in the control condition. This means that after completing the first two tasks, students in the control condition had an average CS persistence intention score of 23.97, while students in the AIF condition had an average score of 26.37 (about 9.59% higher).

6.5.3 H3-idle: Engagement

To investigate H3-idle on student engagement, measured by reducing an individual student's idle time, using ordinary least squares (OLS) linear regression models on programming log data (Table 6.3). We used an OLS regression, since we only had one observation of engagement (idle time) per student. In Model A in Table 6.3, we predicted the percent of total programming time a student spent idle controlling for only for condition, and found that students who received the AIF system spent significantly less time idle (p = 0.014). On average, students spent 22.5% of their programming time idle if they were in the control group, versus only 4.6% if they were in the AIF group. This means that the AIF system had a substantial impact on student engagement, as it reduced idle time by 79.5%. In addition, we did not find evidence that this difference was caused by pre-existing differences between students, as the results remain largely unchanged when we control for prettest score (see Model B, Table 6.3). In sum, we found **strong support for H3-idle**, that the AIF system improves student engagement.

[7]Within-group errors were modeled to have an autoregressive structure with a lag of 1, given the time-lag between observations.

Table 6.3: Estimated coefficients (Standard Error) of OLS linear regression models predicting student engagement (measured as percentage of programming time spent idle).

	Model A	**Model B**
AIF	-0.179 (0.067)*	-0.191 (0.078)*
Pretest	-	0.001 (0.015)
Intercept	0.224 (0.047)***	0.188 (0.074)*
Observations	25	25

Significant codes (p <): + = 0.1, * = 0.05, ** = 0.01, *** = 0.001.

6.5.4 H4-perf: Programming Performance

We investigated H4-perf that the AIF system improved programming performance, by analyzing programming log data on the first two tasks. We used a linear probability model with mixed effects to predict the likelihood that a student completed an objective ('1' = completed, '0' = not completed) during the experimental phase of the study (i.e., the first eight objectives in which the experimental group used the AIF system). Thus, a total of 200 observations (i.e., whether a student completed a given objective) were nested within the 25 participants for this analysis.

We first predicted the likelihood that a student completed an objective, using the student's prettest score and the treatment as predictors (Model A, Table 6.4). We found no effect of a student's prettest score on their likelihood of completing an objective (p = 0.315). The AIF system has a marginally statistically significant impact on programming performance (p = 0.098), as students in the *Exp* condition were 13.1 percentage points more likely to complete an objective than students in the control condition. Thus, an average student (as measured by pretest score = 4) would be expected to complete 81.4% of the objectives if they were in the control condition and 94.5% of the objectives if they were in the AIF condition.

6.5.5 H5-learning: Learning

To investigate H5-learning, that the AIF system improved learning, we examined student performance on the last phase of the study (i.e., on task 3, in which neither group used the AIF system to complete the last five objectives). We again use a linear mixed effects model to predict the likelihood that a student completed an objective ('1' = completed,

Table 6.4: Estimated coefficients (Standard Error) of linear probability models (LPM) with mixed effects and repeated measures predicting the likelihood that a student completed an objective.

	Experimental Phase (Part 1)	**Learning Phase (Part 2)**
	Model A	**Model B**
Pretest	-0.017 (0.017)	0.002 (0.029)
AIF	0.131 (0.08)$^{+}$	0.254 (0.135)$^{+}$
Intercept	0.882 (0.083)***	0.470 (0.142)***
Observations	200	125

Significant codes (p <): + = 0.1, * = 0.05, ** = 0.01, *** = 0.001.

'0' = not completed). We first predict the likelihood that a student completes an objective controlling for a student's prettest score and the treatment (Model C, Table 1). We again find no effect of a student's prettest score on their likelihood of completing an objective (p = 0.952). Students receiving the AIF treatment were 25.4 percentage points more likely to complete an objective than students in the control condition, but this difference was only marginally significant (p = 0.056). Thus, a student with a pretest score of 4 would be expected to complete 47.7% of the objectives if they were in the control condition and 73.1% of the objectives if they were in the AIF condition.

6.5.6 Exploratory Mediation Analyses

We conducted an exploratory, post-hoc mediation analysis to determine if the AIF system improved objective completion in the experimental phase (part 1, with 200 observations) *because* it increased student engagement. We constructed a path model [Ull03] to conduct our mediation analysis, allowing us to statistically suggest causal relationships in our variables: AIF, Engagement, and Objective completion. Controlling for pretest score, we found that once engagement is taken into account, the AIF system did not have a *direct* effect on objective completion (coeff. = -0.008, p = 0.905). Instead, we found evidence for an indirect effect of the AIF system on objective completion *through* the impact of the AIF system on engagement. Figure 6.2 illustrates the results from our path model, which finds that the AIF system reduces idle time by 17.9 percentage points (p = 0.005), and that idle time has a large, negative impact (coeff. = -0.727, p < 0.001) on the likelihood that a student completes an objective. Moreover, our mediation analysis revealed the size of the indirect effect of the AIF system: students who receive the AIF system are 13.0 percentage points more likely to

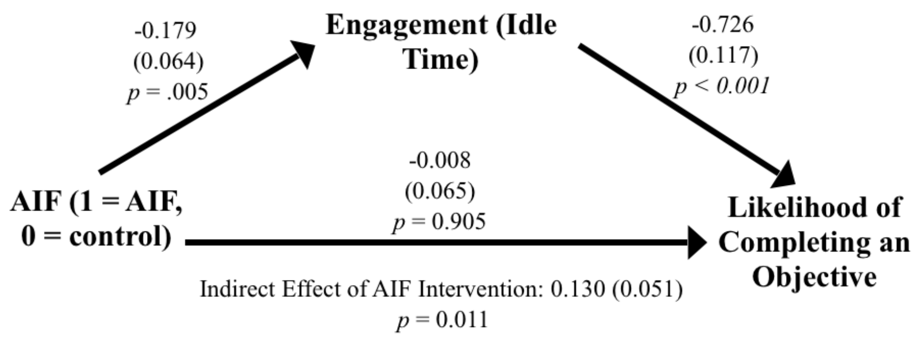

-0.179
(0.064)
p = .005
Engagement (Idle Time)
-0.726
(0.117)
p < 0.001
-0.008
(0.065)
p = 0.905
AIF (1 = AIF, 0 = control)
Likelihood of Completing an Objective
Indirect Effect of AIF Intervention: 0.130 (0.051)
p = 0.011

Figure 6.2: Mediation test results for effect of AIF intervention on idle time and likelihood of completing an objective during the experimental phase. Model controls for the effect of pretest scores. Clustered robust standard errors are shown in parentheses. N = 200 observations from 25 students.

complete an objective (p = 0.011) because the AIF system increased their engagement.

6.6 Discussion and Limitations

Our results provide compelling evidence that AIF can significantly improve students' intention to persist in CS. This impact of the AIF system is especially important, given that our participants had not yet entered university or declared a major, and thus the use of our AIF system could entice more students to study CS. In addition, these findings are important to the CS education community, since prior work in tutoring systems in computing *mainly* focus on the impact of feedback on students' cognitive outcomes, such as learning and performance, rather than affective outcomes which can dramatically impact student decisions. To our knowledge, this is the first evidence that programming feedback can improve students' desire to persist in CS, and this finding is a primary contribution of this work. We believe a primary reason for this impact on student's intentions is our feedback designed to ensure that students receive positive messages and confirmation of their success [Lew11].

Our second compelling result is that AIF significantly improved students' engagement with our programming tasks. In particular, the system dramatically reduced idle time, from almost a quarter of students' time to less than 5%. For context, this effect on idle time is larger than that found in prior work from using a block-based instead of a text-based programming environment [Pri15], which is often touted as an important way to better engage novice programmers. In addition, these findings are consistent with our qualitative interviews, which suggested that the AIF helped keep students on track, letting the students know what they had completed and what there was left to do. These results are also consistent with the "uncertainty reduction" hypothesis presented by Mitrovic et al. [Mit13], suggesting that positive feedback helps students continue working since they are more certain about their progress in an open-ended task. Based on our combined qualitative interview analysis and quantitative log data analysis, we conclude that the AIF system helped students stay engaged. Moreover, our mediation analysis suggests that the AIF significant impacts on idle time directly helped students complete more objectives as we discuss below. We believe that these impacts are a direct consequence of well-designed adaptive immediate feedback that helped students understand their meaningful successes and mistakes.

We also find suggestive evidence that the AIF system improved students' performance and learning. While only marginally significant, the effect sizes on performance and learning were moderately large, and our ability to detect them may have been limited by our smaller sample size. We also attribute these results to our AIF system design to provide positive and corrective immediate feedback. This is because prior work has shown that well-designed adaptive feedback that is provided to students just when it is needed can improve both performance and learning in programming [Mar19b; Mar20c]. Our AIF system also provides students with affective feedback via pop up messages, that may boost student motivation just as it may be flagging. Our mediation analysis shows that the observed improvements in students' performance was likely *due to* the reduction in idle time, so engagement matters for both affective and cognitive outcomes [Blu06]. This suggests that it might be plausible to improve the cognitive benefits of other forms of feedback (e.g. misconception feedback [Gus18], with positive feedback; however, researchers should be cautious not to benefit specific outcomes, and hurt another [Boy08].

This study has five main limitations. First, while our focus was to investigate the impact of our adaptive immediate feedback, our system also provided on-demand hints, which could have affected students' outcomes. However, we analyzed hint usage and found it was not significantly different between the groups. Further, students could and did ask for help from the camp teachers, but our observations reveal that this did not happen frequently or for long. Second, the interviews may reflect response bias or novelty effects that may have led to more positive answers, but we tried to minimize potential bias by asking for both the positive and negative aspects of AIF features. Third, having the interview about the AIF system before task 3 could have improved students' motivation in the *Exp* group to complete the third programming task. Fourth, while in our study procedure we collected a post-test from students, we decided not to report learning gains (i.e. the difference between pretest and posttest scores) since, while doing our analysis, we found 5 students who did not submit their tests (although during the camp all students claimed they completed all the tests). Fifth and finally, there may be other reasons that AIF reduced idle time; for example, breaking down programming tasks into smaller objectives may have made the tasks less difficult for students. However, our analysis of task 3 suggests that this potential reduction in difficulty did not hinder student learning for a subsequent task without AIF.

6.7 Conclusion

The **contributions** of this work are the design and development of a new adaptive immediate feedback system, and a controlled study demonstrating that our AIF system: 1) is well-received by students, 2) improves students intention to persist in CS, 3) increases students' engagement, 4) improves students' performance, and 5) learning. Our interview results confirmed our hypothesis (H1-qual) that the AIF system would be positively perceived by students, and our survey results confirmed our hypothesis (H2-persist) that AIF would improve students' intention to persist in CS. Additionally, by investigating students' log data, we confirmed our hypothesis (H3-idle) that AIF system would improve students' engagement during programming (measured by students' idle time spent while programming). Moreover, from analyzing students' programming performance, our results partially support our hypotheses (H4-perf) that AIF system would improve students' performance and (H5-learning) learning. In future work, we plan to generalize our approach to develop adaptive immediate feedback for more assignments. In addition, we plan to conduct larger classroom studies to investigate the impact of adaptive immediate feedback within the context of graded assignments.

CHAPTER

7

STUDY 5: ADAPTIVE IMMEDIATE FEEDBACK (AIF 3.0) SYSTEM FOR BLOCK-BASED PROGRAMMING: DESIGN AND EVALUATION

This chapter was adapted from: Marwan, Samiha et al. "Adaptive Immediate Feedback for Block-Based Programming: Design and Evaluation". Submitted to the IEEE Transactions on Learning Technologies, 2021. The original text has been modified as follows: The Related Work section has been incorporated into Chapter 2.

7.1 Abstract

Theories on learning show that formative feedback that is immediate, specific, corrective, and positive is essential to improve novice students' motivation and learning. However, most prior work on programming feedback focuses on highlighting student's mistakes, or

detecting failed test cases after they submit a solution. In this work, we present our Adaptive Immediate Feedback (AIF) system, that uses a hybrid data-driven feedback generation algorithm to provide students with information on their progress, code correctness, and potential errors, as well as encouragement in the middle of programming. We also present an empirical controlled study using the AIF system across several programming tasks in a CS0 classroom. Our results show that the AIF system improved students' performance, and the proportion of students who fully completed the programming assignments, indicating increased persistence. Our results suggest that the AIF system has potential to scalably support students by giving them real-time formative feedback and the encouragement they need to complete assignments.

7.2 Introduction

Novice block-based programming environments like Scratch [Mal10], Snap*!* [Gar15] and Alice [Coo00] were designed to foster more positive experiences with programming for novices by enabling creativity, and eliminating syntax errors [Man06; Dan12; Gar15]. Increasingly, introductory programming classrooms in K-12 and college classrooms engage novices to create interesting programs (e.g. with interactive input and graphical output), in block-based environments to create a positive first experience with programming. However, since novices have little prior knowledge, they often face uncertainty when programming, e.g. not knowing how to start or whether their code is correct or not, [Wan21], especially during more complex and open-ended assignments, which may take 30-60 minutes, and have many different correct solution approaches [Den07]. Without confirmation that they are making progress as they work, students may lose motivation and give up early [Gor20], or even delete correct code that they think is responsible for errors [Don19; Sha20]. Additionally, without the skills to detect errors, students may submit incorrect code without realizing it, leading to lower course performance, and missed opportunities to debug their errors.

It is therefore especially critical that novice programmers have access to *timely, formative* feedback to address this uncertainty. In a review on effective feedback, Shute argued that formative feedback can improve students' motivation and learning [Shu08]. Formative feedback is defined as a type of task-level feedback that provides *specific, timely* information to a student in response to a particular problem or task, based on the student's current ability [Shu08]. From a cognitive learning theory (CLT) perspective, formative feedback can reduce students' uncertainty about how well, or poorly, they are performing on a task [Mit13; Paa03], and it can therefore increase students' motivation and persistence to complete tasks by revealing the progress that students have already made [Mar20a]. However, it is hard for computing instructors to provide such formative feedback for every student, especially in larger classes, or during homeworks outside of class. It is also difficult to develop *automated feedback*, since the interactive input and graphical output that make novice programming environments engaging also make it difficult to assess student code with traditional input/output-based test cases.

While some automated feedback approaches have been developed for block-based environments [Bal18; Wan21; Mar20a; Gus17], they have important limitations. First, some of these systems focus on negative *corrective feedback*, with less emphasis on *positive* encouragement for students that marks their progress [Gus18]. Second, some of these systems either require extensive expert effort in hand authoring rules, which is hard to scale across assignments [Mar20a; Gus17], or use error-prone data-driven algorithms to generate such feedback [Mar20b; Sha20]. Lastly and most importantly, few of these systems have been evaluated in authentic classroom settings to measure their impact on student outcomes.

In this work, we present an Adaptive Immediate Feedback system (`AIF`), shown in Figure 7.1, that leverages a hybrid *data-driven* model refined with *experts'* constraints to generate formative feedback with *high* accuracy and *scalability* across programming tasks. This is our third version of the `AIF` system, which we call `AIF 3.0`, or for simplicity, just `AIF`. This system offers students immediate, specific, continuous feedback on their progress through a programming assignment in the Snap*!* environment. To do so, the algorithm behind the `AIF` system breaks a program down into a set of subgoals (i.e. programming task objectives) and calculates student progress on each from 0-100%, which is updated every time a student edits their code. The `AIF` system interface then presents this progress feedback through a subgoal list, and offers encouraging messages as students progress. Our `AIF` system improves over prior systems that provide formative feedback [Gus18; Gus17; Mit13; Mar20b], and extends them in three critical ways. First, the system uses a *hybrid* data-driven subgoal detector algorithm to assess students' progress on programming tasks' subgoals, described in detail in [Mar21a]. These subgoals are generated automatically from student data, and then *refined* using expert-constraints, rather than purely expert-authored rules [Mar20a; Gus17], or a purely data-driven model [Mar20b]. This allowed us to scale the feedback to support a multi-week programming unit, and ensure higher quality feedback. Second, our `AIF` system can assess students progress more granularly, providing an estimation of their progress from 0-100%, rather than binary correct/incorrect feedback, giving students more of a sense of progress. Third, using effective design strategies, the `AIF` system provides all four forms of effective formative feedback; i.e. feedback that is corrective, specific, immediate, and positive.

