An Evaluation of the Impact of Automated Programming Hints on Performance and Learning

Samiha Marwan
North Carolina State University
Raleigh, North Carolina
samarwan@ncsu.edu

Joseph Jay Williams
University of Toronto
Toronto, Canada
williams@cs.toronto.edu

Thomas Price
North Carolina State University
Raleigh, North Carolina
twprice@ncsu.edu

ABSTRACT

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

ACM Reference Format:
Samiha Marwan, Joseph Jay Williams, and Thomas Price. 2019. An Evaluation of the Impact of Automated Programming Hints on Performance and Learning. In *International Computing Education Research Conference (ICER '19), August 12–14, 2019, Toronto, ON, Canada.* ACM, Toronto, Canada, 10 pages. https://doi.org/10.1145/3291279.3339420

1 INTRODUCTION

Computer Science has the highest dropout rate of any STEM B.S. degree in the U.S. [15]. Half of that dropout occurs in students' first year, with multiple studies estimating the rate of students failing their first CS course at 33% [13, 55]. Many studies have highlighted the positive influence that feedback can have on students' learning and motivation in computer science [20, 34, 56]. Thus, researchers have developed a number of tools to provide automated feedback to support novice programmers, including enhanced compiler messages [10], positive feedback [21], and on-demand hints [41, 48].

On-demand hints are particularly promising, since they can be generated automatically (e.g. using student data [45, 48]), allowing them to scale to new problems and contexts. These automated hints are frequently *edit-based, next-step* hints, which suggest an edit that the student can make to bring their code closer to a correct solution. This can be conveyed through textual instructions (e.g. [48]), or by showing a "diff," contrasting the student's code with the suggested code (e.g. [25, 41, 56]). In this work, we refer to these as *code hints*. Figure 1 shows an example of how a code hint is displayed in one block-based programming environment, iSnap [41].

However, a great deal is not known about how code hints impact learners. For example, how do learners perceive code hints? How do code hints impact learning? How can we improve the usability and impact of code hints? A number of small-scale studies have examined user logs in real classrooms, which provide some indication that code hints can help students when they get stuck during programming [41, 47]. However, previous work also suggests that students sometimes find it difficult to interpret why code hints are relevant without having additional explanations [32]. Other research argues these hints may not lead to learning as they give away part of the correct solution [3, 37]. The challenge is that few studies have systematically compared novices doing programming tasks with and without code hints, such as by conducting randomized experiments. In addition, classroom studies typically do not gather extensive data about users' experiences through surveys and semi-structured interviews [9].

This paper aims to evaluate the impact of code hints, and investigate how they can be improved, in the context of block-based programming. Specifically, we investigated whether code hints can be improved with two complementary features: textual explanations, and prompts for students to self-explain the hint in their own words [50, 51]. We investigated students' subjective experiences through interviews (Study 1), and conducted a randomized, controlled experiment to measure the effects of hints on immediate performance, as well as learning, as measured by success on a similar programming task without hints (Study 2).

Study 1 was a pilot study with 10 undergraduate, novice programmers, who received hints with different combinations of additional features (textual explanations and self-explanation prompts), and afterwards, we interviewed them about their experiences. The qualitative data suggested that students appreciated code hints, as these

ICER '19, August 12–14, 2019, Toronto, ON, Canada

ACM ISBN 978-1-4503-6185-9/19/08…$15.00
https://doi.org/10.1145/3291279.3339420

showed concrete next steps and guidance about the right direction. Students reported additional benefits of receiving textual explanations, such as elaborating the logic behind a code hint. On the other hand, students expressed a variety of opinions on self-explanation prompts, with some saying that they were confusing, and others appreciating that the prompts guided them to think deeper about the hint and how their code worked.

To further investigate the impact of hints, Study 2 was a randomized, controlled experiment that compared: (1) No hints; (2) Code hints with textual explanations; (3) Code hints with textual explanations *and* prompts to self-explain the hints. The study was conducted as an online laboratory-style study, rather than a classroom study. We achieved a larger samples and more experimental control by recruiting a sample of 250 Mechanical Turk workers who were novices in programming. While this population differs from students in many ways, they have been suggested as an appropriate alternative to university participants for lab studies [11, 27] and even have been found to behave similarly to online education learners in MOOCs in some situations [18]. Using this population allowed us to conduct a randomized, controlled experiment with a large number of people, which would have been harder to justify in a classroom setting with high stakes outcomes [9]. Study 2 suggested that code hints with textual explanations (with or without self-explanation prompts) improved immediate programming performance, relative to no hints. In addition, we found that learning – as measured by performance on a second task where hints were not provided – was only increased when self-explanation prompts accompanied code hints and textual explanations. This improvement was also limited to objectives on the second task that closely resembled objectives from the first task.

