iSnap: Evolution and Evaluation of a Data-Driven Hint System for Block-based Programming

Samiha Marwan, and Thomas W. Price

***Abstract*—Novice programmers often struggle on assignments, and timely help, such as a hint on what to do next, can help students continue to progress and learn, rather than giving up. However, in large programming classrooms, it is hard for instructors to provide such real-time support for every student. Researchers have therefore put tremendous effort into developing algorithms to generate automated, data-driven hints to help students at scale. Despite this, few controlled studies have directly evaluated the impact of such hints on students' performance, and learning. It is also unclear what specific design features make hints more or less effective. In this work, we present iSnap, a block-based programming environment that provides novices with data-driven, next-step hints in real-time. This paper describes our improvements to iSnap over 4 years, including its "enhanced" next-step hints with three design features: textual explanations, self-explanation prompts and an adaptive hint display. Moreover, we conducted a controlled study in an authentic classroom setting over several weeks to evaluate the impact of iSnap's enhanced hints on students' performance and learning. We found students who received the enhanced hints perform better on in-class assignments and have higher programming efficiency in homework assignments than those who did not receive hints, but that hints did not significantly impact students' learning. We also discuss the challenges of classroom studies and the implications of enhanced hints compared to prior evaluations in laboratory settings, which is essential to validate the efficacy of next-step hints' impact in a real classroom experience.**

***Index Terms*—next-step hints, data-driven hints, block-based programming**

I. Introduction

Programming is a vital skill in many disciplines, as evidenced by the large number of introductory programming courses offered in K-12 and university settings (including for non-CS majors [1]). However, recent surveys show that learning how to program is challenging for students in early programming stages; as concluded in a 2019-survey of Computer Science (CS) failure rates: "It appears that introducing students to computing is still one of computing education's grand challenges [2]." A fundamental challenge for many students is simply getting through the programming assignment – if students get stuck and help is not available, they may grow frustrated or negatively self-assess their programming ability which can lead them to give up or leave the field entirely [3]. When students are unsure how to proceed, a promising way to help them is by providing a next-step hint, which typically suggests an edit that the student should make to their program to bring it closer to a correct solution, allowing them to proceed. *Automated* hints, provided by the learning environment, are particularly useful in large classrooms in situations when the instructor is not available, or when other factors (e.g. fear of looking unintelligent [4]) prevent students from asking for help from course staff [5].

One way to generate these hints is using *data-driven* methods. Unlike traditional "expert models," which require an expert to author each hint and define rules for when it applies (e.g. [6], [7], [8]), data-driven programming hints can be generated automatically using a dataset of solutions [9], [10], [11], [12], e.g. from students in a prior semester. This allows them to scale easily to new problems and contexts. Additionally, some prior evaluations of data-driven hints show that they can approach the quality of expert-authored hints [9], [13], [14]. However, data-driven hints suffer a number of challenges that limit their impact on students' performance and learning in classrooms. First, next-step hints are "bottom-out" hints [15], telling students *what* to do, not *why*, and therefore, students may not fully understand the hint [16]. 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 [17]. Third, next-step hints are usually on-demand hints, and students may avoid hints when they need help, or abuse them to solve the problem without trying – unproductive help-seeking behaviors can harm students' learning [18]. In this paper, we first present the iSnap system, which we have been improving and evaluating for the past 4 years. iSnap is an extension to the Snap*!* block-based programming environment, and the first system to provide real-time data-driven next-step hints to students in a block-based environment [15]. We highlight how the iSnap system has evolved to include *enhanced* hints that incorporate three specific design features: textual explanations, self-explanation prompts, and adaptive hint display, that address next-step hints' challenges mentioned above.

Additionally, there has been little work evaluating the impact of automated, next-step programming hints (whether data-driven or not) on students' outcomes [19], [15], [16]. The evaluations that have been done have a few key limitations that leave open research questions about how to apply data-driven hints in the classroom. First, most of these evaluations have been limited to technical assessments [20], using few (i.e. 1-2) programming assignments [21], [22], [11]. Second, most of these studies have been in limited, laboratory settings [23], [22], [24], which limits our understanding of the impact of hints on students in authentic classrooms. Third, the few studies that have used good measures in authentic classroom settings have had conflicting findings on the impact of data-driven hints on students' outcomes [19], [11], with some suggesting hints can lead to improvement in students' performance and learning [12], [11], and others showing little to no impact [19], [25]. The inconsistency of these results is in large part due to the variety of ways that hints can be displayed which are not always articulated or evaluated in a

way that helps us understand why hint evaluations lead to differing results.

To address these limitations of prior evaluations, this paper also presents an evaluation of the iSnap system through a controlled study, taking place in an authentic classroom setting, over four weeks across multiple assignments, leveraging several assessment mechanisms (e.g. surveys, and post-tests) to show how iSnap's enhanced hints impacted students' performance and learning. This study serves as a conceptual replication of prior evaluations of the iSnap system in a laboratory setting [11] and evaluates how robust these results are when generalizing to a different learning context and population. Replication is a critical contribution of this work, since prior work on data-driven hints lacks such replications, and has reached conflicting findings on their efficacy. For example, will enhanced next step hints be as effective for students in an authentic classroom, with different, complex assignments over multiple weeks, as they are in short-term laboratory studies? We also explore how factors such as problem difficulty or hint quality can mediate the impact of next step hints on students' performance [26], [27], which are not well-explored in prior work.

In this paper, we investigate the following research questions: *In a classroom setting, what is the impact of enhanced next-step hints on students' **RQ1:**) programming performance and **RQ2:**) learning; **RQ3:**) how do students perceive these hints? and **RQ4:** how does problem difficulty and hint quality mediate the effect of enhanced next-step hints on students' outcomes?* Our classroom study results show that enhanced hints have an overall positive, significant impact on students' performance on in-class programming assignments. In addition, we found that such enhanced hints significantly improved students' programming efficiency on homework tasks when hints were available. However, we did not find conclusive evidence that hints improved students' learning. Overall our results suggest that automated next-step hints, when well-designed, can be very useful to students, playing a necessary role in an authentic classroom setting with limited human support, and do not harm learning (as in prior work [18]), despite giving away part of the solution.

In summary, the primary contributions of this work are:

1) The iSnap system that provides enhanced, data-driven hints, including explicit different design features to improve the effectiveness of these hints.
2) An empirical evaluation of iSnap's enhanced hints on students' outcomes in an authentic classroom setting, and how the impact of hints can be mediated with contextual factors, i.e. problem difficulty and hint quality.

II. Related Work

Research in computing education research has developed several educational tools that generate automated hints to support students in programming. In this section we review how hints can be effective from a theoretical perspective, prior work on hint generation techniques (especially data-driven approaches), and empirical evaluations on the impact of these hints.

Theoretical Perspectives on How Hints can Improve Students' Performance and Learning: As Aleven and Koedinger argue [18], from a theoretical perspective, the most fundamental reason to provide hints to students is to help the student complete a problem when they are struggling and might otherwise give up, increasing the amount of learning content the student engages with. However, *cognitive* theories shed light on other ways that hints may promote learning. First, the ACT-R (Adaptive Control of Thought - Rational) cognitive theory posits that learning problem solving requires two types of interrelated knowledge: declarative knowledge, which is a set of known facts or goals (such knowing *what* a variable is), and procedural knowledge, which is a set of production rules that specify how to reach a goal (such as knowing *how* to use a variable) [28], [29]. Under this theory, principle-based hints *that explain a domain concept* can support students to acquire declarative knowledge, though not all automated hints contain such explanations. Hints can also contextualize procedural knowledge by connecting a problem-solving step to a relevant domain concept [30]. Second, the Knowledge-Learning-Instruction (KLI) framework [17] suggests that hints can prompt the acquisition of declarative knowledge by engaging the student in *sense-making* of hints, such as reasoning or self-explaining the hint. Aleven and Koedinger argue that this self-explanation process is essential to achieve *learning* from hints, but that it is unlikely to occur spontaneously [18], and should therefore be actively supported by the hint system. Third, 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 [31]. Wood's theory of contingent tutoring emphasized that the tutor's help should be contingent upon the learner's needs to improve their learning [32]. The basic idea of contingent tutoring is that "when a learner has been set or is trying to achieve a goal and seems to be 'in trouble', then the contingent (human) tutor immediately offers help (page 2, [32])". This theory reveals that it is not only the content of the hints that can increase their effectiveness, but also how/when we are providing students with these hints can affect their impact on students' performance and learning. These theoretical perspectives suggest how hints may promote students' performance and learning, and also reveal the importance of how hints could be designed and presented to achieve that goal, which is our focus in this work. These theoretical models emphasize that learning happens from next-step hints primarily when students are presented with hints that include domain knowledge, and foster reflection on this knowledge, at the time when a student is struggling. We discuss how our design choices for iSnap's enhanced hints align with these goals in Sections III-B, III-C, and III-D.