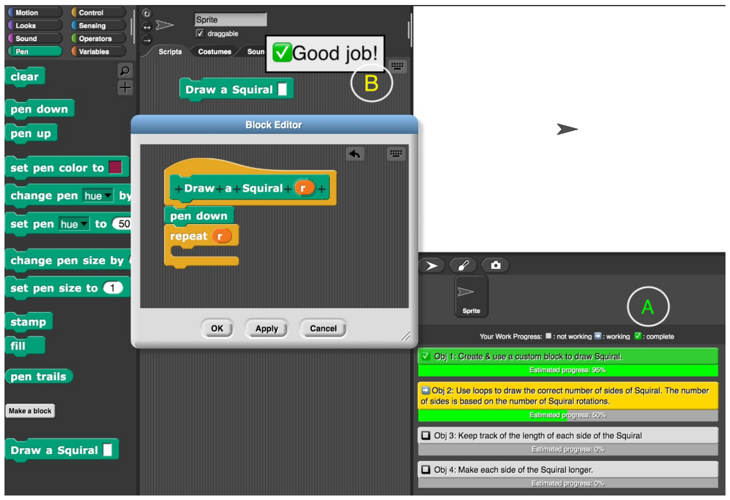
We also present an empirical evaluation of the `AIF 3.0` system over 3 weeks of a university-level introductory computing course for non-majors. This evaluation goes beyond prior evaluations of block-based feedback tools, which were limited to technical assessments [Wan21], lab studies [Mar19b], or single-day studies [Mar20a; Mar20b; Sha20]. We also leverage multiple assessment mechanisms to evaluate the system's impact, including a transfer programming task, post-test, and surveys. This evaluation helps us to understand the potential for such systems for wider adoption in classrooms.

This work seeks to answer the following research questions: *In an authentic classroom setting, what impact does hybrid data-driven adaptive immediate feedback (*`AIF`*) have on students' RQ1) performance, RQ2) rate of task completion, and RQ3) learning, and RQ4) how is it perceived by students?* We hypothesize that by showing students their progress on assignment subgoals, and by highlighting incomplete subgoals, the `AIF 3.0` system will encourage students to persist longer until completing the programming tasks, leading to better performance, and ultimately more learning from those tasks. Our results show that students using the `AIF 3.0` system performed significantly better overall on programming tasks, and had significantly higher overall completion rates. This suggests that students were motivated by the `AIF 3.0` system to persist in completing these tasks. Additionally, we found students who used the `AIF 3.0` system had higher scores in a future transfer task and a post test, but the difference was not significant, suggesting inconclusive results. Survey data highlight *how* the `AIF 3.0` system helped students and also suggest trade-offs in the design of the system. We also present case studies illustrating how students can use the `AIF 3.0` system and reasons why the `AIF 3.0` system led to improvements in student performance and completion rates, compared to students not using the `AIF 3.0` system.

In summary, the key **contributions** of this work are: (1) the `AIF 3.0` system that provides real-time formative feedback, derived from a hybrid data-driven algorithm, in a block-based programming environment; (2) a controlled empirical study in an authentic classroom setting showing increased student performance, and completion rates, suggesting that the `AIF` system can increase students' persistence to complete programming tasks; (3) case studies illustrating how such feedback can encourage students to persist to complete programming assignments correctly, and evidence that the system can be effective in an authentic classroom context.

7.3 Adaptive Immediate Feedback (`AIF`) System Design

In this section, we present the `AIF 3.0` system that uses a hybrid data-driven model to provide students with effective formative feedback. The main idea of the `AIF` system is to simplify the students' learning process, and motivate them to complete their programming tasks, leading to an increase in students' performance and learning. To do so, we designed the `AIF 3.0` system to provide feedback that is *corrective*, *specific*, *immediate*, and *positive*, which are key aspects of effective formative feedback for learning as specified in literature [Shu08; Sch04]. The `AIF 3.0` system augments the Snap*!* block-based programming environment with three main components: (1) A subgoal list, which breaks down the current programming task requirements into 3-5 smaller, more manageable task objectives (shown in Figure 7.1, A). (2) Pop-up messages, which provide students with encouraging messages based on their progress (shown in Figure 7.1, B). (3) An algorithm for data-driven subgoal detection which generates the list of subgoals, and detects students' progress on them in real time, driving the feedback in the subgoal list and the pop-up messages. The first two components represent the interface, described in Section 7.3.1, while the third component presents the back end of the `AIF 3.0` system, described in Section 7.3.2.



Your Work Progress: ◻ : not working ➡ : working ✅ : complete

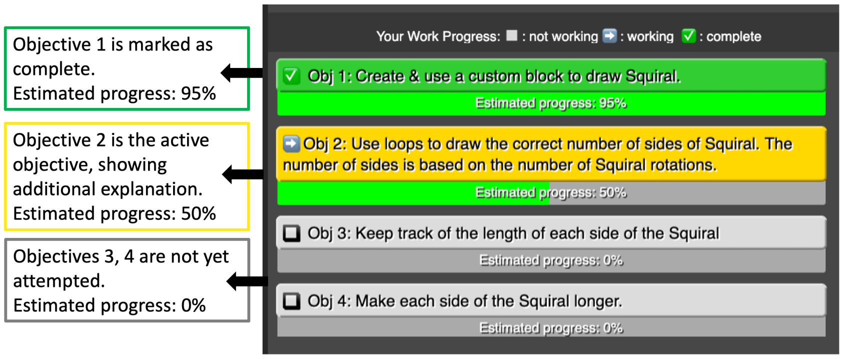
✅ Obj 1: Create & use a custom block to draw Squiral.
Estimated progress: 95%

➡ Obj 2: Use loops to draw the correct number of sides of Squiral. The number of sides is based on the number of Squiral rotations.
Estimated progress: 50%

◻ Obj 3: Keep track of the length of each side of the Squiral
Estimated progress: 0%

◻ Obj 4: Make each side of the Squiral longer.
Estimated progress: 0%

Figure 7.1: The Data-Driven Adaptive Immediate Feedback (AIF) system, with subgoal list (A), and pop-up message (B), augmenting the Snap*!* block-based programming environment.


Objective 1 is marked as complete.
Estimated progress: 95%

Objective 2 is the active objective, showing additional explanation.
Estimated progress: 50%

Objectives 3, 4 are not yet attempted.
Estimated progress: 0%

Your Work Progress: ■ : not working ➡ : working ✓ : complete

✓ Obj 1: Create & use a custom block to draw Squiral.
Estimated progress: 95%

➡ Obj 2: Use loops to draw the correct number of sides of Squiral. The number of sides is based on the number of Squiral rotations.
Estimated progress: 50%

☐ Obj 3: Keep track of the length of each side of the Squiral
Estimated progress: 0%

☐ Obj 4: Make each side of the Squiral longer.
Estimated progress: 0%

Figure 7.2: The subgoal list for the Squiral task, with 4 objectives, each explained on the right.

7.3.1 The AIF 3.0 System Interface

Our current interface design is a product of years of prototyping, deployment, and refinement of prior versions of the AIF system [Mar20a; Mar20b; Sha20]. Below we describe its current interface components in detail.

Subgoal List

The primary feature of our system is a list of subgoals, as shown in Figure 7.1, A. The subgoal list breaks down a given programming task into a set of smaller objectives (i.e. subgoals) that students can attempt one at a time, which has been shown to be effective in improving students' performance in programming [Mar16]. In particular, we created the subgoal list for 3 reasons. First, many novices struggle with how to *start* an assignment [Gor20; Mar21b], so we provide a concrete set of steps (a plan) to get them started. Second, for complex programming tasks, especially open-ended ones, students must navigate a large search space, which can lead to high cognitive load [Swe88]. By breaking down a problem into smaller subgoals, which are smaller structural parts of the overall programming assignment [Mar16], the system should mitigate this challenge and increase their ability to solve the task. Third, for subgoals to be effective, students need to know when they have completed one subgoal, so they can focus on the next. To reduce uncertainty about subgoal (and assignment) completion, the `AIF 3.0` continuously updates progress bars to show the student's estimated progress on each subgoal. This progress is calculated by an algorithm that analyzes student code after each edit, as described in Section 7.3.2. Figure 7.1, A shows the subgoal list of a programming assignment that asks to draw a *Squiral* shape as shown in Figure 7.4. For example, one subgoal is to *"Make each side of the Squiral longer"*.

Subgoal list design: When students open the Snap*!* environment and choose which task to start working on, they will see a list of subgoals shown as objectives, each with a short, hand-authored label[1]. The subgoal list is placed within the programming environment to align with the multimedia learning principle of contiguity, which states that information needed for students performing a task is most useful when it is placed next to where it is needed [Mor99]. Initially, each subgoal background is colored grey to indicate that none of the subgoals have been attempted. Students can interact with the subgoal list in two ways. First, they can click on a subgoal to highlight it in yellow and get more information, as shown in 'subgoal 2' (i.e. obj2) in Figure 7.2. This information consists of expert-authored explanations derived from the task instructions, since research suggests that students prefer feedback augmented with explanatory text [Pri21], and find it more interpretable [Mar19a]. Second, students can double-click on a subgoal to mark it as complete (as shown in 'obj 1' in Figure 7.2), aligning with self-regulated learning principles, which state that learning is improved when students track their own progress and self-assess their learning process [Lok16]. This feature was also recommended by students in evaluations of earlier versions of the system [Mar20a; Mar21b].

Adaptive Progress Feedback Display: The subgoal list adapts to students' code edits by providing feedback on students' progress on each subgoal in real time, aligned with the learning design principles of specific, immediate feedback that enables students to understand what specific actions they have just taken, that are leading to mistakes or progress. Rather than telling students only whether a subgoal is complete or not, as in typical programming autograders [Mar20a; Bal18; Wan20], `AIF 3.0` presents a progress bar underneath each subgoal to show how far students have progressed toward completing that subgoal, as shown in Figure 7.2. This provides students with *specific, corrective* feedback on each edit they make to their code, by showing how that code edit increased or decreased the estimated progress toward a correct solution.

Promoting self-assessment: A key aspect to improve novice programmers' learning is to promote them to self-assess whether their program is correctly solving the given problem [Mar21b]. Therefore, we have purposefully designed two important features to encourage self-assessment and self-regulated learning in the `AIF 3.0` system. First, we provide transparency into how the system works, explaining to students that the algorithm that tracks their progress may sometimes fail to recognize correct solutions, as described

[1]We call the subgoals objectives in the interface to make it more understandable to novices; however we use the word subgoals in the paper for consistency.

Table 7.1: Examples of Pop-up Messages in the AIF 3.0 System.

Student state	**Pop up message**
<1/2 subgoals complete	Way to go! Onto the next objective!
>1/2 subgoals complete	You're almost there! Excellent
All subgoals are done	You finished all the objectives! Well done!
Fixed a broken subgoal	You restored a broken objective! It's Fixed!
Struggle/Idle , <half done	Don't give up! Keep going!
Struggle/Idle, >half done	You're almost there, keep going! You're making great progress!

below. Second, we set the maximum progress on each subgoal to 95%, instead of 100%, to remind students that it is up to them to self-assess and double-click on each subgoal when *they* decide it is complete, which is suggested by prior work [Mar21b].

Pop-up Messages

We developed pop-up messages because prior work shows that positive feedback that praises students' accomplishment and perseverance can promote students' confidence [Cec11], which may increase their persistence to complete tasks. Therefore, the `AIF 3.0` system provides positive feedback in the form of *congratulating* messages to praise students when they complete a subgoal or fix a broken subgoal. It also provides *encouraging* messages when students persevere by spending a long time without any progress, as shown in Figure 7.1, B. To design these messages, one researcher asked undergraduate students to construct messages to (1) praise a friend's achievement when they complete a subgoal, or (2) motivate a friend when they are struggling or losing progress. While students are programming, the `AIF 3.0` system selects a personalized pop-up message based on students' code and actions. For example if the system detects the completion of a subgoal, it will pop up a congratulating message like: 'Good job!', or an encouraging message like 'Don't give up' when a student has spent more than 4 min without making any progress[2]. Table 7.1 shows examples of pop-up messages.

[2]We chose a threshold of 4 minutes to limit pop-up distractions.

7.3.2 Data-Driven Subgoal Detection Algorithm

The `AIF` system provides adaptive immediate feedback on subgoals in reaction to students' code edits. To do so, the algorithm behind the `AIF` system must perform two operations. First, for a given programming task, it must break the task down into a set of subgoals. Second, the algorithm must be able to assess student code at any time (i.e. whether complete or incomplete), and evaluate how complete each subgoal is (0-100%). This feature allows the system to provide adaptive immediate feedback that is specific, corrective and positive, as we discuss in detail below.

In prior work, there are two common approaches to generate and assess subgoals: expert-authored and data-driven approaches. For expert-authored approaches, human experts define the subgoals of a correct solution, and create autograders for each objective to detect if it is complete or incomplete, for example using static code analysis [Mar20a; Gus17]. While expert-authored models are capable of providing highly accurate feedback, creating them is time consuming, requires extensive expert effort, and is hard to implement for open-ended programming tasks due to the large number of possible solutions. The second common approach uses data-driven models, which can detect subgoal completion in the current student's code based on *features* learned from historical student data [Mar20b]. While data-driven approaches are highly scalable across programming tasks, and require less expert effort, they are dependent on the quality of prior students' solutions, leading to a possibility of providing inaccurate feedback [Sha20].

In this work, we developed the `AIF 3.0` system using a hybrid model, combining both a data-driven model refined with expert constraints to detect subgoals with higher accuracy, using a small amount of expert effort [Mar21a][3]. The goal of using the hybrid model is to achieve the best of both the expert and data-driven models - building a system that can intelligently address the diverse but correct ways that students solve problems, while benefiting from human expertise in filtering, combining and communicating the data-driven subgoals.

The `AIF 3.0`'s hybrid data-driven model is responsible for subgoal generation, progress assessment, and feedback generation. This hybrid model extends a data-driven feature detector (DDFD) algorithm developed by Zhi et al [Zhi18]. The DDFD algorithm is designed to extract common features present in prior students' correct solutions (e.g. from prior

[3]We note that the use of the hybrid model for multiple programming tasks and incorporating it in the `AIF` 3.0 system are evaluated in this paper, but the technical details of the hybrid model, including the data-driven detectors and expert constraints, are more completely described in our prior work [Mar21a].

semesters) [Zhi18]. In brief, the DDFD algorithm works as follows: First, it translates students' correct solutions into abstract syntax trees (ASTs). For example, Figure 7.3 shows an example of students' code, and its corresponding AST. Second, it extracts common code shapes (i.e. subtrees of ASTs); such that a group of code shapes can represent a feature of a correct solution. More precisely, a **feature** describes a distinct property for a correct solution, such as: using a 'procedure with one parameter' in the code, or using a 'move' block nested with a variable are two distinct features of a correct solution as shown in Figure 7.3 A, and C, respectively. The DDFD then filters redundant code shapes, and performs hierarchical clustering of frequently co-occurring code shapes to generate more coherent features. In addition, the DDFD algorithm also defines *disjunction* shapes that identify when there are multiple, distinct ways of solving a subgoal, i.e. a set of code shapes where *one* is present in most solutions, but not *others*. For example, a student can draw a square using a 'loop' or a set of redundant blocks, and a correct solution must have *only one* of these disjunction shapes. It is worth noting that while the DDFD algorithm can capture multiple solution strategies, it is still limited to the available historical data. Once the DDFD algorithm learns features of correct solutions, it can be applied to new students' solutions to detect the completion or absence of these features. Since the DDFD algorithm works regardless of whether student code is complete or not, it can be used to provide *immediate* feedback on feature completion.