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

2 RELATED WORK

An increasing number of systems offer automatically generated code hints, usually targeting novice programmers [23, 25, 26, 29, 38, 41, 48]. These hints can be generated using data-driven methods, using student or expert-authored data [26, 30, 45, 48], or using program generation techniques from software engineering (e.g. test-driven synthesis [38], automated program repair [58]). These systems generally offer hints to students as they write code for programming assignments, which requires adaptively supporting a variety of coding approaches and situations. Because these hints are generated automatically, authors have pointed to their potential to provide scalable feedback to a large number of assignments without requiring instructors to author hints manually [45, 48]. However, because they are not authored by instructors, these hints are generally limited to suggesting edits to the student's code, without an explanation [43, 47]. This differentiates code hints from other forms of instructor-authored, automated programming feedback, such as autograder messages (e.g. [8, 36]), enhanced compiler error messages (e.g. [10]), or misconception-driven feedback [24]. A primary goal of this work is to evaluate whether these adaptive and easily generated code hints can also lead to better outcomes for students, and to determine whether other forms of feedback (e.g. textual explanations, self-explanation prompts) can improve them.

2.1 Theoretical and Empirical Justifications.

Theoretical justification for the use of hints in learning environments grows out of early work on cognitive tutoring systems [7], which were designed around the ACT-R cognitive theory [5]. Many of these early tutoring systems, such as the Act Programming Tutor [17] targeted programming, though the theory is not domain-specific. ACT-R stipulates that problem solving requires 2 types of interrelated knowledge: declarative knowledge, which can be represented verbally (knowing *that*, e.g. "what is a variable"), and procedural knowledge, which encodes how to solve a specific problem step (knowing *how*, e.g., "when and how do I declare a variable"). Under ACT-R, problem solving practice emphasizes building procedural knowledge through repeated application of production rules. Principle-based hints that explain a domain concept can therefore be seen as helping students to simultaneously acquire declarative knowledge and contextualize their procedural knowledge accordingly [1]. This helps to prevent students from learning overly specific procedural knowledge (e.g. "a for-loop counter always starts at 0") by providing a means to abstract their problem-specific actions to more general rules. The Knowledge-Learning-Interaction (KLI) framework [28] suggests that hints can prompt this acquisition of verbal (declarative) knowledge by engaging the student in sense-making, or the process of reasoning and constructing explanations.

Aleven and Koedinger argue that this self-explanation process is an essential component of learning from hints, but it is unlikely to occur spontaneously [3]. While principle-based hints can help students build declarative knowledge through sense-making, many automated programming hints are bottom-out hints, saying only what to do but not why (e.g. [25, 41, 48]). Under ACT-R, these serve the primary purpose of getting a student to complete a step or problem more efficiently, as the unsuccessful search for a correct answer is unlikely to result in learning [6]. As explained in Section 3, our design of automated hints draws on these theoretical perspectives by including both textual explanations (similar to principle-based hints), and prompts to encourage students to self-explain the hints.

Empirical evaluations of the specific benefits of code hints. Despite the growing body of work on automatically *generating* programming hints (c.f. [42]), few empirical studies have measured their impact on students' performance and learning. Instead, many evaluations have focused on technical aspects of the hint generation process, such as whether the generated hints are able to resolve known errors in students' code (e.g. [23, 29, 54]), or expert-ratings of hint quality (e.g. [39, 40, 45]). Other work has used laboratory studies to investigate how these hint systems impact students' perceptions. In [43], Price et al. studied students as they completed 2 programming assignments, first aided by a human tutor and then by automated hints. The authors found that automated hints were perceived as quick and easy to use, but were also perceived as less

perceptive and interpretable than human help. Perhaps the most related work comes from Rivers [47], whose dissertation describes a user study conducted with the ITAP tutoring system, which offers data-driven hints for Python programming. Rivers found that less experienced programmers wanted more detailed content in their hints, compared with more experienced programmers, and that most users wanted hints to give as little information as possible, while allowing the option to see more detail if necessary.

A few studies do point to code hints' potential to impact learning; however, none of these studies investigated students completing whole programming tasks. Corbett and Anderson [17] found that on-demand hints in their structured ACT Programming tutor, which led students through a programming problem one step at a time, improved students' performance on a post test. Fossati et al. found that versions of their iList linked list tutor that offered feedback, including on-demand hints, produced improved student learning [21]. Additionally, Choudhury et al. [16] found that students with access to their code style hints produced significantly better-quality solutions than students who did not. Only Rivers [47] compared students with and without access to hints during programming assignments in a classroom context. However, she found no significant difference between the conditions.