Hint Generation: Automated hints are primarily generated either using expert-authored rules – such as in constraint-based [6] or model tracing approaches [8], [33] – or using data-driven methods that leverage historical data to generate hints [9], [12], [15]. We focus in this paper on data-driven hints, which are particularly promising because they require little expert effort to support new problems, and can scale to multiple solution strategies, which is an essential feature for supporting

students in open-ended programming assignments [19], [10], [34]. Personalized, data-driven hints have been successfully generated to support many programming languages, such as python [19], Java [35], [36], Prolog [37], C++ pseudocode [12], and block-based programming languages [15], [10]. These hint-generation algorithms use a variety of approaches, but many align with the original Hint Factory approach [38], originally used in the domain of logic. This approach uses prior student data to model how successful students solve the problem, and then uses this model to guide struggling students to take actions that successful students would take in a similar situation. In Section III-A we provide a more in-depth explanation of one hint generation approach. However, the focus of this paper is on the way a hint system *presents* these hints to students – not the algorithm itself – and the particular design challenges that come along with data-driven hints (detailed in Section III), such as interpretability [39], [16], fostering learning, and help abuse or avoidance [18], [22].

Empirical evaluations: Most evaluations of next-step hints have focused on technical aspects of the hint generation process, such as how often the system can generate a correct hint [40], or expert ratings of the quality of the generated hint (e.g. whether it can resolve a mistake or a missing step in students' code) [13]. For example, Hartmann et al. 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" [14]. They found that 47% of hint requests returned helpful suggestions. Another example is the ITAP tutoring system developed by Rivers et al., which offers data-driven hints for Python programming. Rivers et al. found that the ITAP tutor is able to construct a set of hints leading to a solution for 98% of incorrect solution attempts [9].

While good hint quality is important for hints to be useful [39], it is also no guarantee of hints' impact on students' performance and learning. For example, ITAP's data-driven hints were found to approach the quality of human-authored hints [9], [19], [41]. However, Rivers found no difference in learning between students with and without ITAPs hints, and only suggestive evidence that hints helped students complete practice problems faster [19]. Choudhury et al. found that their data-driven style feedback helped students who had *already completed* programming problems to significantly improve the quality of their solutions, but they did not evaluate learning on a post-test without hints [42].

In addition, prior studies on the impact of data-driven next-step hints on students' outcomes have shown different, and sometimes conflicting, findings, even for the same programming language [12], [19], [25]. This might be due to differences in how hints are presented by the system, or the information the hints provide, such as combining hints with other forms of feedback [12], [6]. For example, Fossati et al. found that the iList tutor (a tutor teaching linked lists, a skill related to programming) that provided next-step hints in addition to other forms of automated support (such as positive feedback) improved students' performance [12]. However, Price et al. found that data-driven next-step hints, in the form of code highlights (i.e. highlighting code to be inserted and crossing out code to be deleted), in a Python programming environment had no effect [25]. This reveals the importance of understanding which design features are necessary to improve the effectiveness of automated hints, and in which situations they were or were not helpful, as well as the importance of *replicating* these evaluations in different contexts. 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" [43]. In Section III we present results from our earlier laboratory evaluations of iSnap, and our goal in this work is to investigate how well these findings generalize to a different learning context and population.

Other studies have shown that *automated* (but not fully data-driven) programming hints can improve students' performance and learning. In the Lisp Tutor, students with a variety of automated feedback learned significantly more than those without feedback, measured by a post-test with no hints [7]. Gusukuma et al. [33] found that students with their automated misconception-driven feedback performed significantly better on an immediate post-test than those without, but not a delayed post-test. These studies offer some evidence that automated programming feedback can improve learning. However, none of these systems was *fully data-driven*, and the feedback included instructor-authored messages and rules that may be difficult to scale to new problems. It is therefore still an open question whether fully data-driven hints can positively impact learning. Additionally, more work is needed to investigate how laboratory studies [7], [11], [42] will generalize to authentic classroom settings, as we do in this work.

III. iSnap System Design

In this section, we present the evolution of *iSnap* – an extension to the Snap*!* block-based programming environment. iSnap can generate data-driven next-step hints automatically (as we describe in Section III-A) that can bring student code closer to the correct solution, with the goal of helping students progress and get unstuck during programming. This is important because improving students' progress will result in improving their programming performance, especially when instructor help is unavailable or it is difficult for students to access due to social or logistical barriers.

In addition to this goal, we strived to improve the impact of next-step hints with three other new design goals. Our **first design goal** is to help students *understand why the hint was given*, not just what to do next. This goal stems from previous evaluations of next-step hints [19], [25], [5], where many students noted that hints could be difficult to interpret. For example, one student in a study on iSnap noted that hints were, "unhelpful, mostly because I wasn't quite sure why it was offering the help that it was offering. It was just like, here's a suggestion- but why?" [5]. Our **second design goal** is to encourage students to *self-explain the hint*, and reflect on why it is needed. Traditionally, next-step hints are "bottom-out" hints, telling the student exactly what to do, without requiring them to reason about the information. Aleven et al.

argue that while such bottom out hints are necessary to help students who are stuck; however, they do not engage students in sense-making (i.e. reflecting on the hint to understand its meaning), which is an essential component of learning [17]. The **third design goal** is to help students *seek this help productively*. In prior work, most tutoring systems provide hints *on-demand* (i.e. upon a student's request). However, literature on help-seeking shows that novice students may avoid requesting hints when they need them, or may keep requesting and following them blindly to reach the correct solution, indicating an unproductive help-seeking behavior, which correlates negatively with learning [18]. Over the past 4 years we have been improving the design of iSnap hints to implement and evaluate these design goals; and investigate how they can be implemented in a way that makes iSnap hints overall more effective and increase their impact on students' performance and learning.

The interface of iSnap is displayed in Figure 1. When students use iSnap, it displays a menu of a set of programming assignments, where they can choose which assignment they want to solve. While students are programming, the iSnap system pops up a hint button attached to student code (as shown in Figure 1 A) when it detects that the student is struggling, using an **adaptive hint display** feature described in Section III-D. If a student clicks on the hint button a hint dialog appears, which includes 3 primary elements: 1) The **next-step hint**, which communicates a suggested code edit through a "diff", contrasting the student's current code (left) with suggested code (right), and highlighting the block to be added (or reordered). For example, in Figure 1 B, the suggested hint is to add the "repeat" block. 2) A **textual explanation**, as shown in Figure 1 C, which explains in plain language the suggested next-step hint, within the context of the specific assignment the student is working on. 3) a **self-explanation (SE) prompt**, as shown in Figure 1 D, which prompts the student to self-explain the suggested hint. The adaptive hint display, the textual explanation, and the SE prompts are the 3 new design features we added to the next-step hints to address our 3 design goals mentioned above, respectively. Figure 1 E shows the "swap roles" button which is a separate addition feature to iSnap discussed in Section IV-D3. Below, we describe the mechanism of generating next-step hints (Section III-A), followed by the mechanism of each design feature with our prior evaluations (Sections III-B, III-C, and III-D).