Despite its advantages, the DDFD algorithm suffers the following limitations. First, these data-driven features are *fine-grained*, making it hard for students, and sometimes even for instructors, to *interpret* the features' code shapes. Second, the DDFD algorithm can generate a large number of features for a small task with just 5-8 lines of code, due to the variety of student strategies to achieve correct solutions. These limitations make it likely that students would not be able to use or interpret information about features detected using DDFD algorithm on their code.

To tackle these challenges, we made several improvements to the DDFD algorithm to detect *subgoal* completion, instead of *feature* completion, in a more *interpretable*, and *concrete* way. We call our modified algorithm a hybrid subgoal detector, and its overall algorithm consists of five steps, using both automated and expert insight, which we explain below (with more details here [Mar21a]).

First, the hybrid subgoal detector algorithm applies the DDFD algorithm to students' correct solutions from several prior semesters to generate clusters of data-driven features. Second, human experts manually group the generated clusters into more meaningful clusters that are interpretable and reflect discrete assignment requirements. For instance, a

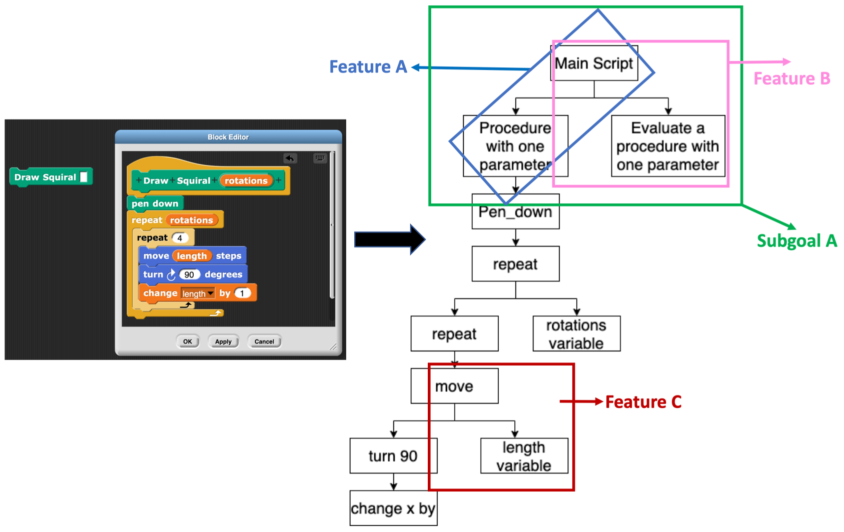


Figure 7.3: An example of an abstract syntax tree (AST) of solution code for the Squiral task. Feature A, B, and C are examples of Squiral's data-driven features.

programming task that includes 7-10 lines of code is broken down into 3 to 4 subgoals which reflect objectives found in the task instructions provided to students. To clarify the difference between a feature and a subgoal, consider the following example in Figure 7.3. Assume that the DDFD algorithm generates 2 features: one requires that student code includes a procedure with one parameter (Figure 7.3, Feature A); the second feature requires the evaluation of a procedure with one parameter (i.e. calling a procedure), Figure 7.3, Feature B. A meaningful subgoal in this case can be the combination of these two features, which means that a correct solution must have a procedure with one parameter which is called (i.e. evaluated) in the main script, as shown in Figure 7.3, subgoal A.

Third, we developed a percent progress estimate that reflects student code's *progress* on each subgoal. For example, if the algorithm generates four data-driven subgoals for a given exercise, and for a given student code the algorithm outputs `{50%, 0, 0, 80%}`, it means that the student code has completed 50% of subgoal 1, 80% of subgoal 4, and 0% of subgoals 2 and 3. The algorithm calculates this percentage by calculating how many code features needed for a given subgoal are present in students' code. This new progress tracking feature developed for `AIF 3.0` allowed it to provide more *specific* progress feedback. Fourth, experts tested and improved the quality of the subgoal detectors. We applied the algorithm to prior students' data and evaluated its accuracy. Human experts then searched for false detections – instances where the algorithm detects the completion or incompleteness of a subgoal but the expert disagrees. We modified the system to enable experts to add new constraints to each subgoal detector. Our experts added these new constraints, and ran the subgoal detectors again on each case to ensure they were corrected, resulting in new subgoals created in a hybrid data-driven and expert-authored manner. The methodology for adding new constraints and their impact on the accuracy of subgoal detection is evaluated in our prior work [Mar21a].

Fifth and last, we augmented each subgoal with a short, human-authored label designed to be *interpretable* for both students and instructors. We presented these labels in a subgoal list for each task as explained in Section 7.3.1. We then use this list to present feedback on each subgoal using the hybrid algorithm that continuously runs in the backend of the Snap*!* programming environment.

Initial Results

In Summer 2019, we developed and tested `AIF` version 1.0 [Mar20a]. We broke down programming tasks into a set of objectives (i.e. subgoals), and developed expert-authored autograders, called objective detectors in our prior work, to detect the absence or completion of each objective in student code. We embedded these autograders in the Snap*!* environment to provide students with adaptive immediate feedback during programming. The initial interface differed from the current version in three ways; it: (1) used only short labels for subgoals, (2) provide correctness feedback, i.e. if a subgoal is complete or not, and (3) did not allow students to check off a subgoal if they think it is complete. We evaluated `AIF 1.0` with students performing short programming tasks in a 1-day summer workshop [Mar20a]. Our controlled study showed that `AIF 1.0` improved students' engagement with the programming tasks, and increased students' intentions to persist in computer science. In addition, we found suggestive evidence that using the `AIF 1.0` system improved students' performance [Mar20a]. However, this study had a small sample size of students who volunteered to spend a summer day learning to program, and evaluated the system with short programming tasks.

In Spring 2020, we built `AIF 2.0` with autograders for more complex tasks using a data-driven feature detection (DDFD) approach, described in Section 7.3.2, to learn subgoals from prior correct solutions [Mar20b]. We conducted a controlled study for one programming homework task and showed that `AIF 2.0` increased students' engagement as measured by increased time on task. We also found suggestive evidence that such increase in active time might have improved students' performance. However, investigating the quality of data-driven subgoals detections, we found instances where false detections (i.e. false positives and false negatives) occurred, which might have affected students' performance or trust in the learning environment [Sha20]. To mitigate these inaccuracies, we developed a hybrid data-driven algorithm, described in more detail in [Mar21a], which adds human constraints to improve the quality generated data-driven feedback.

The remainder of this paper presents our study of `AIF 3.0`, which uses the hybrid data-driven model for multiple assignments across multiple weeks in an authentic introductory non-majors computer science classroom. `AIF` 3.0 generates more specific feedback, and its interface reminds students to reflect on their progress. Such immediate progress feedback can also help students quickly realize and correct mistakes, such as deleting correct blocks, or adding unnecessary blocks that will not change their progress [Don19]. For simplicity, we will refer to `AIF 3.0` as `AIF` throughout the rest of the paper.

7.4 Method

We conducted a controlled classroom study in Spring 2021 in an introductory programming course (CS0) at a public U.S. university. This course introduces the principles of Computer Science (CS) to non-majors, where students learn programming in Snap*!* environment. Our research questions are: **RQ:** In an authentic classroom setting, what impact does a hybrid data-driven `AIF` have on (**RQ1:**) students' performance, (**RQ2:**) task completion rate, and (**RQ3:**) learning, and (**RQ4:**) how is it perceived by students?

We hypothesized that `AIF` will help students know whether they are progressing or not, encouraging them to *persist* longer, which will lead to higher performance (measured by students' grades in these tasks) than students in the *Control* group (**H1 - performance**). Furthermore, knowing that their code is incorrect or incomplete will motivate students to keep working to finish it *fully*, which will result in higher rates of *fully correct*, complete code being turned in (**H2 - completion**). Additionally, by completing more of the assignment correctly, we hypothesize that students will *learn more*, which will be reflected in increased performance on a future transfer task, and on a future post test (**H3 - learning**).

7.4.1 Population

The CS0 course includes 65 undergraduate students, 50 of whom consented to our IRB-approved study. In this population, 63% of participants self-identified as male, 28.26% as female, and 8.6% unspecified; 62.5% are 20 years old or younger, 26.08% are 24 years old or younger, and 8.6% otherwise. Students self-identified their race/ethnicity with 63.03%

White, 15.2% Asian, 10.8% Black or African American, 4.3% Native American, and 6.5% Other. While the course was designed for non-majors without any programming experience; however, some might have had prior programming experience.

7.4.2 Procedure

We conducted a controlled study that lasted for three weeks, as shown in Table 7.2. Due to the COVID-19 pandemic, this class was held online through Zoom. On the first day of the CS0 class, one researcher introduced the study to all students and offered them the opportunity to consent to participate in the research. Students were then randomly assigned to the *AIF* condition or the *Control* condition, allowing for between-subjects comparison. In the *AIF* condition, students received feedback through the `AIF` system, and in the *Control* condition, students did not have access to the `AIF` system. We note that students in both groups had detailed assignment instructions with ordered steps. Additionally, both had equal access to request next-step hints from Snap*!*, which suggests a single edit to bring student code closer to the correct solution. Hints were *already* part of the class to help students when they got stuck, so we do not analyze its effect in this study. Out of the 50 students, 23 were assigned to the *AIF* condition, and 27 were assigned to the *Control* condition.

Table 7.2: Study 5 Procedure.

Week no.	**AIF Group (n=23)**	**Control Group (n=27)**
1	Consent form + Presurvey (50 students)	
	HW 0 [Pre-assessment]	
2	Polygon Maker (in-class)	Polygon Maker (in-class)
	Squiral (HW 1)	Squiral (HW 1)
	Daisy (HW2)	Daisy (HW 2)
3	Brickwall (HW 3 - no feedback)	Brickwall (HW 3 - no feedback)
4	Post-survey	
	Post-test	

Pre-assessment (HW0): In the first week, students were assigned HW0, which is a self-paced, independent programming assignment that teaches students the usage of loops [Wan20]. It includes 9 subtasks, each requiring 3-10 blocks of code, with increasing difficulty.

We used HW0 as a pre-assessment[4] of students abilities, and Snap*!* did *not* provide AIF support to either condition. We did not perform a traditional pre-test, since this was many students' first experience with programming, and we did not want to create a harmful expectation that students should know material that is not yet covered in class.

Practice Tasks: We describe three programming tasks, together called practice tasks, that students completed using AIF or not, based on their condition assignment, with the graphical output of each shown in Figure 7.4. In week 1, students had their first *in-class* programming task called *PolygonMaker*, which asks students to create a procedure with 3 parameters: *'n'*, *'len'*, *'thick'*, to draw a polygon with *'n'* sides, each with with length *'len'*, and thickness *'thick'*. In week 2, students had two homework tasks: *Squiral* and *Daisy*[5]. *Squiral* asks students to create a procedure that takes user input *'r'* and draws a spiral square with *'r'* rotations. *Daisy* asks students to create a procedure that draws a daisy with a user-specified number *'n'* of overlapping circular petals with alternate colors.

Post Measures: In week 3, students completed a post-test programming assignment without the AIF system in either condition, to compare the impact of AIF on learning for a similar transfer task without adaptive immediate feedback. This homework 3, *Brickwall*, asked students to draw a wall of bricks, with alternate rows of bricks using nested procedures, variables, conditions, and loops. In week 4, the instructor gave students a post-survey that collected students' perceptions about whether the AIF was helpful and why; followed by a post test. The post test includes 7 multiple choice questions on variables, loops, and conditionals, adapted from the block-based Commutative Assessment [Wei15].

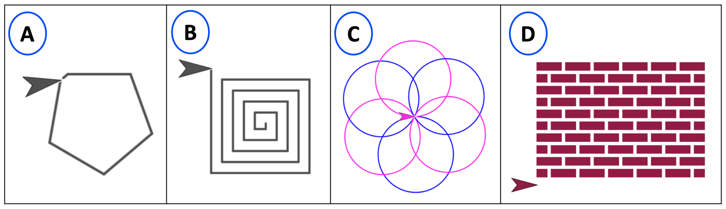


Figure 7.4: Expected Output of Polygon Maker (A), Squiral (B), Daisy (C), and Brick Wall (D).

[4]HW0 was actually due shortly *after* the first in-lab assignment. We discuss implications when analyzing the results.

[5]We omitted one in-class programming task in week 2 from analysis, due to technical issues that caused the AIF system to appear in both conditions.

7.4.3 Measures

In this study, we collected two sources of data: log data and surveys, which we analyzed to measure the following:

Pre-assessment Scores

We measured students' scores in HW0 (our pre-assessment) across both groups. HW0 consists of 9 subtasks, where at least 6 must be completed to earn full credit, and students cannot proceed to the next subtask without finishing the previous ones. All tasks are automatically assessed as complete or incomplete. We categorized students' scores into 3 categories: we gave students a score 1 if they completed < 6 subtasks, 2 if they completed only 6 subtasks, and 3 otherwise. Since the data on HW0 performance is non-normal, we used a Mann-Whitney U test to measure the statistical significance of differences between students' scores in the *Control* and the *AIF* (i.e. treatment) groups.

Performance

We measured the impact of the `AIF` system on students' performance by comparing students' scores (i.e. grades) in all practice tasks across the two treatment conditions. We graded each task, blind to condition, adopting a 4-item binary rubric for each, adapted from the assignment instructions. Each rubric item roughly corresponds to an `AIF` subgoal, and completing all four rubric items is equivalent to completing the programming task. One researcher graded students' submissions of all programming tasks, where the grades ranged from 0% (i.e. student failed to complete all rubric items) to a maximum of 100% (i.e. student successfully completed all rubric items). Across both conditions and all practice tasks, the mean programming performance was 89% (min = 0% ; SD = 22.7%; max = 100%).

Analytical approach: We measured programming performance for all practice tasks together to measure the overall effect of the `AIF` system on students' programming performance through several weeks. To do so, we used linear mixed effects models because the data has a nested structure, where a student's score in each exercise is treated as one unit of analysis, accounting for the lack of independence between observations [Bru11].

Completion Rates

We defined the **completion rates** of each group (*Control* or *AIF*) as *the proportion of students who turned in fully correct, complete code*, where a fully complete submission means students' score (i.e. grade as described above) is 100%. Recall that students were given detailed instructions and animations of the expected output of all the programming tasks, they have 4-6 days to complete any given task, and they also have optional office hours to seek help from TAs if needed. We expect that all students should be able to complete all of the practice tasks, if they were to spend sufficient time and use available help. However, each task counts for only about 3-4% of student overall grades, so perfect performance (i.e. 100% complete) may not be a goal for students who have many obligations. Therefore this measure helps to investigate whether the `AIF` system can *motivate* students to correct their code and keep working to complete it.

To measure task completion, we graded each student submission as complete when their grade is equal to the maximum score; i.e. 100%. Across both conditions and assignments, the mean completion rate was 74% (SD = 44.01%).

Analytical approach: We evaluated students' correctness rate on all three practice tasks together to measure the overall impact of the `AIF` system on students' completeness rate for all three tasks through several weeks of programming. To do so, we used a linear probability model (LPM) with mixed effects to predict the likelihood that a student completed a task ('1' = completed, '0' = not completed). Because data is binary (i.e. whether a student submitted a complete correct solution (score = 100%), or not (score < 100%)), we used Linear probability models (LPMs) which are simply linear mixed effects models that predict *binary* outcomes [VH15b].

Learning

We measured students' learning in two ways. First, we measured students' performance in a later task, HW3, where students in both conditions did not use the `AIF` system and therefore received no feedback. The *BrickWall* homework measures how well students learned programming skills in the practice tasks (with or without AIF), and their ability to apply them to a new task without additional support. HW3 shares similar programming concepts with the previous tasks (e.g. use of procedures, loops and conditions). We used students' performance in *BrickWall* task as a measure of learning, since we assumed that students who learn more from the practice tasks should perform better in this future task. Because we only have one transfer task, we used a Mann-Whitney U test to measure the statistical significance of differences between the scores of students in this transfer task, across the *Control* and the *AIF* groups. We also measured the completeness rate of the transfer task using Fisher's exact test because the data is binary; i.e. whether a student completed the task or not.