Our work builds on these prior evaluations with new elements: empirically investigating the impact of code hints, in *combination* with different supporting features, on learners' preferences, performance and learning, while completing whole programming tasks.

3 DESIGN OF HINT SUPPORT

In this work, we built on an existing system called iSnap [41], a block-based novice programming environment that supports students with code hints. The hints are generated by a data-driven algorithm [45], which uses a database of correct solutions for a given problem to auto-generate hints. The algorithm identifies a solution that closely matches the structure of the student's current code and suggests an edit to the student's code that will bring it closer to that solution. When the system has a hint available, it annotates the student's code with a "HINT" button, as shown in Figure 1 (top). When clicked, the system displays a hint dialog, as shown in Figure 1 (blue box) with the suggested edit. This is communicated through a visual "code hint," which contrasts the student's current code with suggested code. In this work, we evaluated 2 additional features of the system's hints: textual explanations and self-explanation prompts. Both of these features were designed to overcome limitations of code hints, identified in prior work, as explained below.

First, as Gusukuma et al. note, code hints suggest only *what* the student should do, not *why* [24], and prior work suggests that this can make hints difficult for students to interpret [43]. Further, if a primary role of hints is to help students to contextualize their actions with domain knowledge, as Aleven suggests [1], then code hints alone may not facilitate learning. In our previous work we addressed this limitation by adding textual explanations to code hints [32][1]. An example is shown in Figure 1 under the label "Text Hint." These explanations were designed to complement a given code hint by conveying: 1) where to find the suggested block, 2) what the block does, and 3) how it is useful for the given assignment. In this prior work [32], we found that students appreciated the textual explanations, and that they led to improved ability to explain the purpose of hints, but they did not have any significant *additional* impact on programming performance compared to code hints alone.

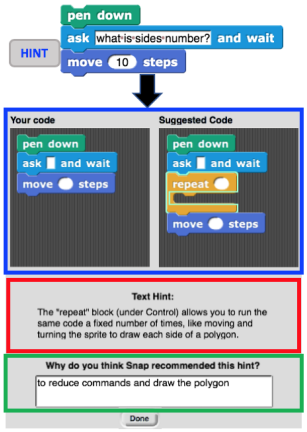

Figure 1: iSnap displays hint button (top). When clicked, it shows a code hint (blue box), textual explanation (red box) and self-explanation prompt (green box).

A second limitation of automated code hints is that they are effectively "bottom-out" hints, telling the student exactly what to do, without requiring them to reason about the information. Prior work suggests that while such bottom out hints are necessary to help students who are stuck, they likely only lead to learning when students spontaneously self-explain the hint, which is rare [3, 51]. To address this, we designed self-explanation prompts, as shown at the bottom of Figure 1. These prompts randomly show one of a variety of messages, such as "Why do you think the system recommended this hint?" and "What is this hint trying to help you to understand or do?" that encourage the student to think critically about the hint itself and its relation to their code. The self-explanation prompt is open-ended, and users can write anything in the response field, with at least 20 characters in order to close the hints dialog. In domains other than programming, several studies suggest that self-explanations can benefit learning (e.g. in mathematics [57]), and such self-explanation prompts were particularly useful for low prior knowledge students in a biology class [33]. However, there is also evidence that learners may be distracted or frustrated by such prompts [52], and there is comparatively less work exploring the generation of self-explanations in programming [35, 53].

4 STUDY 1: PERCEPTIONS OF HINTS

Our goal with this study was to understand students' subjective perceptions of code hints, textual explanations and self-explanation prompts, specifically when and how they are helpful, and how they can be improved. We ran a pilot study with undergraduate students at a research university where students programmed with different combinations of hint support, and we conducted interviews with

[1] In our current implementation, these explanations are written manually and selected automatically [32]. The hints evaluated in this work are therefore technically "semi-automated." However, our goal in this work was to explore and evaluate possible ways to improve code hints, and we leave questions of generation for future work.

students afterwards. We identified key themes from the interviews, which informed our subsequent study and offer design implications for how and when to use different kinds of hint support.

4.1 Methods

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

Programming Environment: All programming in this study took place inside of iSnap[41]. The system automatically offered a hint every 2 minutes by annotating the student's code with a hint button (Figure 1, top). This hint automatically updated as the student edited their code. This student was free to click on the hint immediately, or wait. Hints accumulated over time, such that if a student did not ask for any hints for 4 minutes, the student could then request 2 hints in a row. This setup was intentionally designed to address challenges evaluating hints in prior work, where many students either avoided hints [4, 46, 47], or abused them by repeatedly requesting them [2]. By proactively showing hint buttons every 2 minutes, the system encouraged frequent help use without forcing it, and the 2 minute timer prevented overreliance on help. To evaluate all hint types, in this study students saw 4 types of hints: code hint only, code hint with textual explanation, code hint with self-explanation prompts and code hint with both textual explanation and self-explanation prompts. Each time a student requested a hint, they were given a different random hint type, which ensured that each student saw as many different hint types as possible.