A. Generating Next-step Programming Hints

iSnap's next-step hints are generated by the data-driven `SourceCheck` algorithm. The algorithm is described in detail in [10], [13], and we provide a simplified explanation here. SourceCheck uses a database of correct solutions (represented in the form of Abstract Syntax Trees (ASTs)) for a given problem to generate hints automatically. To provide students with next-step hints in real time, after every code edit, the `SourceCheck` algorithm uses a code-specific distance metric to select the solution in its database that the student has made the most progress toward. It then calculates a set of edits (i.e. hints) to bring the student's code closer to the correct solution. The algorithm then prioritizes the hints (e.g. based on their order in the student's code). The current version of iSnap selects the earliest unseen hint to display to the student. The `SourceCheck` algorithm generates personalized hints that match the current student's solution strategy, if one exists in the solution database. The database of correct solutions used to generate hints can come from any source. The first version of the iSnap system generated hints from historically correct *student* solutions collected from prior semesters [15], since this data is readily available. However, later work showed that higher quality hints can be generated using solution templates, which define a variety of possible solution approaches, authored by instructors [13]. While different versions of iSnap have used different data resources, the evaluation presented in this paper uses the instructor-based solution templates.

Figure 1 shows an example of a next-step hint provided to a student while programming *Polygon Maker* exercise (described in Section IV-C). This hint suggests the next expected code block to be added in student code, which is the "repeat" block (i.e. a loop) – necessary to draw sides of the polygon based on a given number of sides. This example clarifies how the suggested data-driven next-step hint can help student progress, not only by suggesting which block to use, but also by visualizing *where* to place this code block.

B. Textual Explanation Design

The first support feature added to next step hints is a *textual explanation*, which is a brief expert-authored text explanation accompanying a next-step hint. This feature addresses our first design goal, which is to make hints more interpretable to students. Unlike fully data-driven approaches, these textual explanations require manual expert effort. Prior work shows that the presence of instructional explanations can help students understand why and when certain procedures are appropriate in example problems [44], [8], which can help students acquire declarative and procedural knowledge needed to improve their learning as stated in the ACT-R cognitive theory [28]. We therefore expect that the effort needed to author textual explanations to data-driven next-step hints is worthwhile and would similarly help students to understand hints, and improve students' outcomes during problem solving.

As suggested by prior work [6], [10], we designed the textual explanations to include three pieces of information: (1) What the suggested code block (i.e. the hint) does, (2) how it is related to the programming assignment, and (3) where the suggested block can be found. As shown in Figure 1 C, a next step hint that suggests the addition of a "repeat block" in the *Polygon Maker* exercise has the following textual explanation: *"The repeat block (under Control) allows you to run the same code a fixed number times, like drawing each side of a polygon"*. For each programming task, we manually authored textual explanations for *each* block in instructor-based solution templates. We then labeled each block in the correct solution AST with one or more relevant textual explanations. When the iSnap system suggests any next-step hint, it displays the corresponding textual explanation as well. If there is more than one textual explanation for a given block, the system displays

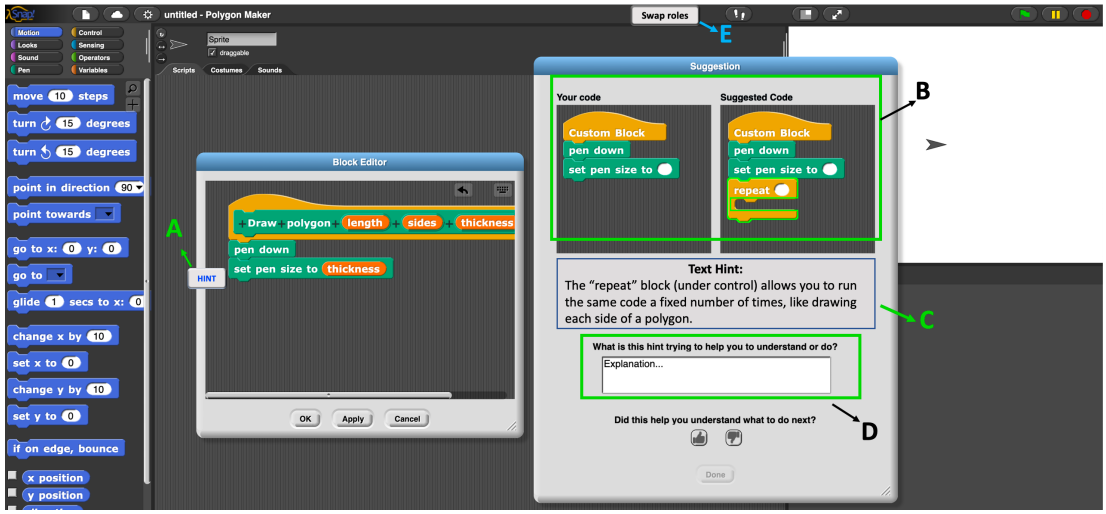

Fig. (1) The iSnap block-based programming environment. When the hint button (A) is clicked, a suggested next-step hint (B) is displayed, augmented with a textual explanation (C) and a self-explanation prompt (D). A "swap roles" button (E) is added on top to allow pair programming.

the first explanation that the student has not yet seen. We sometimes created multiple textual explanations for a given block in the solution to provide different forms of information to help students better understand the next-step hint, without giving them redundant information. A similar approach has been used in the MistakeBrowser system to annotate hints generated with student data as well [45].

In our prior evaluations of iSnap, we compared the impact of adding textual explanations to next-step hints [16], [11]. We found in a laboratory study that participants who were provided with hints augmented with textual explanations (experimental group) perceive iSnap's support as significantly more useful, relevant and interpretable and had a better understanding of the hints provided than participants who received only next-step hints (control group) [16]. However, we did not find a difference in performance between the experimental and control groups [16]. In another laboratory study we found that participants who had access to next-step hints with textual explanations, performed significantly better (i.e. achieved higher scores) than those who did not receive hints at all in programming tasks with hints available, but we did not find a difference in their learning [11].

C. Self-Explanation Prompt Design

The second design feature added to next-step hints is a *self-explanation* (SE) prompt, which is displayed after a student receives a hint, asking the student to reflect on why iSnap gave them that hint. Figure 1 D shows an example of a SE prompt. This feature addresses our second design goal, which is to allow students to *"make sense"* of the hint, which is an active learning approach that can improve students' learning from next-step hints as stated in the KLI framework [17]. This design feature is also adapted from previous work, which shows that prompting students to self-explain after viewing worked examples can improve their learning [46], [44]. As suggested by Aleven et al. [18], we apply this principle to our "bottom-out" next-step hints, which when combined with textual explanations, may act as micro-scale worked examples that can positively impact students' performance and learning. We designed the prompts to randomly show one of a variety of messages, such as "Why do you think iSnap recommended this hint?" and "What is this hint trying to help you to understand or do?" that encourages the student to think critically about the hint itself and its relation to their code. These self-explanation prompts are open-ended, and students can write anything in the response field, but they are forced to write at least 15 characters. While this may sound frustrating, it is an essential design feature to make students stop and self-explain, or otherwise they will simply ignore it [22]. In prior work, we found that most students normally answer the prompts [16].

In domains other than programming, several studies show that self-explanations can improve students' learning [47], [48]. In programming, Vihavainen et al. found that students who received self-explanations with supporting multiple-choice questions performed better on a programming exam [49]. In our prior work, in a laboratory study, we found that adding self-explanation to next-step hints improved participants' performance (i.e. scores) in programming tasks with hints, and improved their learning in isomorphic tasks without hints [11].

D. Adaptive Hint Display

Our third design choice focused on how the hints are presented and made available to students, what we call the *adaptive hint display*. Traditionally, hint systems have provided on-demand hints, where the student is required to recognize their need for help and request it. Other systems have provided *proactive* hints, which may interrupt students to suggest help when the system deems it necessary. iSnap balances these two approaches, prompting students, but in an unobtrusive way, and only when needed. This feature addresses our third design goal, which is to provide students with hints only

when they need it, as drawn by the contingent tutoring theory [32]. Additionally, this design choice serves as a guidance to encourage students to *productively* seek help, and to limit hint abuse (i.e. when students are not in need of help) and hints avoidance [50]. This is an important design choice because prior work shows that unproductive help-seeking behavior correlates with a decrease in student learning [18].