Second, we measured students' learning by comparing their post-test scores. The post-test includes 7 questions, and for each question, students got a score of '1' if they correctly answered this question, or '0' otherwise. As a result, the maximum score (i.e. 100%) of any student is 7 and the minimum is 0. We used a Mann-Whitney U test to measure the statistical significance of differences in students' (non-normal) post-test scores across the *Control* and the *AIF* groups.

Students' Perceptions

We conducted thematic analysis on students' open-ended responses in the post survey, to identify themes on why the `AIF` system is more or less helpful. We followed six steps suggested in the practical guide of performing thematic analysis by Maguire et al [Mag17]. Two researchers, independently, (1) got familiar with the data, (2) generated a total of 8 initial codes, (3) and then met to combine initial codes in dominating themes which resulted in 5 themes, (4) reviewed all themes together, and then (5) refined them to focus on only two main themes: *mechanisms* for which the `AIF` system was helpful, and *limitations* in its design. Step 6 was the write up of the results, which we present in Section 7.5.5.

7.5 Results & Analysis

These results are presented in the order of the methods above. We first investigate the pre-assessment to ensure the random assignment to groups was balanced with respect to performance. We then analyze student log files reflecting their programming work to explore our hypotheses: H1 on performance, H2 on completion, and H3 on learning. We then present some case studies to illustrate how the `AIF` system impacted these measures, and finally we explore the student survey results to understand student perceptions of the system.

A Mann-Whitney U test on student scores on HW0 shows that there was no significant difference ($p = 0.9$) on the pre-assessment (described in Section 7.4.2), between the *AIF* group (M=86.4%), and the *Control* group (M=87.7%). This demonstrates that the random group assignment did not result in different prior programming ability between the groups. Therefore, any differences between the groups should be due to the `AIF` system.

7.5.1 Performance results(H1 - performance)

To measure performance, we compared scores of students who attempted the practice tasks. Figure 7.5 visualizes boxplots and averages for each condition's performance on each task. As shown in Figure 7.5, students using the `AIF` system have consistently *higher* practice task scores than that of the *Control* group.

We then used a linear mixed-effects model (as in Section 7.4.3), to compare combined performance on the practice tasks between the *AIF* group and the *Control* group. In Table 7.3, we use condition ($AIF = 1$, $Control = 0$) and *TaskLevel*[6] as independent variables in our model to predict students' performance as the dependent variable. The total number of observations is 146 instead of 150 (3 tasks * 50 students) since we excluded 4 students because of missing data.[7].

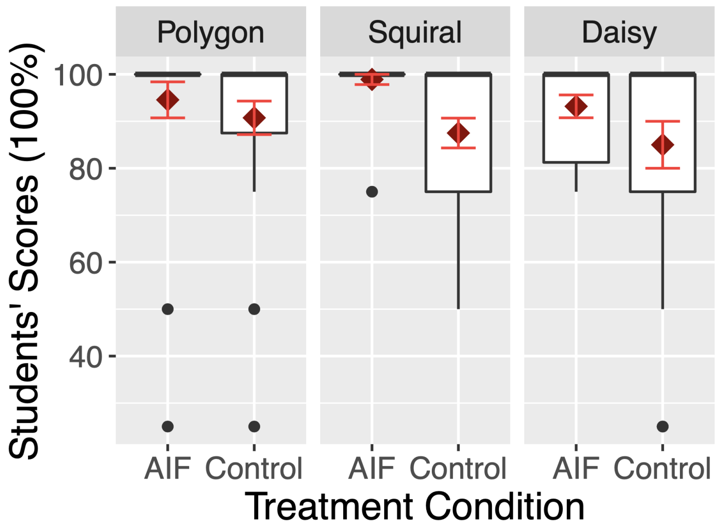


Figure 7.5: Student Practice Task Scores for AIF and Control Groups.

Model A (shown in Table 7.3) does not show a statistically significant impact of *TaskLevel* on students' performance. However, the model shows that the `AIF` system significantly improves students' performance. As shown in Table 7.3, the *AIF* variable has a significant

[6]'TaskLevel' represents the categorical order of the programming tasks with 2 indicator variables to represent distinct categories [Skr09], where '0,1' is PolygonMaker, '1,0' is Squiral, and '0,0' is Daisy.

[7]Missing data includes 1 student in the *Control* group who did not attempt *Squiral* and 3 students who did not attempt *Daisy* (two of them are in the *AIF* group, and one in the *Control* group). We chose to exclude these students because the programming environment did not impact their score (i.e. their '0' score).

impact on performance ($p = 0.025$), and students using the `AIF` performed, on average, 8% points higher, compared to the *Control* students across all 3 practice tasks. Since the average grade was 89%, this represents a distinct improvement. **These results provide support for H1-performance, as the `AIF` system improves students' performance across the practice tasks**.

Table 7.3: Estimated coefficients (Standard Error) of linear probability models (LPM) and linear mixed-effects (LME) models predicting students' performance (i.e. grades) on programming tasks (Model A), and the likelihood that a student completed programming tasks (Model B), respectively.

	Model A	**Model B**
	Coeff (Std.Err)	Coeff (Std.Err)
Intercept	87.889 (2.263)***	0.666 (0.053)***
AIF	7.678 (3.329)*	0.201 (0.079)*
PolygonMaker	-2.56 (2.156)	-0.082 (0.054)
Squiral	-1.789 (2.141)	0.004 (0.054)
Observations	146	146

Significant codes ($p <$): + = 0.1, * = 0.05, ** = 0.01, *** = 0.001.

7.5.2 Completion Rates (H2 - completion)

The improvement in scores with `AIF` confirms H1, but does not help us understand *how* students benefited. We hypothesized that the `AIF` system would benefit students by 1) encouraging them to persist, and 2) helping them to catch mistakes, both of which would translate into higher task completion rates. Therefore, we measured students' willingness and ability to persist to task completion using the completion rate *i.e. a binary score of 1 for 100% score and 0 otherwise.* Grouping all the three tasks together, we use a linear probability model (LPM) with mixed effects to predict the likelihood that a student completed a task ('1' = completed, '0' = not completed, dependent variable), using the student's treatment condition and the *TaskLevel* as predictors (independent variables) as we described above in Section 7.5.2. As shown in Model B in Table 7.3, we find no effect of the *TaskLevel* on students' likelihood of completing a task ($p = 0.12, 0.93$). However, we found that the `AIF` system had a significant effect on completion rate ($p = 0.01$), and students using the `AIF` system were overall 20% more likely to complete a programming task than students in the

Control group. Looking at each task separately, we found that a higher number of students in the *AIF* group completed each task (91.3%, 95.7%, and 72.7%) than that of the *Control* group (74.1%, 57.7%, and 68%) in *Polygon Maker*, *Squiral*, and *Daisy* tasks, respectively. We see that the group using the `AIF` system was consistently more likely to complete each assignment. **These results provide support for H2-Completion, but also suggest the need for further investigation, as discussed in Section 7.5.6.**

Post-hoc Analysis: To understand how the `AIF` system promoted persistence, we examined students' progress over time to investigate if there is a relationship between the amount of time students spent programming and whether their submissions were complete. We hypothesized that the `AIF` system might help students complete the programming tasks correctly in two ways: 1) it might help students better understand the task objectives and whether/when they have completed them, leading to fewer *incorrect* submissions, and 2) it might encourage students to continue working to finish the programming task rather than giving up, leading to fewer *incomplete* submissions. To investigate these two outcomes, we visualized students' progress over time. We measured active time from when students began their first code edit until they exported their attempt, and excluded any time where students spent > 5 minutes idle (i.e. making no code edits).

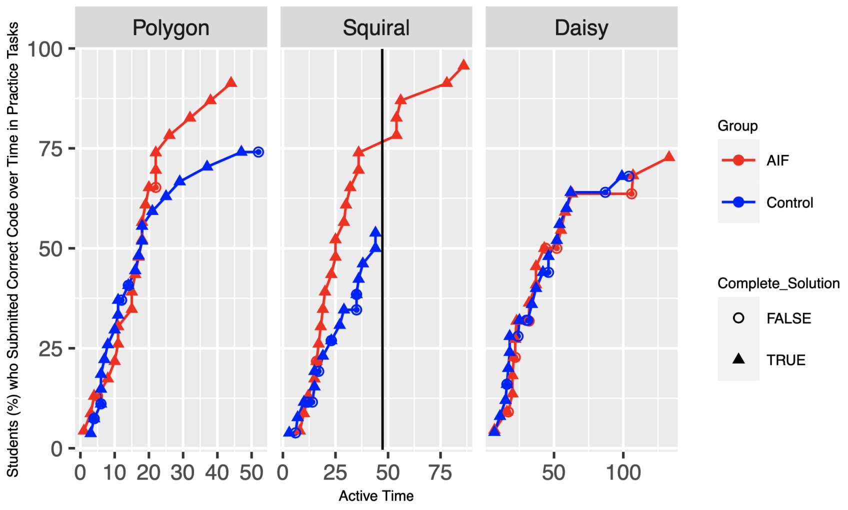


Figure 7.6: Percent of students who submitted correct code over time in practice tasks. The triangular points indicate complete submissions, and the circular points indicate incomplete submissions.

Figure 7.6 plots the percentage of all students in each group who had submitted a *complete, correct* submission (y-axis) as time *progressed* (x-axis). Each point represents one student submission, but only complete submissions (triangles) correspond to an increase in the y-value. First, in *Polygon Maker* and *Squiral* tasks, we see a gap suggesting that both groups were submitting code, but `AIF` group submissions were less likely to have errors. Additionally, in *Squiral*, we see that 22% of students (5 / 23) in the *AIF* group *continued* to work and submit correct code, even after all *Control* group students had submitted (solid black line), suggesting increased persistence to submit complete code. This is consistent with the fact that students in the *AIF* condition spent more time on each task, spending 2, 9, and 4 minutes more than that spent by the *Control* group, in *Polygon Maker*, *Squiral*, and *Daisy*, respectively. These exploratory results suggest that students in the *AIF* condition were not necessarily *more efficient* at solving each problem, but instead, they may have invested the necessary time, which paid off in the form of increased task completion rates.

7.5.3 Learning results (H3 - learning)

We hypothesized that students who received adaptive immediate feedback on their progress and completed more correct tasks would therefore learn more and perform better on subsequent programming and assessment tasks. We first measured learning by investigating student transfer performance on HW3 *BrickWall*, where neither group used the `AIF` system nor received hints during programming this task. We found that students in the *AIF* group completed 5.13% (M = 91.6%; Med = 100; SD = 14.43) *more* subgoals than the *Control* group (M = 86.54%; Med = 100; SD = 17.65) in *BrickWall* task. While a Mann-Whitney U Test does not show a significant difference ($p = 0.31$), perhaps due to a ceiling effect (most students performed perfectly in both groups), there was a medium effect size (Cohen's $d = 0.34$)[8]. We also compared the proportion of students submitting correct attempts in *BrickWall*. We found that 71.43% of the *AIF* group submitted correct attempts, which is higher than that submitted by the *Control* group (57.7%), but a Fisher's exact test shows that this difference was not significant ($p = 0.37$). It is also worth noting that the 6 *slowest* students to turn in *BrickWall* were all in the *Control* group (22% of the group), taking longer than 77 minutes (Figure 7.7, solid black line) to submit the assignment, and only 50% of these (i.e. 3) were fully correct. While we see from the results that students using the `AIF` system performed better in the transfer task, and in less time than that of the *Control* group, **these results provide inconclusive evidence to support H3-learning, i.e., the `AIF` system**

[8]We report statistical tests, along with effect sizes, to provide a complete picture of the results, and help the reader better interpret the effect of the `AIF` system. However, because each condition has a small sample size (i.e. <30), these tests are only likely to detect large effects and should be interpreted cautiously.

did not significantly improve students' performance in a transfer task.

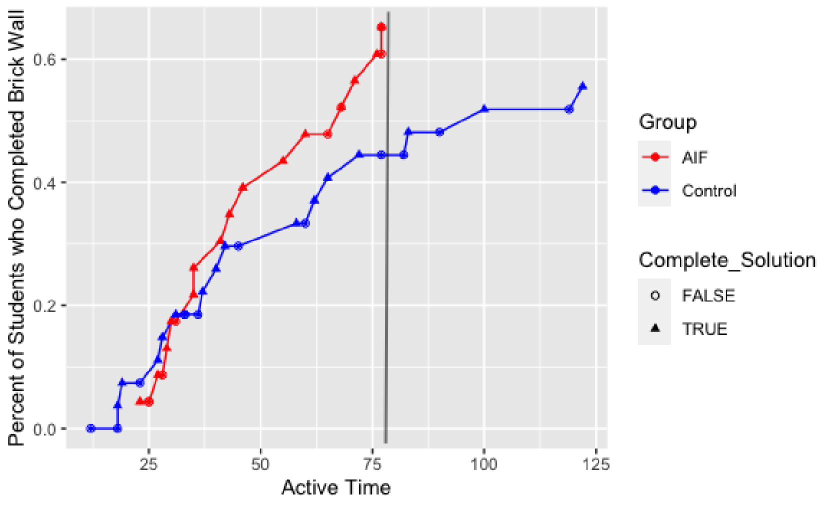


Figure 7.7: Percent of students who submitted fully correct code over time in Brick Wall homework. The triangular points indicate complete submission, and the circular points indicate incomplete submission.

Our second measure of learning is comparing students' scores in the post-test (described in Section 7.4.3). We found that 48 students took the post-test; 21 in the *AIF* group, and 27 in the *Control* group. On average, we found that the *AIF* group solved more questions correctly (M = 89.11%; Med = 85.71%; SD = 10%) than the *Control* group (M = 83.07%; Med = 85.71%; SD = 15.87%). A Mann-Whitney U test showed that this difference was not significant ($p = 0.23$); but had a medium effect size (Cohen's $d = 0.44$). Taken together, **these results suggest that the `AIF` system does not harm learning (i.e. by making the tasks too easy), and *may* provide a benefit to learning if a larger study were performed.**

7.5.4 Case Studies

We provide here two case studies for students *Sam* and *Em* to illustrate *how* a student can use the `AIF` system, and reasons *why* the `AIF` system may have increased students' performance and motivation to submit complete programming tasks. We created these case studies by replaying students' log data for students when using the `AIF` system and students in the *Control* group, looking for differences in performance. Case study *Sam*, represents a student using the `AIF` system who keeps programming when the system tells them their subgoals are incomplete; however, their code output *looks* complete. Case study *Em* represents a student in the *Control* group who submits incomplete code with missing components that never occurred for students in the *AIF* group.

Case Study Sam

This is a case study of student *Sam* who was assigned to the *AIF* group and submitted a complete attempt of *Squiral* task after 19 min of work. This case study presents how students can use the `AIF` system, and how it might have helped them complete a task. Sam started by creating a custom block named 'draw squiral' nested with a move and turn blocks, where the `AIF` system showed them 33%, 16%, and 18% increase in objective 1, 2, and 4, respectively. *Sam* kept working for 3 more minutes, running their code 29 times, without notable progress and the system popped up an encouraging pop up message: 'keep going'. *Sam* then created a 'length' variable and used it in the move block, and immediately the system noted they completed subgoal 3. Sam asked for a hint that suggested using a 'change _ by _' block; however, *Sam* used a different block that has a similar color to the 'change' block, and no increase in its corresponding subgoal took place. *Sam* then clicked on objective 1, where the system marked it in yellow and presented more descriptive text of objective 1 (as described in Section 7.3). Sam then double clicked on objective 3 and marked it as complete, and then clicked again on objective 1. Afterwards, *Sam* asked for a couple of hints, and followed them immediately, leading to completion of objectives 2 and 4, which the student then marked as complete. The system by default then turned objective 1 to yellow, as it is the remaining incomplete objective. While *Sam*'s code was producing the correct output for *Squiral*, the code did not use an input parameter for the number of rotations in the 'draw squiral' custom block. Sam then spent 3 more minutes to successfully add the parameter, and the system popped up a message: 'You finished the 4 objectives!'. Finally, *Sam* ran their code with several inputs and then submitted their code.

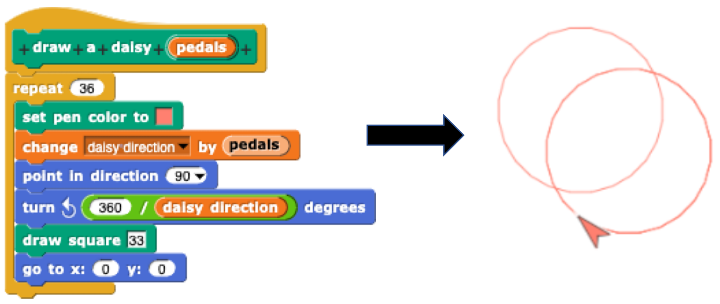
Sam's log data showed that their code was often able to generate the correct output, but had missing components needed for full credit. However, *Sam* persisted until all the objectives were complete. This may have been due to the `AIF` system's tracking of *Sam*'s progress, and presenting how far they had progressed in each objective. In contrast, in the *Control* group, we found that 91% of students who had incomplete code had submitted code that seemed to generate the correct output but was missing components of a correct solution. This might be due to their lack of understanding of some programming concepts, like how to use function parameters, or students relying on assessing the output generated by their code. From an instructors' perspective, these seem like natural mistakes for novices to make. Overall, *Sam*'s case study shows how helpful the AIF is in prompting students to self-assess their code and helps them determine whether it meets the assignment's requirements; which led them to have higher performance and completion rates than that of the *Control* group.

Case Study Em

This is a case study of student *Em* who was assigned to the *Control* group, and therefore they did not use the AIF system. *Em* spent 31 minutes actively programming the *Daisy* homework, and ended up submitting incomplete code. At the beginning of solving *Daisy*, *Em* started by creating a procedure to 'draw a circle', and at this moment, if they *had had access* to the AIF system, it would have shown that they completed the objective to 'draw a circle'. *Em* then kept working ∼20 minutes and requested 9 hints while trying unsuccessfully to complete the 'draw daisy' procedure. *Em* then deleted *all* their correct code, which is a common behavior among novices when they are *uncertain* about the correctness of their code [Don19]. At this moment, if *Em* had access to the AIF system, it would have shown that their progress in one of the objectives went down to 0%, which might have encouraged them to restore their deleted code. *Em* then stopped working for 2 hours, and when they returned, they made a couple of edits and finally turned in incomplete code as shown in Figure 7.8 after a total of 34 minutes. If *Em* had been using the AIF system, they would have seen that their last edits improved their progress ∼ 15% and that they were overall 65% close to the correct solution. In contrast, students using AIF for *Daisy* spent an average of 44 minutes, and every student using AIF achieved at least 75% progress on this task (as shown in Figure 7.5). These results suggest that providing only hints to students are not enough; reinforcing the finding by Mitrovic et al. that "a tutoring system that teaches to student errors can be improved by adding a capability to teach to their successes as well [Mit13]."

7.5.5 Student perceptions

To gain insights into the students' *perspectives* on the AIF system, we examined students' responses to an open-ended question. Only 29 students took the optional survey, 20 in the *AIF* group and 9 in the *Control*. We performed thematic analysis (described in Section 7.4.3) to analyze students' responses in the open-ended question: "how or when were the objectives' list with progress bars helpful or less helpful in Snap?" Our analysis resulted into



```
+draw+a+daisy+(pedals)+
repeat 36
    set pen color to
    change daisy direction by pedals
    point in direction 90
    turn ↺ (360 / daisy direction) degrees
    draw square 33
    go to x: 0 y: 0
```

Figure 7.8: Submitted code by student Em and its corresponding output on the right.

two main themes: *mechanisms* by which the AIF system was helpful, and *limitations* in its design, which we report below. Note that when reporting quotes from students' responses, we put an anonymous student ID preceding their quote (e.g. [S1] means student 1).

Mechanisms by which the AIF system was helpful: This theme highlights the extent to which students' responses align with our second hypothesis (H2-completion) about the AIF system. Overall, 12 out of 20 students made open-ended comments about how the AIF system was helpful. Four out of 20 students identified that the system helped them to keep track of their progress: "[The objective progress bars] let me know if I'm headed in the right direction or not" [S2]. Another student stated: "It lets me know what is needed to be done so I am not lost making mistakes and unnecessary edits" [S4]. Both of these comments show that the AIF system may reduce students' uncertainty and improve performance. Other students (3 out of 20) appreciated the feedback on correctness: S11 saying "... it helps me see how close I am to... getting the code right." These responses align with our hypothesis that providing students with immediate feedback on their progress motivates them to persist to completion.

Limitations in the AIF design: Students' comments also revealed trade-offs in the design of adaptive feedback, and ways the AIF system could have been improved. Five (out of 20) students described the progress bars as "not always correct" [S8], saying "sometimes it gets stuck on 95%" [S10]. Students likely forgot or failed to understand the provided explanation that the system progress rates were based on similarity to prior solutions, and the max was set to 95% to remind them of their responsibility to self-assess their code before submission. Another limitation is that the system provides objective labels that fit the data-driven subgoals; however as one student noticed, "the [objectives'] progress bar definition seemed vague and arbitrary at times [S6]". These limitations suggest that students may take the numeric scores and objective descriptions more literally than intended.

In sum, a majority of the students described the system as helpful because it helped them keep track of their progress or validate their code, two important factors that help reduce uncertainty and provide motivation to persist. However, some students felt the system did not always match their expectations. The nature of the automated feedback and the open-ended programming tasks make perfect, immediate feedback impossible; however, we are encouraged that this limitation did not harm students' system use, performance, or learning. Furthermore, the system incorporated a visual and numeric progress assessment, providing a tangible reminder to students that they should self-assess their progress while programming.

7.5.6 Discussion

Overall, our results provide consistent evidence that our `AIF` system benefits students through its formative feedback that was designed to be corrective, immediate, specific, and positive. We now discuss each of our hypotheses:

H1 - Performance – Supported: We found that `AIF` led to increased overall performance on tasks, reflected by more assignment objectives completed. These results show that the `AIF` system has the potential to increase students' grades *without* giving away any of the solution code (e.g. like hints may do). This performance increase is important because in introductory CS courses, students frequently make frequent, negative self-assessments of their programming ability that may lead students to leave the field [Gor20]. `AIF` may help, not only by giving the student a sense of progress and accomplishment during programming, but also by securing more positive feedback from instructors when turning in more correct code. While this work did not measure students' affective outcomes, our own prior work suggests that adaptive immediate feedback can lead to increased intentions to persist in computing [Mar20a].

H2 - Correctness – Supported: Our results suggest that, across tasks, students with `AIF` were more likely to turn in fully complete and correct code. Our case studies, surveys, and log data analyses suggest two likely mechanisms for this improvement. First, students who struggle on challenging independent homework tasks sometimes give up, like student *Em* did, without completing the task. Those with `AIF` may have been encouraged by the increasing progress bars and positive pop-up messages that show students they are "headed in the right direction [S2]". As shown in Figure 7.6, students in the *AIF* group continued to work on the *Squiral* and *Daisy* homeworks, turning in correct code long after the *Control* students. Since the code was ultimately correct, it suggests that this extra time was well spent.

Second, some students may have been unaware that their code failed to meet assignment objectives, as we discussed in *Sam*'s case study, leading them to turn in erroneous code. The `AIF` system's automated assessment can highlight this incorrect code, giving students the opportunity to fix it. Figure 7.6 shows that students *do* fix these errors, as evidenced by the gap between the percentage of students turning in fully correct code on the *Polygon Maker* and *Squiral* tasks over time. In both cases, our results show not only that the `AIF` system can encourage students to address these errors, but that students are often *capable of fixing them*, as long as they are made aware of them.

While our data cannot show whether students gave up intentionally or missed errors in their code, or how precisely the `AIF` changed their behavior, our hypothesis aligns with prior research and theory on the efficacy of formative feedback. Using the `AIF` system, students received immediate specific feedback, then corrected errors and turned in more complete code, which are behaviors that align both with instructor desires, and learning theories to improve learners' outcomes [Thu13]. We also note that even in the rare occasions when the system's feedback is inaccurate, as noted in Section 7.5.5, prompting students to engage in self-regulated learning skills, such as progress monitoring and self-assessment, can improve students' learning [Lok16].

There are two explanations that may have led to improved completion (more scores of 100%) and correctness (i.e. higher scores) with `AIF`. First, the `AIF` system simplifies the learning process by dividing the task into a set of subgoals, and provides corrective immediate feedback on each. This task breakdown might have reduced students' cognitive load, and based on cognitive load theory, this could lead to an improvement in their scores [Paa03]. Another interpretation is that, because students can see a change in their progress when they add or delete correct blocks, they might be more cautious not to delete correct code, which is an influence on students' 'behavior', leading to higher scores. This was also suggested by students' responses in the post-survey when one student said: *"It lets me know what is needed to be done so I am not lost making mistakes and unnecessary edits* [S4]."

H3 - Learning – Inconclusive: We found suggestive evidence that students with `AIF` performed better on a subsequent transfer task *without* hints or feedback, as well as on a post-test. However, neither of these effects were significant. It is possible that our sample size of 50 consenting students was too small to detect this difference. It is also possible that the primary benefit of `AIF` is on the task where it is given, not on learning. However, we argue that if assignments are well-designed, encouraging students to complete more of the task objectives should reasonably have some benefit to learning, as students who persist are exposed to more learning content. In addition, prior work suggests that increased self-efficacy is associated with increased subsequent performance [Lis16], and it is possible that students receiving positive feedback from `AIF` and from improved task performance, gained self-efficacy as well, though we did not measure this.

Limitations: This work has three limitations: survey data, missing data, and possible confounds. First, only 20 out of 23 students in the *AIF* group took the post survey and therefore our qualitative results may not be representative of all students in the *AIF* group. Second, while we suggest that the one mechanism by which the `AIF` system improves student performance and learning is by creating opportunities for positive self-assessment, we did not use surveys to measure self-assessment. This was due to some technical issues with the deployment of the surveys. However, students' responses in the post survey indicate that the `AIF` system may have promoted student self-assessment during programming, and our results in prior work show that such formative feedback improved students' intentions to persist in CS [Mar20a]. We plan to measure self-assessment and self-efficacy in future work.

While we argue that our results are largely consistent with our hypothesis, there are other possible confounds. Students in the *AIF* group may have had more prior programming experience, though we found no evidence of this in our pre-assessment task. We also note that all students had access to automated hints, and hint usage may have affected our results. However, we found no large or significant differences in the number of hints requested across groups. It is possible that the hint complements the `AIF`, increasing its effectiveness, but it is also possible that the additional help in *both groups* actually diminished our ability to detect differences between the groups, e.g. by creating a ceiling effect on assignment performance. Lastly, we note that increased student persistence to submit complete solutions has a trade-off that it requires additional student time, which students may not appreciate. We did find that the *AIF* group spent somewhat more time on homeworks (though the difference was not significant), but this also seemed to pay off in increased performance.

7.6 Conclusion

The `AIF 3.0` system uses a hybrid data-driven model to generate automated formative feedback on assignment subgoals, during programming. Its adaptive progress feedback was designed to align with theories about effective feedback for learning. In this paper, we evaluated the `AIF 3.0` system over three weeks and multiple assignments in an authentic computer science non-majors classroom setting. Our results showed that the `AIF` system improved student scores across all programming tasks, and increased the number of students' code submissions that were completely correct. Our qualitative analysis showed that students perceived the system to be helpful because it let them track their progress and prevented them from wasting time on unnecessary edits. Our case studies demonstrated how a student without adaptive feedback may give up and submit incomplete tasks; and a contrasting case that shows how the `AIF` system can resolve this behavior. In future work we will continue to refine our feedback and interfaces, and use surveys to measure the impact of the `AIF` system on students' self-assessment and self-efficacy.

CHAPTER

8 CONCLUSION

In this final chapter, I revisit the general research questions of this thesis, discuss studies' findings that answer each RQ, and summarize the thesis' contributions. In the end, I also present possible directions for future work. As a reminder, the goal of this dissertation is to explore best practices for automated programming support, specifically for next-step hints and formative feedback, that can improve students' cognitive and affective outcomes.

8.1 Research Questions

8.1.1 RQ1: How do specific design choices (i.e. textual explanations, self-explanations, and hint displays) impact automated next-step hints to improve students' performance and learning?

I organize this answer by talking about each design choice separately, and how and why it improved (or did not improve) the effectiveness of next-step hints on students' outcomes during programming.

First, next-step hints can get students unstuck; however, hints alone do not teach the student what the problem is, and may also be hard to interpret [Mar19a]. Therefore, students can easily dismiss it, leading to no positive effect on performance or learning. In Chapter 3, I addressed this limitation by adding "textual explanations" that describe (1) the purpose of the hint, (2) the problem it is solving, and (3) how the student can apply it. These brief textual explanations can help learners to connect the hint with their code, making the hint more convincing to be applied (i.e. followed) and, therefore, might lead to an immediate increase in students' performance. I found in a qualitative study in Chapter 3 that all students preferred having textual explanations with next-step hints, and that textual explanations succeeded in increasing the interpretability of next-step hints. In a quantitative study with Mturk crowdworkers in Chapter 3, I found that next-step hints with textual explanations increased students' performance in programming tasks, in comparison with students who did not receive hints at all [Mar19b]. In addition, I found no evidence that hints with textual explanations improved students' learning. This can be probably because textual explanations are only useful when students need help to understand the hint [Riv17a; Pri21; Mar19a], but can also be simply ignored when they are not needed as evidenced in prior work that students often do not read textual hints [Mar19a; Fos15]. Another possible reason is that textual hints are more beneficial when students have some prior knowledge so that they can easily understand the hint, but not for novices at the beginning of a course [Mar19a]. While I intentionally design the textual explanation to be easily understood by novices, these results suggest the need for more investigations to simplify the hints' textual explanations.

The second challenge in next-step hints is that there is little evidence that they can increase students' learning in programming, probably because hints are "bottom-out", saying only what to do but not why [Har10; Pri17e; Riv17b]. Therefore, I complimented next-step hints with self-explanation prompts (my second design choice) to encourage students to think and reason about the hints; which is an effective instructional learning activity as informed by the KLI framework [Koe12]. In Chapter 3 and 5, I found evidence that next-step hints with textual explanations and SE prompts increased learners' performance in programming tasks. Also, in Chapter 3, I found that these hints increased students' learning as evidenced by an increase in their performance in future isomorphic tasks without hints. This effect might be because prompting students to self-explain a hint, makes them pause, read the hint and reflect on it (or simply, make students "make sense" of the hint [Koe12]) that resulted in improvement in students' performance and learning. In addition, because these prompts increased the time students spent viewing the hint (as found in [Mar19b]), students become more aware of the hints, and might be more convinced to follow them leading to increased performance (i.e. increased scores in programming tasks). One can argue that this effect of hints was seen when *both* textual explanation and SE prompts were augmented to hints. While I did not quantitatively compared having next-step hints with SE prompts only versus having the two design features together with hints, my investigations of participants answers to SE prompts (Chapter 3 and 5) reveal that textual explanations helped students better interpret the next-step hints and these explanations may have supported students to reflect on the hints (as noted in students' self-explanations), leading to an increased learning.