Our adaptive hint display is implemented using an adapted version of the data-driven `SourceCheck` algorithm, which works as follows. We developed a feature in iSnap that keeps track of students' progress by comparing the current student's code with prior correct solutions. If the distance between both ASTs is decreasing (i.e. the student solution is getting closer to the correct solution), the system will identify the student as being in a *"progressing"* state; otherwise, it identifies them as being in a *"struggling"* state. Every two minutes, iSnap checks the student state, and if the student is detected to be struggling, it pops up a hint button (as shown in Figure 1 A), which blinks for three seconds to draw their attention. The student can choose to click on it if they need help, or ignore it if they do not wish to be interrupted. If the student does not click on the hint, iSnap continues to accumulate hints every two minutes as long as the student continues to struggle, and the student can open these hints at any time they wish. These hints are *proactively* displayed based on the student's struggling state, and they are opened *on-demand* to give students the control on whether to see the hint or not; combining both benefits of proactive and on-demand hint displays.

In our prior work, we implemented a similar version of adaptive hint display, but it estimates student progress differently [22]. We found that students using adaptive hint display were less than half as likely to engage in unproductive help-seeking strategies (such as help abuse) than students using traditional on-demand hint display. We also found some inconclusive evidence suggesting that providing students with hints, only when they need it, may improve their learning in a future task without hints [22], though uncertainty in these results suggests the need for further evaluation.

IV. Methods

The prior work presented in the previous section shows that each of our design features for iSnap have improved it, and they have hopefully addressed some of the limitations in prior hint systems (including earlier versions of iSnap), which prevented them from showing impact on students' outcomes [9], [25], [22]. However, our prior evaluations have been limited to single-day, laboratory studies [11]. To understand how robust our prior findings on data-driven enhanced next-step hints are, and whether they generalize to a different population and learning context, below we present our second contribution: an empirical evaluation of the iSnap system over 4 weeks of a university-level introductory computing course for non-majors. This evaluation leverages multiple assessment mechanisms to evaluate the system's impact on students' outcomes. In this study, we seek to answer the following research questions: **RQ:** In a classroom setting, what is the impact of enhanced next-step hints on students' (**RQ1:**) programming performance and (**RQ2:**) learning?; **RQ3:**) how do students perceive these hints?; and (**RQ4:**) how does problem difficulty and hint quality mediate the effect of enhanced next-step hints on students' outcomes? This study is a *conceptual replication* [51], in which the procedures are different (in this case different population, duration, context, tasks, etc.), but includes similar research questions (i.e., what is the impact of hints on students' performance and learning).

A. Population

We conducted this study in a CS0 classroom in a public US university, 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.

B. Programming Environment

During the first 4 weeks, students learn block-based programming using the iSnap programming environment, in which they do their classroom and homework exercises. Due to COVID-19, classrooms were held online via Zoom. We discuss implications of the online setting in Section VII. For both classroom and homework exercises, iSnap provided some students with the enhanced next-step hints (depending on their condition, discussed below). As a part of the class requirements, *all students* (regardless of condition) 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 for 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 IV-D3 we fully discuss the pair programming feature, challenges, and how we dealt with it during data collection and data analysis.

C. Procedure

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

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

TABLE (I) Study procedure.

Week no.	Hints group (n = 30)	Control group (n = 32)
1	Consent form + Pre-survey (n = 62)	
	Polygon Maker (in class)	Polygon Maker (in class)
2	Squiral (in-class)	Squiral (in-class)
	Guessing Game (in class)	Guessing Game (in class)
	Daisy (HW1)	Daisy (HW1)
3	Frogger (in-class)	Frogger (in-class)
	BrickWall (HW2 - no hints)	
4	Post-survey and post-test	

provided no hints. As shown in Table I, 32 students were assigned to the `Control` condition, and 30 students were assigned to the `Hints` condition. Afterwards, to accommodate pair programming, students were randomly paired according to whether they had consented or not, and their condition, such that: for students who consented, we 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, we chose to give them access to the `Hint` condition, and we discuss the potential implications of this in Section VII.

Figure 2 shows a solution example for each programming assignment with its corresponding expected output. In **week 1** students had their first *in-class* programming assignment called *Polygon Maker*, which asks students to create a procedure with 3 parameters: *'n'*, *'len'*, *'thick'*, to draw a polygon with *'n'* sides, each with length *'len'*, and thickness *'thick'*. In **week 2**, students had three programming assignments: *Squiral* (in-class), *Guessing Game* (in-class), and *Daisy* (homework). *Squiral* asks students to create a procedure that takes user input for *'r'* and draws a spiral-shape 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-shape with a user-specified number *'n'* of overlapping circular petals with alternate colors.

In **week 3**, students had two programming assignments, *Frogger* (in-class), 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, we 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 [52] and described in Section IV-D2, followed by a post-survey to collect their perceptions about the usefulness of the hint features, and how they could be improved.

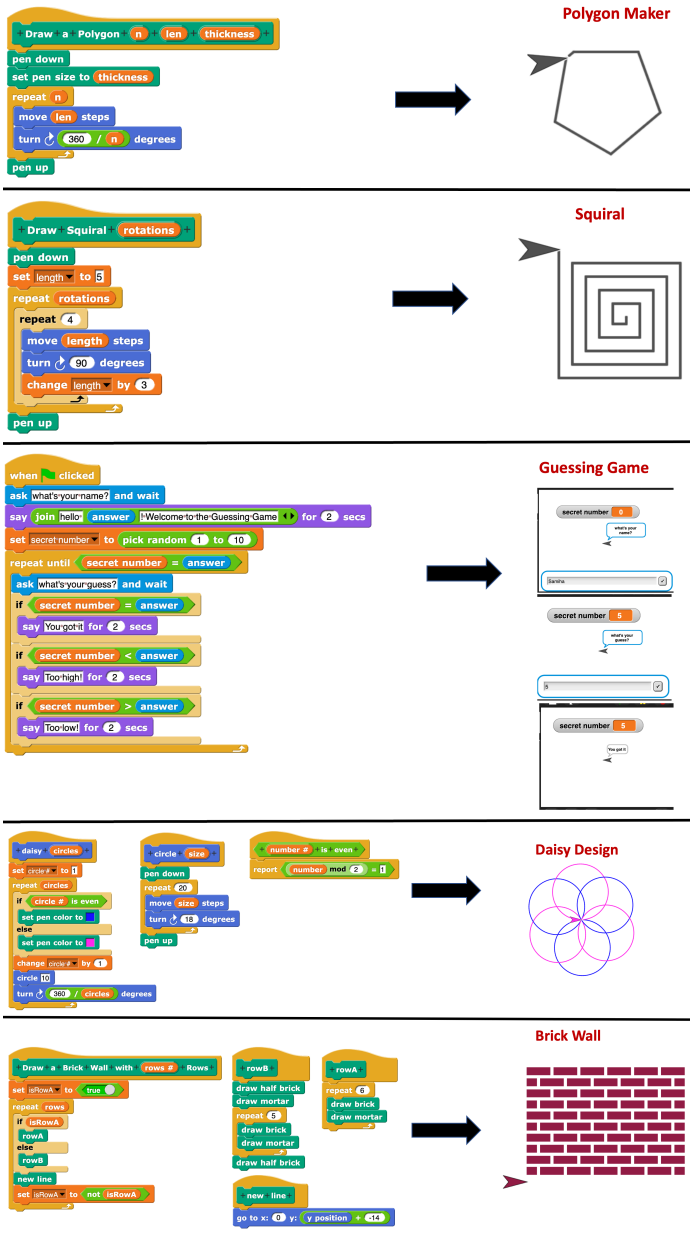

Fig. (2) One sample solution for each programming task (on the left), and their corresponding output (on the right).

D. Analysis & Measures

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

1) Performance: We used two measures of performance: students' scores on programming assignments, and time spent on these assignments.