The third design choice is not particularly related to next-step hints, it is more about how we can support students to productively seek help; which is challenge identified in prior work [Ale06; Ale03; Woo99]. This is a challenge because if hints can lead to learning, this will happen *only* when students do not avoid or abuse hints. To improve students' help-seeking behavior, in Chapter 4, I first constructed a taxonomy for unproductive help-seeking behavior in block-based programming environments. Based on this taxonomy, prior empirical studies, and other theoretical perspectives, I then developed the data-driven hint display algorithm that decides when the student is in need of help, by simply comparing their progress (i.e. time taken to complete a subtask) to the time taken by previous students. Giving students a hint at the time when they struggle might be a good motive for them to open the hint. Also, because students might not know/notice the option to ask for a hint, I displayed hints by popping a flashing hint button next to the student's code to grasp their attention, but not in a frustrating way. Overall, these are the three features of the data-driven hint display (detailed in Chapter 4): (1) it gives students proactive hints only when they are not progressing, (2) it grasps students' attention that a hint button exists, and (3) it gives students the control to open the hint or not. In Chapter 4, I found that providing hints using the data-driven hint display algorithm prevented students from abusing help and were less likely to unproductively avoid it. This overall shows an improvement in students' help-seeking behavior. I also found suggestive evidence that providing hints only at times when students are struggling, allows students to process hints better, which leads to improving their programming performance in a future task without hints. I argue that Chapter 4 presents a preliminary design of a hint display in a programming environment – having the advantages of both on-demand and proactive hint displays that prior literature has conflicting findings about the effect of each.

8.1.2 RQ2: What is the impact of enhanced next-step hints during block-based programming, on students' performance, and learning?

As a reminder, the enhanced next-step hints are hints augmented with textual explanations, self-explanation prompts, and presented to students using a data-driven hint display (discussed in Chapter 4). In Chapter 5 I conducted a controlled study in an authentic classroom to evaluate the impact of enhanced next-step hints on students' performance, learning and perceptions. I found that the enhanced next-step hints improved students' overall programming performance in in-class tasks, and improved programming efficiency in homework tasks. However, I found inconclusive evidence that the enhanced next-step hints improve students' learning in an assessment programming task, or in a post-test. Possible interpretations for this finding is discussed in Chapter 5 (Section 5.4.1). In Chapter 3, I found, in a laboratory study, that enhanced hints improved students' learning in isomorphic programming tasks. Below in Section 8.1.4, I discuss how consistent the impact of enhanced next-step hints across multiple learning contexts.

8.1.3 RQ3: What is the impact of the formative adaptive immediate feedback (AIF) system on students' affective and cognitive outcomes?

As a reminder, in Study 4 and 5, I developed the adaptive immediate feedback (AIF) system, that provides students with formative feedback that is specific, immediate, corrective and positive. The *AIF* system presents students with subgoals of a given programming task, attached with human-authored explanations, and provides corrective and positive feedback on each, in real-time, while students are programming. Also, the *AIF* system provides praise and motivational positive feedback messages when students spend a long time without progress, or when they complete a subgoal, or when they fix a subgoal that they have already completed. In Chapter 6, I found that such formative feedback improved three affective outcomes for students: (1) engagement, (2) intentions to persist in CS, and (3) motivation. First, I found an increase in students' engagement with the programming task as evidenced by a significant reduction in the idle time spent during programming. In other words, participants working with the *AIF* system kept active in programming, without spending much idle time, as compared to students not using the *AIF* system. Second, I found a significant increase in students' self-perceptions of their intentions to persist in CS after using the *AIF* system. Therefore, systems that can provide students with effective formative feedback during programming may have the potential to improve retention for students in CS courses. Third, I found an improvement in students' motivation to keep working in programming tasks as measured qualitatively from students' responses in interviews and surveys. Additionally, in Chapter 7, I found that the improved version of the *AIF* system (i.e. *AIF 3.0*) encouraged students to submit more complete/correct programming assignments, which suggests increased motivation and persistence to complete programming assignments. This is one of the first studies to show a significant impact of a feedback system on students' affective outcomes similar to human tutoring feedback.

In terms of students' cognitive outcomes, in Chapter 6 and 7, I found that both versions of the *AIF* system improved students' performance and task completion rates. This positive cognitive effect might be due to the following. First, dividing a task into a set of subgoals, and providing immediate corrective and positive feedback on each, might have reduced students' cognitive load, which led to an improvement in their scores, as suggested by the cognitive learning theory (discussed in Chapter 7) [Paa03]. Second, providing students with immediate positive and corrective feedback might have reflected the "uncertainty reduction" mode of learning [Mit13; Fos15], where reducing students' uncertainty can increase their motivation to complete programming tasks leading to an increase in their performance. Third, for all students, whether using the *AIF* system or not, the iSnap programming environment provided them with enhanced next-step hints as well. While I did not find a difference in students' hint usage across the treatment conditions, the results of the studies in Chapter 6 and 7 might be because a tutoring system is more helpful when it teaches students both their errors and successes [Mit13]. The fourth and final interpretation of these results is that the increase in students' affective outcomes, like motivation and engagement with the task, might have led to increased students' performance since they spent more time attempting to complete the task. This also aligns with prior human tutoring studies that show that improving students' affective outcomes can predict students' grades and success [Lep93; DE09].

One major limitation of this work is that the way I provided students with formative feedback was through a list of subgoals that might restrict students' creativity and limit them from trying other approaches. However, I argue that this method is still beneficial due to the following: First, the subgoal list provided by the expert-authored *AIF* system (in Chapter 6), is driven from the programming task instructions given by the instructor, and therefore, possibly all students will just follow the given instructions. Second, the programming tasks in Chapter 6 are short ones, and there is a limited number of possible solutions. Third, this feature can not only be developed in a feedback system, but teachers and instructors can use a similar design of a subgoal list as a static support to students, that I have demonstrated its effectiveness in prior work (not added in this thesis) [Mar21b]. Fourth, the *AIF 3.0* system (using the hybrid data-driven algorithm), generates its subgoal list based on historical data that captures several students problem solving strategies, and I only added human-authored explanations to each guided by the programming assignment instructions given by the instructor. Fifth, and last, each of these data-driven subgoals can be solved using multiple approaches, which did not hinder students' creativity in solving programming tasks.

8.1.4 RQ4: How consistent are the impacts of next-step hints/formative feedback across different populations and learning contexts?

In terms of the enhanced next-step hints, I found similar effects on students' cognitive outcomes across populations and learning contexts. In particular, in a *60-min online study* with recruited *Mturk crowdworkers*, I found that the enhanced hints improved their performance in *short programming tasks*, and improved their learning in later isomorphic tasks. In a *4-week authentic classroom study* with *undergraduate* students across *several programming tasks* with varying complexity, I found that the enhanced next-step hints improved students performance in in-class tasks, and I found an inconclusive increase in students learning in an assessment task. In both studies, participants preferred having next-step hints with textual explanations rather than next-step hints alone, or next-step hints with SE prompts, and reported conflicting opinions about the usefulness of SE prompts.

These results can overall generalize that enhanced next-step hints are well-perceived by students, can improve learners' immediate performance, performance in future tasks without hints, but not learning in a post-test. I discuss several interpretations for this result in Section 7.5.6 in Chapter 5. Therefore, it remains open whether enhanced automated next-step hints can improve learning in post-tests. Furthermore, what is difficult to generalize is whether these results represent all programming problems. This is because, in Chapter 5, I found that hint quality affects how students used hints, which might have affected their programming performance. What is obvious from Study 3 and Study 5, is that enhanced next-step hints are effective in short programming tasks, but their effectiveness in more complex ones are highly related to the hints quality.

In terms of formative feedback, I found that the *AIF* system using *expert-authored rules* in *short programming tasks* has a marginal significant effect on improving *high-school* students programming performance, and learning in an assessment programming task during a *one-day summer camp* (detailed in Chapter 6). However, I did not analyze the impact of the *AIF* system on students' learning on a post-test because I found that many participants did not submit their tests, which is one of the laboratory studies' challenges. In Study 5, I found that the *AIF 3.0* system using a *hybrid data-driven algorithm* for *several programming tasks* with varying complexity improved *undergraduate* students' programming performance in a *4-week authentic classroom study* (detailed in Chapter 7). Also, in Study 5 I found inconclusive evidence that the *AIF 3.0* system improved students' learning in an assessment programming task, and in a post-test. The case studies in both studies show that participants have similar experiences, where they were engaged with the programming task, and motivated to complete it, when using the *AIF* system. Together, I suggest that the impact of an automated formative feedback on students' cognitive outcomes is confirmed (i.e. can improve performance, and *may* improve learning) across multiple populations and learning contexts. In terms of affective outcomes, I was unable to compare the effect I found in the camp study (Chapter 6) with the classroom study (Chapter 7) due to classroom technical problems that prevented me from collecting pre and post students' self-perceptions surveys (e.g. survey about students' intentions to persist). However, students' responses in open-ended surveys about the usefulness of the *AIF* system were very similar across both studies.

While only few studies in CS education seek to replicate existing work, I argue that without such replication, our understanding and trust of the effect of automated support would be incomplete and, maybe, incorrect. I, therefore, encourage the CS education researchers to open up a path for the community to replicate their novel contributions and validate how generalizable their work could be.

8.2 Contributions

The contributions of this thesis can guide future research directions for both the Computing Education Research (CER) and Human Computer Interaction (HCI) communities in designing, developing and evaluating better feedback systems in programming. Overall, these are the main contributions of this thesis work:

1. I designed and developed different hint features to tackle the limitations, and improve the effectiveness, of next-step hints on students' cognitive outcomes in programming. Such enhanced next-step hints are developed for multiple open-ended programming tasks, with varying complexities, in a block-based programming environment. In addition, these enhanced next-step hints have been deployed in the iSnap programming

environment and have been proven to be useful for students across several semesters in two CS0 classrooms at North Carolina State University, and it can continue to be used in the future as well. Overall, this work shows evidence on how to design next-step hints to be more effective in programming classrooms.

2. I introduced a taxonomy of students' unproductive help-seeking behaviors in block-based programming. This taxonomy is different because it defines specific unproductive help-seeking behavior categories, which make it applicable for researchers to design programming environments' interfaces that can deter this behavior. In addition, this taxonomy can also inform classroom instruction and guide teachers on how to improve their students' help-seeking behavior.

3. I designed and developed two versions of an adaptive immediate feedback system (*AIF*, and *AIF* 3.0) at various points in development. The *AIF* system is based on expert-authored autograders, and the *AIF* 3.0 system is based on a hybrid data-driven algorithm. The *AIF* system is the first feedback system that offers all forms of effective formative feedback for multiple open-ended block-based programming tasks with varying complexities. I integrated the *AIF* system with the iSnap programming environment and deployed it for use in Summer camps and a CS0 course at North Carolina State University. The *AIF* system has been able to not only improve students' cognitive outcomes, but also their affective outcomes, which is a preliminary finding in this thesis. The usefulness of the *AIF* system shows its potential to be used in future camps and semesters as well. In addition, other researchers are making more investigations on how its data-driven feedback impacted students' programming behavior, and there are likely possible directions in integrating machine learning mechanisms to improve the quality of its generated feedback.

4. I conducted empirical controlled studies to evaluate the impact of the enhanced next-step hints and *AIF* systems on students' cognitive and affective outcomes. These empirical evaluations are one of the first that leverage multiple assessment mechanisms to evaluate the impact of automated support on students' outcomes, including multiple programming tasks, transfer tasks, post-test, surveys, and interviews. Also, the classroom studies in this thesis are particularly important because they show classroom challenges (such as pair programming instructions) and I present possible solutions and study methodologies that can encourage future research in conducting more authentic classroom studies.

5. I explored how my findings generalize (i.e. replicated) across different populations, and multiple learning contexts. While replication is a scientific method that either decreases false positives or increases confidence that the result actually exists, there is a lack of reproducibility in CS education research. In this thesis I conducted multiple studies, in both laboratory and classroom settings and demonstrated which automated support effects were consistent and which were not, and suggested interpretations for these results (See Chapter 5 and 7).

8.3 Future Work

This work opens up several directions for future work. First, there is still room to improve next-step hints quality, not just by improving the quality of the data or adding more design choices, but by considering the specifications of the programming task that hints are created for. This improvement could be with respect to the hint *display*. For example, for tasks with multiple custom blocks, how can a system provide hints *relevant* to the custom block that the student is currently working on? One suggestion could be to provide students with a panel of hints, and the student can scroll until they open the hint they are looking for. Another suggestion is to provide hints, through the *AIF* system, relevant to the subgoal that the student is currently working on. While I attempted the latter in my research work, there are some technical issues, but it would be worth studying the impact of providing hints on the specific subgoal a student is working on, versus providing a random hint.

Second, this work has been attempted in one programming language, and it is worth extending to other programming languages as well. In one of my prior work collaborations, not added in this thesis, I evaluated next-step hints with textual explanations in a Python learning environment [Pri21]. However, the preliminary results show a different effect than that I found in block-based programming. While this might be due to the different visualization of how next-step hints were displayed; there is plenty of work to improve next-step hint design choices in text-based languages as well; and worth attempting to use my suggested design choices in text-based programming environments and investigate their effect.

Third, through my research work, I found that using the data-driven feature detection algorithm for more complex tasks (like games) is a hard task. This is due to the multiple ways, and noisy code blocks, used by students in prior semesters; leading to the generation of multiple (~ 100) correct features that are even hard to interpret by human experts. One way to solve this challenge is by limiting the data used to generate features, or use golden standard students' solutions instead. In future work I hope to attempt this idea, investigate the quality of the generated formative feedback for complex tasks, and then evaluate its impact on students' outcomes.

Finally, with the goal of improving students' learning, I am planning to design and develop instructional tools that teach students how to decompose a programming task into subgoals, instead of providing them with a subgoal list. Prior work shows that allowing students to decompose a task into subgoals decreases their cognitive load, and increases their ability to transfer learning across contexts; however, novice students lack the knowledge to do so. I hope to develop such instructional design and complement it with SE prompt as a way to not only support students' learning, but also maintain a long-term positive effect on students' programming performance and learning in introductory to programming classrooms. This is, and will continue to be, my ultimate goal and research interest during my research work in improving computing education technologies.

BIBLIOGRAPHY

[Aha16] Ahadi, A. et al. “Replication in computing education research: researcher attitudes and experiences”. *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*. 2016, pp. 2–11.

[Ale13] Aleven, V. “Help Seeking and Intelligent Tutoring Systems: Theoretical Perspectives and a Step Towards Theoretical Integration”. *International Handbook of Metacognition and Learning Technologies* **28**.January (2013), pp. 197–211.

[Ale01] Aleven, V. & Koedinger, K. R. “Investigations into Help seeking and Learning with a Cognitive Tutor”. *Papers of the AIED 2001 Workhop ’Help Provision And Help Seeking In Interactive Learning Environments’*. 2001, pp. 47–58.

[Ale03] Aleven, V. et al. “Help Seeking and Help Design in Interactive Learning Environments Vincent”. *Review of Educational Research* **73**.3 (2003), pp. 277–320.

[Ale06] Aleven, V. et al. “Toward Meta-cognitive Tutoring: A Model of Help Seeking with a Cognitive Tutor”. **16**.2 (2006), pp. 101–128.

[Ale09] Aleven, V. et al. “Scaling up programming by demonstration for intelligent tutoring systems development: An open-access web site for middle school mathematics learning”. *IEEE transactions on learning technologies* **2**.2 (2009), pp. 64–78.

[Ale16a] Aleven, V. et al. “Example-tracing tutors: Intelligent tutor development for non-programmers”. *International Journal of Artificial Intelligence in Education* **26**.1 (2016), pp. 224–269.

[Ale16b] Aleven, V. et al. “Help helps, but only so much: Research on help seeking with intelligent tutoring systems”. *International Journal of Artificial Intelligence in Education* **26**.1 (2016), pp. 205–223.

[And95] Anderson, J. R. et al. “Cognitive Tutors: Lessons Learned”. *The Journal of the Learning Sciences* **4**.2 (1995), pp. 167–207.

[And96] Anderson, J. “ACT: A simple theory of complex cognition.” *American Psychologist* (1996).

[Ang08] Angrist, J. D. & Pischke, J.-S. *Mostly harmless econometrics: An empiricist’s companion*. Princeton university press, 2008.

[Ash00] Ashwell, T. “Patterns of teacher response to student writing in a multiple-draft composition classroom: Is content feedback followed by form feedback the best method?” *Journal of second language writing* **9**.3 (2000), pp. 227–257.

[Ass17] Association, C. R. et al. *Generation CS: Computer Science Undergraduate Enrollments Surge Since 2006.(2017)*. 2017.

[Bak04] Baker, R. S. et al. “Detecting student misuse of intelligent tutoring systems”. *International conference on intelligent tutoring systems*. Springer, 2004, pp. 531–540.

[Bak11] Baker, R. S. d et al. “Towards predicting future transfer of learning”. *International Conference on Artificial Intelligence in Education*. Springer, 2011, pp. 23–30.

[Bal18] Ball, M. *Lambda: An Autograder for*
$snap$. Tech. rep. Electrical Engineering and Computer Sciences University of California at Berkeley, 2018.

[Ban77] Bandura, A. “Self-efficacy: toward a unifying theory of behavioral change.” *Psychological review* **84**.2 (1977), p. 191.

[Bec08] Beck, J. E. et al. “Does help help? Introducing the Bayesian Evaluation and Assessment Methodology”. *Proceedings of the International Conference on Intelligent Tutoring Systems*. 2008, pp. 383–394.

[Bec16] Becker, B. A. et al. “Effective compiler error message enhancement for novice programming students”. *Computer Science Education* **26**.2-3 (2016), pp. 148–175.

[Bec18] Becker, B. A. et al. “The Effects of Enhanced Compiler Error Messages on a Syntax Error Debugging Test” (2018).

[Beh11] Behrend, T. S. et al. “The viability of crowdsourcing for survey research”. *Behavior Research Methods* **43**.3 (2011), p. 800.

[Ben95] Benjamini, Y. & Hochberg, Y. “Controlling the false discovery rate: a practical and powerful approach to multiple testing”. *Journal of the Royal statistical society* **57**.1 (1995), pp. 289–300.

[Ben07] Bennedsen, J. & Caspersen, M. E. “Failure rates in introductory programming”. *ACM SIGCSE Bulletin* **39**.2 (2007), p. 32.

[Ben19] Bennedsen, J. & Caspersen, M. E. “Failure rates in introductory programming: 12 years later”. *ACM Inroads* **10**.2 (2019), pp. 30–36.

[Ber05] Bergin, S. et al. “Examining the role of self-regulated learning on introductory programming performance”. *Proceedings of the first international workshop on Computing education research*. ACM, 2005, pp. 81–86.

[Bir14] Biró, P. & Csernoch, M. “Deep and surface metacognitive processes in non-traditional programming tasks”. *2014 5th IEEE Conference on Cognitive Infocommunications (CogInfoCom)*. IEEE, 2014, pp. 49–54.

[Blu06] Blumenfeld, P. C. et al. *Motivation and cognitive engagement in learning environments*. na, 2006.

[Boy08] Boyer, K. E. et al. “Learner characteristics and feedback in tutorial dialogue”. *Proceedings of the Third Workshop on Innovative Use of NLP for Building Educational Applications*. Association for Computational Linguistics. 2008, pp. 53–61.

[Bra06] Braun, V. & Clarke, V. “Using thematic analysis in psychology”. *Qualitative research in psychology* **3**.2 (2006), pp. 77–101.

[Bru11] Bruin, J. *INTRODUCTION TO LINEAR MIXED MODELS*. 2011 (accessed April 6, 2020). URL: `https://stats.idre.ucla.edu/stata/ado/analysis/`.

[But98] Butler, R. “Determinants of Help Seeking: Relations Between Perceived Reasons for Classroom Help-Avoidance and Help-Seeking Behaviors in an Experimental Context.” *Journal of Educational Psychology* **90**.4 (1998), pp. 630–643.

[Cec11] Cech, E. et al. “Professional role confidence and gendered persistence in engineering”. *American Sociological Review* **76**.5 (2011), pp. 641–666.

[Che13] Chen, X. & Soldner, M. *STEM Attrition: College Students’ Paths Into and Out of STEM Fields*. Tech. rep. National Center for Education Statistics, Institute of Education Sciences, U.S. Department of Education, 2013.

[Chi09] Chi, M. T. “Active-constructive-interactive: A conceptual framework for differentiating learning activities”. *Topics in cognitive science* **1**.1 (2009), pp. 73–105.

[Chi89] Chi, M. T. et al. “Self-explanations: How students study and use examples in learning to solve problems”. *Cognitive science* **13**.2 (1989), pp. 145–182.

[Chi94] Chi, M. T. et al. “Eliciting self-explanations improves understanding”. *Cognitive science* **18**.3 (1994), pp. 439–477.

[Cho16] Choudhury, R. R. et al. “Scale-driven automatic hint generation for coding style”. *Proceedings of the International Conference on Intelligent Tutoring Systems*. 2016, pp. 122–132.

[Coh85] Cohen, V. B. “A reexamination of feedback in computer-based instruction: Implications for instructional design”. *Educational Technology* **25**.1 (1985), pp. 33–37.

[Con00] Conati, C. & VanLehn, K. “Toward Computer-based Support of Meta-cognitive Skills: A Computational Framework to Coach Self-explanation”. *International Journal of Artificial Intelligence in Education* **11**.1 (2000), pp. 389–415.

[Coo00] Cooper, S. et al. “Alice: a 3-D tool for introductory programming concepts”. *Journal of Computing Sciences in Colleges*. Vol. 15. 5. Consortium for Computing Sciences in Colleges. 2000, pp. 107–116.

[Cor01] Corbett, A. & Anderson, J. R. “Locus of Feedback Control in Computer-Based Tutoring: Impact on Learning Rate, Achievement and Attitudes”. *Proceedings of the SIGCHI Conference on Human Computer Interaction*. 2001, pp. 245–252.

[Cor10] Corbett, A. et al. “A Cognitive Tutor for genetics problem solving: Learning gains and student modeling”. *Journal of Educational Computing Research* **42**.2 (2010), pp. 219–239.

[Cor89] Corbett, A. & Anderson, J. R. “Feedback timing and student control in the LISP Intelligent Tutoring System”. *Proceedings of the Fourth International Conference on AI and Education*. 1989, pp. 64–72.

[Cor04] Correll, S. J. “Constraints into preferences: Gender, status, and emerging career aspirations”. *American sociological review* **69**.1 (2004), pp. 93–113.

[Cou00] Council, N. R. et al. *How people learn: Brain, mind, experience, and school: Expanded edition*. National Academies Press, 2000.

[Cus18] Custer, S. et al. “Toward Data-Driven Education Systems: Insights into Using Information to Measure Results and Manage Change.” *Center for Universal Education at The Brookings Institution* (2018).

[Dan12] Dann, W. et al. “Mediated transfer: Alice 3 to Java.” *SIGCSE*. Vol. 12. Citeseer. 2012, pp. 141–146.

[Dav18] Davis, D. et al. “Evaluating Crowdworkers as a Proxy for Online Learners in Video-Based Learning Contexts”. *Proceedings of the ACM on Human-Computer Interaction* **2**.CSCW (2018), p. 42.

[Den07] Denner, J. & Werner, L. “Computer programming in middle school: How pairs respond to challenges”. *Journal of Educational Computing Research* **37**.2 (2007), pp. 131–150.

[Den15] Denning, P. J. & Gordon, E. E. “A technician shortage”. *Communications of the ACM* **58**.3 (2015), pp. 28–30.

[Der08] Derks, D. et al. “Emoticons and online message interpretation”. *Social Science Computer Review* **26**.3 (2008), pp. 379–388.

[DE09] Di Eugenio, B. et al. “Towards explaining effective tutorial dialogues”. *Annual Meeting of the Cognitive Science Society*. 2009, pp. 1430–1435.

[Dia18] Diana, N. et al. “Data-driven generation of rubric criteria from an educational programming environment”. *Proceedings of the 8th International Conference on Learning Analytics and Knowledge*. 2018, pp. 16–20.

[Don19] Dong, Y. et al. “Defining Tinkering Behavior in Open-ended Block-based Programming Assignments”. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, 2019, pp. 1204–1210.

[Dun64] Dunn, O. J. “Multiple comparisons using rank sums”. *Technometrics* **6**.3 (1964), pp. 241–252.

[Dwe86] Dweck, C. S. “Motivational processes affecting learning.” *American psychologist* **41**.10 (1986), p. 1040.

[Eps02] Epstein, M. L. et al. “Immediate feedback assessment technique promotes learning and corrects inaccurate first responses”. *The Psychological Record* **52**.2 (2002), pp. 187–201.

[Esp10] Espasa, A. & Meneses, J. “Analysing feedback processes in an online teaching and learning environment: an exploratory study”. *Higher education* **59**.3 (2010), pp. 277–292.

[Fis12] Fishbach, A. & Finkelstein, S. R. “How feedback influences persistence, disengagement, and change in goal pursuit”. *Goal-directed behavior* (2012), pp. 203–230.

[Fos15] Fossati, D. et al. “Data Driven Automatic Feedback Generation in the iList Intelligent Tutoring System”. *Technology, Instruction, Cognition and Learning* **10**.1 (2015), pp. 5–26.

[Gar15] Garcia, D. et al. “The beauty and joy of computing”. *ACM Inroads* **6**.4 (2015), pp. 71–79.

[Ger16] Gerdes, A. et al. “Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback”. *International Journal of Artificial Intelligence in Education* **27**.1 (2016), pp. 1–36.

[Gor20] Gorson, J. & O’Rourke, E. “Why do CS1 Students Think They’re Bad at Programming? Investigating Self-efficacy and Self-assessments at Three Universities”. *Proceedings of the 2020 ACM Conference on International Computing Education Research*. 2020, pp. 170–181.

[Gup17] Gupta, R. et al. “DeepFix: Fixing Common Programming Errors by Deep Learning”. *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 1. 2017, pp. 1345–1351.

[Gus17] Gusukuma, L. et al. “Authoring feedback for novice programmers in a block-based language”. *2017 IEEE Blocks and Beyond Workshop (B&B)*. IEEE, 2017, pp. 37–40.

[Gus18] Gusukuma, L. et al. “Misconception-driven feedback: Results from an experimental study”. *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM. 2018, pp. 160–168.

[Hao19] Hao, Q. et al. “A systematic investigation of replications in computing education research”. *ACM Transactions on Computing Education (TOCE)* **19**.4 (2019), pp. 1–18.

[Har10] Hartmann, B. et al. “What Would Other Programmers Do? Suggesting Solutions to Error Messages”. *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 2010, pp. 1019–1028.

[Hau13] Hausmann, R. G. M. et al. “An evaluation of the effectiveness of just-in-time hints”. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* **7926 LNAI** (2013), pp. 791–794.

[Hel09] Hellevik, O. “Linear versus logistic regression when the dependent variable is a dichotomy”. *Quality & Quantity* **43**.1 (2009), pp. 59–74.

[Hic14] Hicks, A. et al. “Building Games to Learn from Their Players: Generating Hints in a Serious Game”. *Proceedings of the International Conference on Intelligent Tutoring Systems*. 2014, pp. 312–317.

[Jen01] Jenkins, T. “The motivation of students of programming”. *Proceedings of the 6th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2001, Canterbury, UK, June 25-27, 2001*. 2001, pp. 53–56.

[Joh16] Johnson, D. E. “ITCH: Individual Testing of Computer Homework for Scratch Assignments”. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 2016, pp. 223–227.

[Kar09] Karabenick, S. A. & Newman, R. S. “Seeking help: Generalizable self-regulatory process and social-cultural barometer.” (2009).

[Kar15] Kardan, S. & Conati, C. “Providing adaptive support in an interactive simulation for learning: An experimental evaluation”. *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 2015, pp. 3671–3680.

[Kat06] Katz, S. et al. “Gender, achievement, and persistence in an undergraduate computer science program”. *ACM SIGMIS Database: the DATABASE for Advances in Information Systems* **37**.4 (2006), pp. 42–57.

[Kit08] Kittur, A. et al. “Crowdsourcing user studies with Mechanical Turk”. *Proceedings of the SIGCHI conference on human factors in computing systems*. ACM. 2008, pp. 453–456.

[Kle15] Kleij, F. M. Van der et al. “Effects of feedback in a computer-based learning environment on students’ learning outcomes: A meta-analysis”. *Review of educational research* **85**.4 (2015), pp. 475–511.

[Ko04] Ko, A. J. et al. “Six learning barriers in end-user programming systems”. *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 2004, pp. 199–206.

[Koe12] Koedinger, K. R. et al. “The Knowledge-Learning-Instruction framework: Bridging the science-practice chasm to enhance robust student learning”. *Cognitive science* **36**.5 (2012), pp. 757–798.

[Koe13] Koedinger, K. & Stamper, J. “Using data-driven discovery of better student models to improve student learning”. *Proceedings of the International Conference on Artificial Intelligence in Education*. 2013.

[Laz14] Lazar, T. & Bratko, I. “Data-Driven Program Synthesis for Hint Generation in Programming Tutors”. *Proceedings of the International Conference on Intelligent Tutoring Systems*. Springer, 2014, pp. 306–311.

[Laz17] Lazar, T. et al. “Automatic Extraction of AST Patterns for Debugging Student Programs”. *Proceedings of the International Conference on Artificial Intelligence in Education*. 2017, pp. 162–174.

[Le09] Le, N.-T. et al. “Evaluation of a constraint-based homework assistance system for logic programming”. *Proceedings of the 17th International Conference on Computers in Education* (2009), pp. 51–58.

[Lee11] Lee, M. J. & Ko, A. J. “Personifying programming tool feedback improves novice programmers’ learning”. *Proceedings of the seventh international workshop on Computing education research*. ACM. 2011, pp. 109–116.

[Lee15] Lee, M. J. & Ko, A. J. “Comparing the effectiveness of online learning approaches on CS1 learning outcomes”. *Proceedings of the eleventh annual international conference on international computing education research*. ACM. 2015, pp. 237–246.

[Lep93] Lepper, M. R. et al. “Motivational techniques of expert human tutors: Lessons for the design of computer-based tutors”. *Computers as cognitive tools* **1993** (1993), pp. 75–105.

[Lew11] Lewis, C. M. et al. “Deciding to major in computer science: a grounded theory of students’ self-assessment of ability”. *Proceedings of the seventh international workshop on Computing education research*. 2011, pp. 3–10.

[Lis16] Lishinski, A. et al. “Learning to program: Gender differences and interactive effects of students’ motivation, goals, and self-efficacy on performance”. *Proceedings of the 2016 ACM Conference on International Computing Education Research*. 2016, pp. 211–220.

[Lok16] Loksa, D. & Ko, A. J. “The role of self-regulation in programming problem solving process and success”. *Proceedings of the 2016 ACM conference on international computing education research*. 2016, pp. 83–91.

[Luc07] Luckin, R et al. “Beyond the code-and-count analysis of tutoring dialogues”. *Artificial intelligence in education: Building technology rich learning contexts that work, R. Luckin, KR Koedinger, and J. Greer, Eds. IOS Press* (2007), pp. 349–356.

[LR18] Luxton-Reilly, A. et al. “Introductory programming: a systematic literature review”. *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. 2018, pp. 55–106.

[Mag17] Maguire, M. & Delahunt, B. “Doing a thematic analysis: A practical, step-by-step guide for learning and teaching scholars.” *AISHE-J: The All Ireland Journal of Teaching and Learning in Higher Education* **9**.3 (2017).

[Mal10] Maloney, J. et al. “The scratch programming language and environment”. *ACM Transactions on Computing Education (TOCE)* **10**.4 (2010), p. 16.

[Man06] Mannila, L. et al. “What about a simple language? Analyzing the difficulties in learning to program”. *Computer Science Education* **16**.3 (2006), pp. 211–227.