Assignments' scores: To calculate students' scores on programming assignments, we 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 points due to late submissions leading to inaccurate scores, we 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 improving students' scores in the in-class assignments, we 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" [53]. 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 on the in-class assignments, the mean of students' scores was 88.4% (min = 0% ; SD = 20.83%; max = 100%). For further analysis of students' scores in any *individual* programming assignment, we 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. To provide more insights of the results, we report statistical tests along with effect sizes. We computed effect size estimates for Mann–Whitney U nonparametric test using the 'r' statistic described in [54] because our data was non-normal, where a large effect is $>= 0.5$, a medium effect is 0.3, and a small effect is 0.1. However, these results should be considered with caution since the sample size for each condition is ~ 30.

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

Analytical Approach: Similar to the analysis on students' scores, we 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 assignments. Across both conditions in the in-class assignments, the average student active time (in minutes) was 20 (min = 2; SD = 14.52; max = 81). In addition, for any individual task, we used a Mann-Whitney U test to measure the statistical significance of differences in the active time across conditions.

2) Learning: We measured students' learning in two ways. First, we compared students' performance in the *BrickWall* homework assignment, where both conditions have no access to hints. The *BrickWall* task 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 one. 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. The maximum score in the post-test is 14/14, i.e. 100%. The post-test questions consisted of Snap code tracing and fill-in-the-blank code completion questions[2], which directly aligned with the learning objectives of the in-class and homework assignments focusing on variables loops, conditionals and procedures.

Analytical Approach: For each of our learning measures (i.e. the assessment task, and the post-test), we 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).

3) Pair Programming Analysis: 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 roles" button (shown in Figure 1 E) 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, that logs all student actions (i.e. code edits) during programming [15], is modified to also log time instances when students switch roles.

When analyzing students' log data while they were pair programming, we found three primary pair behaviors: First, some students worked separately on two different iSnap 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 assignment in-class, while others had to finish it (separately) for homework. Because of these discrepancies, when analyzing students' log data, we treated each student as a *separate* instance, where each student had a final submitted attempt, which we used to calculate students' performance, and this attempt has a single code trace (logged by iSnap), which we used to measure time on task. Students who worked in pairs all the time, their 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. For those latter students, we calculated their active time in a given programming assignment by combining the time they worked together, in addition to the time they worked separately.

4) Students' Perceptions: Using collected post surveys, we performed quantitative analysis to evaluate students' ratings

[2]You can view post-test questions here: www.go.ncsu.edu/posttestfall20

in Likert-scale questions about the usefulness of iSnap's enhanced next-step hints, and which design feature they prefer. For open-ended questions, we reviewed all students' responses to gain qualitative insights about students' perspectives on hints during an authentic classroom experience. We identified positive and negative themes, as in thematic analysis [55], that indicated reasons why enhanced next step hints, or why specific design features (e.g. SE prompts), were helpful or unhelpful, respectively.

5) 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, we measured *hint usage*, as a measure of hint quality, in three ways: (1) how many hints students have *opened* in each programming assignment; (2) of these hints, how many were followed, i.e. whether students applied the hint's suggestion within a maximum of 10 code edits[3]; and (3) how students answered SE prompts. To grade students' answers to open-ended SE prompts, we used a simplified rubric compared to prior work [16], [56]. In particular, we 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 (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").

V. Results

We present the results in the order of the research questions. We analyzed in-class assignments together, since we are interested in the overall effect of hints on performance, time and learning, as discussed in Section IV-D on the in-class assignments: *PolygonMaker*, *Squiral*, *Guessing Game*. We chose to analyze the *Daisy* homework assignment 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 II shows descriptive statistics of students' performance and time in all the programming assignments.

Statistical Comparisons: We investigate our RQs through two statistical models and five statistical tests below, using an alpha value of 0.05. As suggested by [57], [58], [59], we have not adjusted the alpha value to control the experiment-wise error rate, but instead justify for each test "what was done and why, and we discuss the possible interpretations of each result," allowing the reader to weigh the evidence in light of this information. We scope our claims accordingly.

A. RQ1- Performance

Recall that we measured students' performance by calculating students' scores on the programming assignment and their active time spent on these assignments.

[3]We 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.

TABLE (II) The mean/medians of students' performance scores (in %) and active time (in minutes) for each programming task across the `Hints` and `Control` group. The bold numbers indicate which group has a higher value (not the statistical significance).

	Students' scores		**Active time**	
	Hints	Control	Hints	Control
Polygon Maker	**92 / 100**	76.85 / 100	10.1 / 10.17	**11.1 / 10**
Squiral	**92.2 / 100**	84.26 / 100	**27.8 / 24.9**	21.9 / 16
Guessing Game	91.9 / 100	**93.6 / 100**	23.67 / 19	**24.2 / 23**
Daisy	**90.4 / 100**	86.5 / 100	42.65 / 28.5	**62.1 / 44**
BrickWall	**97.8 / 100**	94 / 100	57.87 / 57	**61 / 54**

Assignment scores: To evaluate the impact of hints on students' scores for the 3 in-class assignments, we used a linear mixed-effects model (Model A), with *score* as the dependent variable, *condition* and *task* as independent variables (fixed effects), and *student* as a random effect. As shown in Table III, *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 tasks as 2 dummy variables (also known as indicator variables) to represent distinct categories [60], 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 students' submissions in the three tasks. As shown in Table III, Model A shows that only being in the `Hint` condition significantly improves students' scores (p = 0.04), such that students who had access to hints performed on average $\sim$7.76% points higher than that of the `Control` students across all the 3 in-class programming tasks.

For the *Daisy* homework task we found that students in the `Hints` condition have higher scores (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; Effect size r = 0.1), possibly due to a ceiling effect[4].

Time on Task: To measure the time students spent on solving in-class programming assignments, we combined students' active time (as discussed in Section IV-D1) on the in-class assignments: *PolygonMaker*, *Squiral*, *Guessing Game*. Similar to Model A, we used a linear mixed-effects model controlling 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 Model B in Table III, 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, as discussed in Section VI-B.

For the *Daisy* homework task we 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`

[4]A ceiling effect happens when a large percentage of observations score near the "upper" limit on a test or an assignment (in our case the upper limit was 100%). This ceiling effect makes it difficult to compare the difference between the means of the control and Hint group.

TABLE (III) Estimated coefficients (Standard Error) of linear mixed-effects (LME) models predicting students' scores on programming assignments (Model A), and students' active time spent on programming tasks (Model B), respectively.

	Model A		**Model B**	
	Coeff (SE)	p-value	Coeff (SE)	p-value
Intercept	83.64 (2.66)	<0.001	19.94 (1.77)	<0.001
Condition	7.76 (3.8)	0.043	1.61 (2.5)	0.52
Polygon	6.35 (3.34)	0.06	-6.22 (2.14)	<0.01
Squiral	-0.45 (3.24)	0.88	9.87 (2.21)	<0.001
Observations	160			

group (M = 62.11; Med = 44; SD = 46.22), and a Mann-Whitney U test showed that this difference is significant (p = 0.01; Effect size r = -0.31). As a post-hoc analysis, we measured students' programming efficiency to investigate whether this decrease in time affected students' performance on the programming homework. We measured programming efficiency by calculating: students' performance score (in %) / active time spent (in minutes). We 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 medium effect size (Effect size r = 0.33). 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 hints improved students' programming efficiency in a homework task, where no pair programming took place and with less help from the instructors and peers.

B. RQ2 - Learning:

We first investigated learning by comparing students' performance on HW2 *BrickWall* where all students did not have access to hints. We found that students in the `Hints` group have higher scores (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; Effect size r = 0.12), possibly due to a ceiling effect (the median score was 100%). In terms of active time, we found that the `Hints` group completed the *Brick Wall* 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.

We then compared students' scores in the post-test (i.e. Quiz 1), where a total of 53 students took the post-test; 26 in the `Hints` group, and 27 in the `Control` group. We 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; Effect size r = 0.001). A review of individual post-test problems shows that this result was generally consistent across problems. While this result does not show that 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.