[Mar17a] Margulieux, L. & Catrambone, R. “Using Learners’ Self-Explanations of Subgoals to Guide Initial Problem Solving in App Inventor” (2017), pp. 21–29.

[Mar16] Margulieux, L. E. et al. "Employing subgoals in computer programming education". *Computer Science Education* **26**.1 (2016), pp. 44–67.

[Mar17b] Martin, C. et al. "Designing engaging learning experiences in programming". *International Conference on Computer Supported Education*. Springer. 2017, pp. 221–245.

[Mar19a] Marwan, S. et al. "The Impact of Adding Textual Explanations to Next-step Hints in a Novice Programming Environment". *Proceedings of the 24th Annual ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE19 (forthcoming)*. 2019.

[Mar21a] Marwan, S. et al. "Just a Few Expert Constraints Can Help: Humanizing Data-Driven Subgoal Detection for Novice Programming". *Proceedings of the International Conference on Educational Data Mining (EDM'21)* (2021).

[Mar21b] Marwan, S. et al. "Promoting Students' Progress-Monitoring Behavior during Block-Based Programming". *Proceedings of the 21st Koli Calling International Conference on Computing Education Research [To be Published]* (2021).

[Mar19b] Marwan, S. et al. "An Evaluation of the Impact of Automated Programming Hints on Performance and Learning". *Proceedings of the International Computing Education Research Conference*. 2019.

[Mar20a] Marwan, S. et al. "Adaptive Immediate Feedback Can Improve Novice Programming Engagement and Intention to Persist in Computer Science". *Proceedings of the 2020 ACM Conference on International Computing Education Research*. 2020, 194–203.

[Mar20b] Marwan, S. et al. "Immediate Data-Driven Positive Feedback Increases Engagement on Programming Homework for Novices" (2020).

[Mar20c] Marwan, S. et al. "Unproductive Help-seeking in Programming: What it is and How to Address it". *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. 2020, pp. 54–60.

[McN17] McNamara, D. S. "Self-Explanation and Reading Strategy Training (SERT) Improves Low-Knowledge Students' Science Course Performance". *Discourse Processes* (2017).

[Mer12] Merolla, D. M. et al. "Structural precursors to identity processes: The role of proximate social structures". *Social Psychology Quarterly* **75**.2 (2012), pp. 149–172.

[Mit99] Mitrovic, A. & Ohlsson, S. “Evaluation of a constraint-based tutor for a database language” (1999).

[Mit02] Mitrovic, A. et al. “Using Evaluation to Shape ITS Design: Results and Experiences with SQL-Tutor”. *User Modeling and User-Adapted Interaction* **12** (2002), pp. 243–279.

[Mit04] Mitrovic, A. et al. “DB-suite: Experiences with three intelligent, web-based database tutors”. *Journal of Interactive Learning Research* **15**.4 (2004), pp. 409–432.

[Mit07] Mitrovic, A. et al. “Intelligent tutors for all: Constraint-based modeling methodology, systems and authoring”. *IEEE Intelligent Systems* **22** (2007), pp. 38–45.

[Mit13] Mitrovic, A. et al. “The effect of positive feedback in a constraint-based intelligent tutoring system”. *Computers & Education* **60**.1 (2013), pp. 264–272.

[Miw13] Miwa, K. et al. “Stoic Behavior in Hint Seeking when Learning using an Intelligent Tutoring System”. *Proceedings of the Annual Meeting of the Cognitive Science Society*. Vol. 35. 35. 2013.

[Mor99] Moreno, R. & Mayer, R. E. “Cognitive principles of multimedia learning: The role of modality and contiguity.” *Journal of educational psychology* **91**.2 (1999), p. 358.

[Mor04] Moreno, R. & Mayer, R. E. “Personalized messages that promote science learning in virtual environments.” *Journal of educational Psychology* **96**.1 (2004), p. 165.

[Mor15] Morrison, B. B. et al. “Subgoals , Context , and Worked Examples in Learning Computing Problem Solving”. *Proceedings of the International Computing Education Research Conference*. 2015, pp. 21–29.

[Mur05] Murray, R. C. & VanLehn, K. “Effects of Dissuading Unnecessary Help Requests While Providing Proactive Help.” *AIED*. Citeseer, 2005, pp. 887–889.

[Mur06] Murray, R. C. & VanLehn, K. “A comparison of decision-theoretic, fixed-policy and random tutorial action selection”. *International Conference on Intelligent Tutoring Systems*. Springer, 2006, pp. 114–123.

[Nar04] Narciss, S. & Huth, K. “How to design informative tutoring feedback for multimedia learning”. *Instructional design for multimedia learning* **181195** (2004).

[Nat01] Nathan, M. J. et al. “Expert blind spot: When content knowledge eclipses pedagogical content knowledge”. *Proceedings of the third international conference on cognitive science*. Vol. 644648. 2001.

[NLG81] Nelson-Le Gall, S. “Help-seeking: An understudied problem-solving skill in children”. *Developmental Review* **1**.3 (1981), pp. 224–246.

[NLG86] Nelson-Le Gall, S. *Help-seeking behavior in learning*. ERIC Clearinghouse, 1986.

[New94] Newman, R. S. “Adaptive help seeking: A strategy of self-regulated learning.” (1994).

[Paa03] Paas, F. et al. “Cognitive load measurement as a means to advance cognitive load theory”. *Educational psychologist* **38**.1 (2003), pp. 63–71.

[Paa18] Paaßen, B. et al. “The Continuous Hint Factory -Providing Hints in Vast and Sparsely Populated Edit Distance Spaces”. *Journal of Educational Data Mining* (2018), pp. 1–50.

[Pal05] Palmer, D. “A Motivational View of Constructivistâinformed Teaching”. *International Journal of Science Education* **27**.15 (2005), pp. 1853–1881.

[Per14] Perelman, D. et al. “Test-Driven Synthesis for Automated Feedback for Introductory Computer Science Assignments”. *Proceedings of the Workshop on Data Mining for Educational Assessment and Feedback*. 2014.

[Pie15] Piech, C. et al. “Autonomously Generating Hints by Inferring Problem Solving Policies”. *Proceedings of the ACM Conference on Learning @ Scale*. 2015, pp. 1–10.

[Pri19] Price, T. W. et al. “A Comparison of Two Designs for Automated Programming Hints.” (2019).

[Pri15] Price, T. W. & Barnes, T. “An Exploration of Data-Driven Hint Generation in an Open-Ended Programming Problem”. *Proceedings of the Workshop on Graph-Based Data Mining*. 2015.

[Pri17a] Price, T. W. et al. “Evaluation of a Data-driven Feedback Algorithm for Open-ended Programming”. *Proceedings of the International Conference on Educational Data Mining*. 2017.

[Pri17b] Price, T. W. et al. “Factors Influencing Students’ Help-Seeking Behavior while Programming with Human and Computer Tutors”. *Proceedings of the International Computing Education Research Conference*. 2017.

[Pri17c] Price, T. W. et al. “Factors Influencing Students’ Help-Seeking Behavior while Programming with Human and Computer Tutors”. *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, 2017, pp. 127–135.

[Pri17d] Price, T. W. et al. “Hint Generation Under Uncertainty: The Effect of Hint Quality on Help-Seeking Behavior”. *Proceedings of the International Conference on Artificial Intelligence in Education*. 2017.

[Pri17e] Price, T. W. et al. “iSnap: Towards Intelligent Tutoring in Novice Programming Environments”. *Proceedings of the ACM Technical Symposium on Computer Science Education*. 2017.

[Pri21] Price, T. W. et al. “Exploring Design Choices in Data-driven Hints for Python Programming Homework”. *Proceedings of the Eighth ACM Conference on Learning@ Scale*. 2021, pp. 283–286.

[Pri18] Price, T. et al. “The impact of data quantity and source on the quality of data-driven hints for programming”. *Proceedings of the International Conference on Artificial Intelligence in Education*. 2018.

[Pri20] Price, T. et al. “An Evaluation of Data-driven Programming Hints in a Classroom Setting”. *Proceedings of the International Conference on Artificial Intelligence in Education (forthcoming)*. 2020.

[Puu98] Puustinen, M. “Help-seeking behavior in a problem-solving situation: Development of self-regulation”. *European Journal of Psychology of education* **13**.2 (1998), p. 271.

[Raz10] Razzaq, L. & Heffernan, N. T. “Hints: is it better to give or wait to be asked?” *International Conference on Intelligent Tutoring Systems*. Springer, 2010, pp. 349–358.

[Raz07] Razzaq, L. et al. “What level of tutor feedback is best”. *Proceedings of the 13th Conference on Artificial Intelligence in Education, IOS Press*. 2007.

[Ren02] Renkl, A. “Worked-out examples: Instructional explanations support learning by self-explanations”. *Learning and Instruction* **12**.5 (2002), pp. 529–556.

[Rio17] Riordan, M. A. “Emojis as tools for emotion work: Communicating affect in text messages”. *Journal of Language and Social Psychology* **36**.5 (2017), pp. 549–567.

[Riv17a] Rivers, K. “Automated Data-Driven Hint Generation for Learning Programming”. PhD. Carnegie Mellon University, 2017, pp. 1–140.

[Riv17b] Rivers, K. & Koedinger, K. R. “Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor”. *International Journal of Artificial Intelligence in Education* **27**.1 (2017), pp. 37–64.

[Riv17c] Rivers, K. & Koedinger, K. R. "Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor". *International Journal of Artificial Intelligence in Education* **27**.1 (2017), pp. 37–64.

[Rol14] Roll, I. et al. "On the benefits of seeking (and avoiding) help in online problem-solving environments". *Journal of the Learning Sciences* **23**.4 (2014), pp. 537–560.

[Rom20] Romero, C. & Ventura, S. "Educational data mining and learning analytics: An updated survey". *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* **10**.3 (2020), e1355.

[Roy05] Roy, M. & Chi, M. T. "The self-explanation principle in multimedia learning". *The Cambridge handbook of multimedia learning* (2005), pp. 271–286.

[Sch04] Scheeler, M. C. et al. "Providing performance feedback to teachers: A review". *Teacher education and special education* **27**.4 (2004), pp. 396–407.

[Sch84] Schunk, D. H. "Self-efficacy perspective on achievement behavior". *Educational psychologist* **19**.1 (1984), pp. 48–58.

[Sch06] Schworm, S. & Renkl, A. "Computer-supported Example-based Learning: When Instructional Explanations Reduce Self-explanations". *Computers & Education* **46**.4 (2006), pp. 426–445.

[Sha20] Shabrina, P. et al. "The Impact of Data-driven Positive Programming Feedback: When it Helps, What Happens when it Goes Wrong, and How Students Respond". *Educational Data Mining in Computer Science Education (CSEDM) Workshop @ EDM'20*. 2020.

[Shi08a] Shih, B. et al. "A Response Time Model for Bottom-Out Hints as Worked Examples". *Proceedings of the International Conference on Educational Data Mining*. 2008, pp. 117 –126.

[Shi08b] Shih, B. et al. "A Response Time Model For Bottom-Out Hints as Worked Examples." *Proceedings of the 1st International Conference on Educational Data Mining, EDM* (2008).

[Shi18] Shin, H. et al. "Understanding the Effect of In-Video Prompting on Learners and Instructors". *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: ACM, 2018, 319:1–319:12.

[Shu08] Shute, V. J. "Focus on formative feedback". *Review of educational research* **78**.1 (2008), pp. 153–189.

[Sim19] Simon et al. “Pass Rates in Introductory Programming and in other STEM Disciplines”. *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. 2019, pp. 53–71.

[Skr09] Skrivanek, S. “The use of dummy variables in regression analysis”. *More Steam, LLC* (2009).

[Sta08] Stamper, J & Barnes, T. “The hint factory: Automatic generation of contextualized help for existing computer aided instruction”. *Proceedings of the 9th International Conference on Intelligent Tutoring Systems Young Researchers Track*. 2008.

[Swe88] Sweller, J. “Cognitive load during problem solving: Effects on learning”. *Cognitive science* **12**.2 (1988), pp. 257–285.

[Thu13] Thurlings, M. et al. “Understanding feedback: A learning theory perspective”. *Educational Research Review* **9** (2013), pp. 1–15.

[Tol20] Toll, D. et al. “Current State and Next Steps on Automated Hints for Students Learning to Code”. *2020 IEEE Frontiers in Education Conference (FIE)*. IEEE. 2020, pp. 1–5.

[Ull03] Ullman, J. B. & Bentler, P. M. “Structural equation modeling”. *Handbook of psychology* (2003), pp. 607–634.

[Vae14] Vaessen, B. E. et al. “University students’ achievement goals and help-seeking strategies in an intelligent tutoring system”. *Computers & Education* **72** (2014), pp. 196–208.

[Van06] VanLehn, K. “The behavior of tutoring systems”. *International journal of artificial intelligence in education* **16**.3 (2006), pp. 227–265.

[Van11] VanLehn, K. “The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems”. *Educational Psychologist* **46**.4 (2011), pp. 197–221.

[Van05] Vanlehn, K. et al. “The Andes physics tutoring system: Lessons learned”. *International Journal of Artificial Intelligence in Education* **15**.3 (2005), pp. 147–204.

[Vie17] Vieira, C. et al. “Writing in-code comments to self-explain in computational science and engineering education”. *ACM Transactions on Computing Education (TOCE)* **17**.4 (2017), p. 17.

[Vih15] Vihavainen, A. et al. “Benefits of Self-explanation in Introductory Programming”. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education - SIGCSE ’15* **68** (2015), pp. 284–289.

[VH15a] Von Hippel, P. “Linear vs. logistic probability models: Which is better, and when”. *Statistical Horizons* (2015).

[VH15b] Von Hippel, P. “Linear vs. logistic probability models: Which is better, and when”. *Statistical Horizons* (2015).

[Wan17] Wang, K. et al. “Data-Driven Feedback Generator for Online Programing Courses”. *Proceedings of the ACM Conference on Learning @ Scale*. 2017, pp. 257–260.

[Wan20] Wang, W. et al. “Crescendo: Engaging Students to Self-Paced Programming Practices”. *To be published in the 51st ACM Technical Symposium on Computer Science Education (SIGCSE ’20)*. 2020.

[Wan21] Wang, W. et al. “SnapCheck: Automated Testing for Snap Programs”. ITiCSE’21, to appear. Association for Computing Machinery, 2021.

[Wat14] Watson, C. & Li, F. W. B. “Failure rates in introductory programming revisited”. *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education*. ACM. 2014, pp. 39–44.

[Wat11] Watson, C. et al. “Learning programming languages through corrective feedback and concept visualisation”. *International Conference on Web-Based Learning*. Springer. 2011, pp. 11–20.

[Wat12] Watson, C. et al. “BlueFix: Using crowd-sourced feedback to support programming students in error diagnosis and repair”. *Proceedings of the International Conference on Web-based Learning*. 2012, pp. 228–239.

[Wei15] Weintrop, D. & Wilensky, U. “Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs.” *ICER*. Vol. 15. 2015, pp. 101–110.

[Wil05] Wiliam, D. “Keeping learning on track: Formative assessment and the regulation of learning”. *Review process* **20** (2005).

[Wil16] Williams, J. J. et al. “Revising Learner Misconceptions Without Feedback: Prompting for Reflection on Anomalies”. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. CHI ’16. Santa Clara, California, USA: ACM, 2016, pp. 470–474.

[Win96] Winslow, L. E. “Programming pedagogy—a psychological overview”. *ACM Sigcse Bulletin* **28**.3 (1996), pp. 17–22.

[Woo99] Wood, H & Wood, D. “Help seeking, learning and contingent tutoring”. *Computers & Education* **33**.2-3 (1999), pp. 153–169.

[Yi17] Yi, J. et al. “A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments”. *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 2017, pp. 740–751.

[Zhi18] Zhi, R. et al. “Reducing the State Space of Programming Problems through Data-Driven Feature Detection”. *Proceedings of the Educational Data Mining in Computer Science Education Workshop at the International Conference on Educational Data Mining*. 2018.