C. RQ3 - 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, we 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 our prior work that shows that students prefered next-step hints with textual explanations [16], [11], even if they do not read the hint explanations, and that prompting students to self-explain is not prefered by most students [11]. Second, when students were asked to rate (on a scale from 1 to 5) the overall usefulness of hints, we found that 77% rated 3 or above. This shows that overall students in the `Hints` group found that hints were helpful. Below we report themes on why students' perceived hints as helpful or not, and why they preferred specific design choices. We present quotes from students' responses by adding an anonymous student ID preceding their quote (e.g. [S1] means student 1).

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 we found in the *Guessing Game* task (shown in Table II, which was mainly due to the lower hint quality as we will discuss in the next Section.

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

TABLE (IV) 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 answers
Polygon Maker	24/25	45.83%	26	88.46%	69.23%
Squiral	28/29	51.7%	82	78.1%	96.34%
Guessing Game	26/27	42.3%	62	45.16%	98.38%
Daisy	21/26	47.6%	71	77.46%	87.32%

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* [16]. 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, we 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]."* We 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). We argue that this opens new directions for improving the design of SE prompts, that can further reinforce their impact on students' outcomes in programming.

D. RQ4 - Effect of Problem Difficulty & Hint Quality

Problem difficulty: As shown in Model A in Table III, we found no main effect of any of the programming assignments ($p > 0.05$) on students' scores, suggesting that *overall* the assignments were similarly difficult . To investigate whether the effectiveness of hints was mediated by the assignment itself, we investigated students' scores on each task separately. As shown in Table II, we found that the `Hint` group has higher scores than the `Control` group in all assignments except *Guessing Game*, but that the magnitudes vary (e.g. 15 points higher on *Polygon Maker*, but only 3 on *BrickWall*). This suggests that hints may have been more useful on some assignments than others, and may not be similarly effective on every assignment. We discuss why hints may not have helped on the *Guessing Game* task below.

Hint quality: 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, as described in Section III-D. 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. As a result, we investigated hint quality through students' hint usage.

To investigate students' hint usage, we first looked at the number of hints opened by students in the `Hints` condition. As shown in Table IV, the majority of students received at least one hint button in all assignments; and, 42.3% - 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 [16], [22] 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 [5]. To investigate if the opened hints were useful and interpretable to students (i.e. a second measure of hint quality), we measured the hints' follow rate (as explained in Section IV-D5). As shown in Table IV, 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, we found only 45.16% of hints were followed.

Digging further into *Guessing Game* hints, we found three reasons for this low hint follow rate. First, as shown in Figure 2, 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* task, 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 on student self-explained a hint saying: "I don't think this is necessary because my code works!". Third, some students attempted to break the *Guessing Game* task into several subparts, and then combine them at the end (often called a prototyping behavior – a common tinkering behavior in programming [61]). 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 data-driven hint generation system, not only in the iSnap system, but it can also occur in other programming tutors, which require future improvements to their underlying

TABLE (V) Examples of self-explanation prompts and students' answers to these prompts.

Task	SE prompt	Student informative answer
Polygon Maker	Why do you think Snap recommended this hint?	I need to tell what thickness the pen needs to be
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
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?	Have the number or petals adjust so the conditional function detects the shift

algorithms. Unlike the *Guessing Game* assignment, when we investigated the hints opened by students in the other assignments (that have a high hint follow rate), we 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 task it is generated for.

Our last measure in hint usage is investigating students' answers to the self-explanation prompts, as a way to understand how such SEs might have impacted students' understanding of the hints. As explained in Section IV-D5, we calculated how many SE answers were informative versus uninformative. Table IV 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 [11], [17]. Table V shows an example of SE prompts, and some of students' informative and informative answers.

VI. Discussion

A. RQ1-RQ3

In classroom settings, what is the impact of the enhanced next-step hints on students' (RQ1) performance, (RQ2) learning and (RQ3) how do students perceive these hints? For **performance**, we 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 our prior work [11], and suggest that hints accomplish their primary purpose of helping students progress [18]. Even though hints required students to take time to read and answer the SE prompts, we found that the `Hints` group achieved higher scores and submitted their code faster on the *Daisy* homework. Taking these results together - higher scores on in-class assignments, and faster completion on HWs, we might summarize this by saying that the enhanced next-step hints led to an increase in students' programming *efficiency*.

Looking into performance in terms of students' active time spent on the programming tasks, we found a consistent trend, where the `Hints` group finished faster than the `Control` group in all tasks except in *Squiral*, as shown in Table II. Digging further into student log data and hint usage in *Squiral*, we 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.

It is important to note that the enhanced next-step hints were accompanied with textual explanations and SE prompts and were provided in an adaptive hint display. Our results suggest that these design features are *effective* for creating impactful next-step hints, and our prior work [11] suggests that they may even be *necessary* to see longer-term impact. At the end of this section, we provide a more detailed discussion about the consistency of our findings across studies. From a theoretical perspective, we explain how such design features improved the impact of hints: First, *textual explanations* of the next-step hints incorporate declarative knowledge into the hint, which helps the student to apply and learn from it, according to the ACT-R cognitive theory [28]. Second, prompting students to *self-explain* hints might have encouraged them to stop and think to make sense of the hint, rather than quickly skimming it. This process of "sense making" of hints is essential for learning, as suggested by the KLI framework [17]. 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 [32], [62]. This supports Wood's theory of *contingent tutoring*, which emphasized that the tutor's help should be contingent upon the learner's needs to improve their learning [32].

For **learning**, we found the `Hints` group performed better on *BrickWall* (i.e. the assessment task), and similarly in the post-test, with insignificant differences, suggesting inconclusive findings about whether hints improved learning. We suggest two possible reasons for this result. First, students did uniformly well on the assessment task (median grade 100%, 87.5% of students fully completed the task), perhaps because it came after 3 weeks of programming practice, which might 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 [32], [30], [18]. Regardless, it seems clear that hints did not *impede* learning, since prior work shows that hints, which give away part of the answer, can reduce student learning, for example if help abuse occurs [18], [30]. 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,** overall, we found 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 [63], or because they do not like doing the extra work [11]. While one can argue that hints with SE prompts can *not* be helpful if students do not prefer to answer the prompts, we 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 [64].

Last but not least, in terms of performance, time, and students' perceptions, the current study results are consistent with (i.e. replicated) our prior findings with a population of recruited participants during an online one-hour study in a laboratory setting [11]. However, we did not find a similar learning effect to what we found in our 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 effect. Overall, this suggests that the effect of enhanced next-step hints is somewhat robust across multiple populations and learning contexts.

B. RQ4

How do problem difficulty and hint quality mediate the effect of enhanced next-step hints on students' performance? In terms of the effect of problem type (or difficulty), overall, we found that the hints improved overall student performance on in-class and homework programming tasks; as shown in Model A in Table III. This suggests that the programming assignment complexity does not mediate the impact of hints, since we 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 couple of hours at most, and so our results do not speak to the impact of hints in more advanced programming courses.

In terms of the impact of hints' quality, we found that hints provided in *Guessing Game* programming task were not of high quality, compared to the other tasks, as discussed in Section V-D, 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 their *quality* (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 [13]." 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.

C. Design Implications

Our results have some implications for how future educational technologies should be designed to support novice programmers. Each of the design choices discussed in Sections III-B, III-C, and III-D could likely contribute to the hints' helpfulness, which we have also found in our prior work [11], [22], [16]. It might therefore be worth applying such design features to other, somewhat similar, automated support (such as worked examples, compiler messages, or autograder test cases) to enhance their impact on students' performance and learning in programming labs and homework tasks. For example, autograder test cases could be enhanced by adding self-explanation prompts that can improve students' learning from the test cases. Additionally, compiler errors could be enhanced by adding textual explanations (as in enhanced compiler error messages [65], [66]). Furthermore, the use of adaptive hint display has shown promising impact in this work as well as in our prior work [22], which can be applied in other classrooms whether automated help is available or not. For example, by developing a system that encourages/reminds students to ask for help from instructors, or message boards based on their logged scores and performance throughout the semester.

This work does not only explore the effect of some design choices, but it also presents the challenge of designing automated technology, and adapting this design for deployment in classrooms. Our results show (as discussed in Section VI-A) that classrooms have several challenges that can make it hard to see effects of automated technology on students' programming performance and learning. Such as the use of pair programming, the distribution of instructor/TA help, students' prior knowledge, etc. Even if these challenges were under control, one implication of this work is that some help may not lead to measurably improved learning, maybe because it is hard to measure (such as measuring learning after several weeks where students might have already practiced well). Last but not least, one recommendation for future classrooms, regardless of the programming language used, is that researchers need to investigate the quality of hints used (whether authored by experts or generated by algorithms) before applying them in classrooms. This is important since our work shows that hints with poor quality can have a negative effect on students' performance (as discussed in Section VI-B).

VII. Limitations and Conclusion

This study suffers some limitations, many 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 we 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, pair programming is an effective collaborative strategy that might be a reason that students did better in *both* conditions. However, since pair

programming took place in both the `Hints` and `Control` group, any differences we see should be attributable to hints, not pair programming. It is possible that pair programming *mediated* the effect of hints, allowing them to be more effective than individual work (e.g. by providing a context for discussing the hints, though the reverse is also possible (e.g. if pair programming created a larger ceiling effect, with both groups doing well)). Fourth, as discussed in Section V, in our statistical analysis, we did not correct for multiple comparisons, and we encourage the reader to weigh our evidence in light of the 7 comparisons we performed, as well as the education theory and prior work discussed in Section II. Finally, this study took place online and during the COVID-19 pandemic, which included additional challenges for students. While these challenges would have affected both conditions of the study (and therefore would not be responsible for detected differences), it may affect the generalizability of the study.

In summary, this paper presented the evolution of the iSnap system, which provides data-driven next-step hints enhanced with textual explanations, self-explanation prompts, and an adaptive hint display, to address common limitations of data-driven next-step hints. Overall, the contributions of this work are: (1) a presentation of iSnap's enhanced next-step hints to improve hints' effectiveness; (2) a controlled study in an authentic classroom setting showing that enhanced hints improved students' programming performance in in-class assignments, and programming efficiency in homework assignments; (3) findings on how students' usage of hints is affected by the hints' quality and the programming task; and (4) a conceptual replication of prior evaluations of the iSnap system, and a comparison of results, largely confirming their generalizability across different populations and learning contexts.

REFERENCES

[1] Simon, A. Luxton-Reilly, V. V. Ajanovski, E. Fouh, C. Gonsalvez, J. Leinonen, J. Parkinson, M. Poole, and N. Thota, "Pass rates in introductory programming and in other stem disciplines," in *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, 2019, pp. 53–71.

[2] J. Bennedsen and M. E. Caspersen, "Failure rates in introductory programming: 12 years later," *ACM Inroads*, vol. 10, no. 2, pp. 30–36, 2019.

[3] J. Gorson and E. O'Rourke, "Why do cs1 students think they're bad at programming? investigating self-efficacy and self-assessments at three universities," in *Proceedings of the 2020 ACM Conference on International Computing Education Research*, 2020, pp. 170–181.

[4] R. Butler, "Determinants of help seeking: Relations between perceived reasons for classroom help-avoidance and help-seeking behaviors in an experimental context." *Journal of Educational Psychology*, vol. 90, no. 4, p. 630, 1998.

[5] T. W. Price, Z. Liu, V. Catete, and T. Barnes, "Factors Influencing Students' Help-Seeking Behavior while Programming with Human and Computer Tutors," in *Proceedings of the International Computing Education Research Conference*, 2017.

[6] A. Mitrovic, S. Ohlsson, and D. K. Barrow, "The effect of positive feedback in a constraint-based intelligent tutoring system," *Computers & Education*, vol. 60, no. 1, pp. 264–272, 2013.

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

[8] A. Gerdes, B. Heeren, J. Jeuring, and L. T. van Binsbergen, "Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback," *International Journal of Artificial Intelligence in Education*, vol. 27, no. 1, pp. 1–36, 2016.

[9] K. Rivers and K. R. Koedinger, "Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor," *International Journal of Artificial Intelligence in Education*, vol. 27, no. 1, pp. 37–64, 2017.

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

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

[12] D. Fossati, B. Di Eugenio, S. Ohlsson, C. Brown, and L. Chen, "Data driven automatic feedback generation in the ilist intelligent tutoring system," *Technology, Instruction, Cognition and Learning*, vol. 10, no. 1, pp. 5–26, 2015.

[13] T. Price, R. Zhi, Y. Dong, N. Lytle, and T. Barnes, "The impact of data quantity and source on the quality of data-driven hints for programming," in *Proceedings of the International Conference on Artificial Intelligence in Education*, 2018.

[14] B. Hartmann, D. Macdougall, J. Brandt, and S. R. Klemmer, "What Would Other Programmers Do? Suggesting Solutions to Error Messages," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2010, pp. 1019–1028.

[15] T. W. Price, Y. Dong, and D. Lipovac, "iSnap: Towards Intelligent Tutoring in Novice Programming Environments," in *Proceedings of the ACM Technical Symposium on Computer Science Education*, 2017.

[16] S. Marwan, N. Lytle, J. J. Williams, and T. Price, "The impact of adding textual explanations to next-step hints in a novice programming environment," in *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, 2019, pp. 520–526.

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

[18] V. Aleven, I. Roll, B. M. McLaren, and K. R. Koedinger, "Help helps, but only so much: Research on help seeking with intelligent tutoring systems," *International Journal of Artificial Intelligence in Education*, vol. 26, no. 1, pp. 205–223, 2016.

[19] K. Rivers, "Automated Data-Driven Hint Generation for Learning Programming," PhD, Carnegie Mellon University, 2017.

[20] W. Wang, C. Zhang, A. Stahlbauer, G. Fraser, and T. Price, "Snapcheck: Automated testing for snap programs," ser. ITiCSE'21, to appear. Association for Computing Machinery, 2021.

[21] L. Gusukuma, D. Kafura, and A. C. Bart, "Authoring feedback for novice programmers in a block-based language," in *2017 IEEE Blocks and Beyond Workshop (B&B)*. IEEE, 2017, pp. 37–40.

[22] S. Marwan, A. Dombe, and T. W. Price, "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*, 2020, pp. 54–60.

[23] S. Marwan, G. Gao, S. Fisk, T. Price, and T. Barnes, "Adaptive immediate feedback can improve novice programming engagement and intention to persist in computer science," in *Proceedings of the International Computing Education Research Conference*, 2020, p. 194–203.

[24] P. Shabrina, S. Marwan, T. W. Price, M. Chi, and T. Barnes, "The impact of data-driven positive programming feedback: When it helps, what happens when it goes wrong, and how students respond," in *Educational Data Mining in Computer Science Education Workshop @ EDM*, 2020.

[25] T. Price, S. Marwan, J. Williams, and M. Winters, "An evaluation of data-driven programming hints in a classroom setting," in *Proceedings of the International Conference on Artificial Intelligence in Education (forthcoming)*, 2020.

[26] V. J. Shute, "Focus on formative feedback," *Review of educational research*, vol. 78, no. 1, pp. 153–189, 2008.

[27] R. G. M. Hausmann, A. Vuong, B. Towle, S. H. Fraundorf, R. C. Murray, and J. Connelly, "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)*, vol. 7926 LNAI, pp. 791–794, 2013.

[28] J. Anderson, "ACT: A simple theory of complex cognition." *American Psychologist*, 1996.

[29] J. Whitehill, "Understanding act-r-an outsider's perspective," *arXiv preprint arXiv:1306.0125*, 2013.

[30] V. Aleven, "Help Seeking and Intelligent Tutoring Systems: Theoretical Perspectives and a Step Towards Theoretical Integration," *International Handbook of Metacognition and Learning Technologies*, vol. 28, no. January, pp. 197–211, 2013.

[31] V. Aleven, B. Mclaren, I. Roll, and K. Koedinger, "Toward meta-cognitive tutoring: A model of help seeking with a cognitive tutor," *International Journal of Artificial Intelligence in Education*, vol. 16, no. 2, pp. 101–128, 2006.
[32] H. Wood and D. Wood, "Help seeking, learning and contingent tutoring," *Computers & Education*, vol. 33, no. 2-3, pp. 153–169, 1999.
[33] L. Gusukuma, A. C. Bart, D. Kafura, and J. Ernst, "Misconception-driven feedback: Results from an experimental study," in *Proceedings of the 2018 ACM Conference on International Computing Education Research*, 2018, pp. 160–168.
[34] D. Toll, A. Wingkvist, and M. Ericsson, "Current state and next steps on automated hints for students learning to code," in *2020 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2020, pp. 1–5.
[35] M. Suarez and R. Sison, "Automatic construction of a bug library for object-oriented novice java programmer errors," *Intelligent Tutoring Systems*, 2008.
[36] S. Gross, B. Mokbel, B. Paassen, B. Hammer, and N. Pinkwart, "Example-based feedback provision using structured solution spaces," *International Journal of Learning Technology*, vol. 9, no. 3, p. 248, 2014.
[37] T. Lazar and I. Bratko, "Data-Driven Program Synthesis for Hint Generation in Programming Tutors," in *Proceedings of the International Conference on Intelligent Tutoring Systems*. Springer, 2014, pp. 306–311.
[38] J. Stamper and T. Barnes, "The hint factory: Automatic generation of contextualized help for existing computer aided instruction," in *Proceedings of the 9th International Conference on Intelligent Tutoring Systems Young Researchers Track*, 2008.
[39] T. W. Price, R. Zhi, and T. Barnes, "Hint Generation Under Uncertainty: The Effect of Hint Quality on Help-Seeking Behavior," in *Proceedings of the AIED Conference*, 2017.
[40] A. Hicks, B. Peddycord, and T. Barnes, "Building games to learn from their players: Generating hints in a serious game," in *International Conference on Intelligent Tutoring Systems*. Springer, 2014, pp. 312–317.
[41] T. W. Price, Y. Dong, R. Zhi, B. Paaßen, N. Lytle, V. Cateté, and T. Barnes, "A Comparison of the Quality of Data-driven Programming Hint Generation Algorithms," *International Journal of Artificial Intelligence in Education*, 2019.
[42] R. R. Choudhury, H. Yin, and A. Fox, "Scale-driven automatic hint generation for coding style," in *International Conference on Intelligent Tutoring Systems*. Springer, 2016, pp. 122–132.
[43] A. Ahadi, A. Hellas, P. Ihantola, A. Korhonen, and A. Petersen, "Replication in computing education research: researcher attitudes and experiences," in *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, 2016, pp. 2–11.
[44] M. T. Chi, M. Bassok, M. W. Lewis, P. Reimann, and R. Glaser, "Self-explanations: How students study and use examples in learning to solve problems," *Cognitive science*, vol. 13, no. 2, pp. 145–182, 1989.
[45] A. Head, E. Glassman, G. Soares, R. Suzuki, L. Figueredo, L. D'Antoni, and B. Hartmann, "Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis," in *Proceedings of the ACM Conference on Learning @ Scale*. ACM, 2017, pp. 89–98.
[46] A. Renkl and R. K. Atkinson, "Learning from examples: Fostering self-explanations in computer-based learning environments," *Interactive learning environments*, vol. 10, no. 2, pp. 105–119, 2002.
[47] J. J. Williams, T. Lombrozo, A. Hsu, B. Huber, and J. Kim, "Revising learner misconceptions without feedback: Prompting for reflection on anomalies," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI '16. New York, NY, USA: ACM, 2016, pp. 470–474.
[48] C. Conati and K. VanLehn, "Toward Computer-based Support of Meta-cognitive Skills: A Computational Framework to Coach Self-explanation," *International Journal of Artificial Intelligence in Education*, vol. 11, no. 1, pp. 389–415, 2000. [Online]. Available: http://telearn.archives-ouvertes.fr/hal-00197335/
[49] A. Vihavainen, C. S. Miller, and A. Settle, "Benefits of Self-explanation in Introductory Programming," *Proceedings of the 46th ACM Technical Symposium on Computer Science Education - SIGCSE '15*, vol. 68, pp. 284–289, 2015.
[50] V. Aleven, E. Stahl, S. Schworm, F. Fischer, and R. Wallace, "Help Seeking and Help Design in Interactive Learning Environments Vincent," *Review of Educational Research*, vol. 73, no. 3, pp. 277–320, 2003.
[51] Q. Hao, D. H. Smith IV, N. Iriumi, M. Tsikerdekis, and A. J. Ko, "A systematic investigation of replications in computing education research," *ACM Transactions on Computing Education (TOCE)*, vol. 19, no. 4, pp. 1–18, 2019.
[52] D. Weintrop and U. Wilensky, "Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs." in *ICER*, vol. 15, 2015, pp. 101–110.
[53] J. Bruin. (2011 (accessed April 6, 2020), Feb.) Introduction to linear mixed models. [Online]. Available: https://stats.idre.ucla.edu/stata/ado/analysis/
[54] C. O. Fritz, P. E. Morris, and J. J. Richler, "Effect size estimates: current use, calculations, and interpretation." *Journal of experimental psychology: General*, vol. 141, no. 1, p. 2, 2012.
[55] M. Maguire and B. Delahunt, "Doing a thematic analysis: A practical, step-by-step guide for learning and teaching scholars." *The All Ireland Journal of Teaching and Learning in Higher Education*, vol. 9, 2017.
[56] L. Margulieux and R. Catrambone, "Using Learners' Self-Explanations of Subgoals to Guide Initial Problem Solving in App Inventor," pp. 21–29, 2017.
[57] K. J. Rothman, "No adjustments are needed for multiple comparisons," *Epidemiology*, pp. 43–46, 1990.
[58] T. V. Perneger, "What's wrong with bonferroni adjustments," *Bmj*, vol. 316, no. 7139, pp. 1236–1238, 1998.
[59] A. D. Althouse, "Adjust for multiple comparisons? it's not that simple," *The Annals of thoracic surgery*, vol. 101, no. 5, pp. 1644–1645, 2016.
[60] S. Skrivanek, "The use of dummy variables in regression analysis," *More Steam, LLC*, 2009.
[61] Y. Dong, S. Marwan, V. Catete, T. Price, and T. Barnes, "Defining tinkering behavior in open-ended block-based programming assignments," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, 2019, pp. 1204–1210.
[62] R. C. Murray and K. VanLehn, "A comparison of decision-theoretic, fixed-policy and random tutorial action selection," in *International Conference on Intelligent Tutoring Systems*. Springer, 2006, pp. 114–123.
[63] M. Roy and M. T. Chi, "The self-explanation principle in multimedia learning," *The Cambridge handbook of multimedia learning*, pp. 271–286, 2005.
[64] B. Shih, K. R. Koedinger, and R. Scheines, "A response time model for bottom-out hints as worked examples." *Proceedings of the 1st International Conference on Educational Data Mining, EDM*, 2008.
[65] B. A. Becker, G. Glanville, R. Iwashima, C. McDonnell, K. Goslin, and C. Mooney, "Effective compiler error message enhancement for novice programming students," *Computer Science Education*, vol. 26, no. 2-3, pp. 148–175, 2016.
[66] B. A. Becker, K. Goslin, and G. Glanville, "The effects of enhanced compiler error messages on a syntax error debugging test," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 2018, pp. 640–645.

Samiha Marwan is a Computing Innovation (CI) Fellow and a Postdoctoral Researcher at the University of Virginia since March 2022. Samiha completed her PhD at NC State University in 2021. Her primary research focus is on developing intelligent support features in block-based programming languages to improve novices' cognitive and affective outcomes.

Thomas W. Price is an Assistant Professor of Computer Science at NC State University. He directs the Help through INTelligent Support (HINTS) Lab, which develops learning environments that automatically support students through AI and data-driven help features. Dr. Price has been recognized by the STARS Computing Corps for his leadership in computing outreach.