Procedure: One researcher conducted the study[2] with each participant individually over a 75 minute period. Prior to arrival, participants filled out a short pre-survey that collected demographic information. The researcher started by asking the student to read through a short tutorial on programming in iSnap for 5-10 minutes. The tutorial covered the user interface of iSnap and explained all programming concepts needed for the later programming tasks (loops, input/output and drawing) using a combination of text and short example animations. Since the goal of our study was to study how students use and learn from help, the tutorial was intentionally short, and students were expected to learn as they programmed. Next, the researcher asked the student to read the instructions of the first programming task and then asked the student to work on Task 1 for 15 minutes. Task 1 asked the student to create a program to draw a polygon with any number of sides (chosen at runtime by the user). After 15 minutes, the student was given the option to take another 5 minutes to complete the task if desired, after which they were asked to stop. Almost 90% of students were able to complete, or nearly complete, this first task. After the student finished Task 1, the researcher conducted a semi-structured interview (Interview 1) about the student's experience with iSnap. This interview lasted 4-6 minutes, during which the researcher asked questions about the specific hints that the student received during Task 1. The researcher showed the student each hint that the student had received, along with the code that the student had written at that point in time. The researcher asked the student about the timing of each hint, what was helpful about it, whether they trusted it, how it can be improved and what motivated them to ask for hints.

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

4.2 Results: Interview Analysis

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

4.2.1 Code hints. **These helped students to see a clear next action they could take, which they could visually compare to their current work.** Students (4 out of 10) declared that code hints were useful because they contrasted students' current code with suggested code: *" it just really helped when evaluating where I*

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

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

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

4.2.2 Code hints with Textual Explanations. **These provide complementary benefits, where the explanation helps students understand the "how" and "why" of a code hint**. Students (7 out of 10) noted that the textual explanation gave useful but different information from a code hint: *"[the code hint] shows which block to use and the text gives an idea of what to use it for."* [P6]. When we asked students whether they prefer code hints alone or code hints with textual explanations, 2 students noted that code hints alone can be enough, and that adding explanations *"can not be not helpful. If you do not need then you do not need."* [P9], and *"I did not find it as helpful as doing it myself."* [P10]. The rest of students preferred having both because *"the text hints helped me to understand the visual hints [code hints] on a deeper level"*[P7]. These results suggest that students appreciate explanations with code hints, and there is little cost to the student, as they are easy enough to ignore.

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

Self-explanation prompts were criticized for being frustrating and confusing. A few students (3 out of 10) had vocally negative comments on adding self-explanation prompts, as confusing: *"I was confused about the question... because I did not know what the hint was giving to me."* [P3] and not giving any support *"it is not giving me anything back, it is just asking me if I understood it."* [P5]. We observed that most of the students who were more critical of the prompts were also not able to complete both programming tasks. This agrees with prior work suggesting that students with lower prior knowledge may have difficulty with open-ended self-explanation prompts, since they lack the domain knowledge to construct meaningful explanations [49].

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

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

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

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

5 STUDY 2: LEARNING AND PERFORMANCE

Having positive insights on hints types from students' interviews, we were encouraged to investigate the effect of hints on performance and learning transfer on a large-scale study. To do so, we recruited crowd workers through Amazon's Mechanical Turk (MTurk) platform who have attested to have little to no prior knowledge in programming. Previous work has shown that recruiting crowd workers can be an effective form of conducting large-scale user studies in lieu of using university participants [11, 27], and this has been previously employed in computing education research to assess the efficacy of online learning approaches [31]. This experiment aims to answer the following research questions: What is the effect of hints with and without self-explanation prompts on: 1) learners' perspectives on iSnap's helpfulness with and without receiving hints? 2) learners' performance?, and 3) learning, as measured by performance on future tasks without hints?

5.1 Methods

Population: In this study, we recruited 250 total crowd workers through Amazon's Mechanical Turk (MTurk) platform. As in Study

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

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

Procedure: This study used a similar procedure to Study 1, though it was conducted online rather than in person. Learners completed the same tutorial and the same 2 programming tasks in the same programming environment, and received the same automated hints. However, since our goal with Study 2 was to measure the impact of hints on performance and learning, there were 4 differences in the procedure. First, each learner was randomly assigned to one of three conditions that determined what type of hint (if any) our system provided in the first task: 1) no hint (control condition), 2) **C**ode hints with **T**extual explanations (*CT*), and 3) **C**ode hints with **T**extual explanations and self-**E**xplanation prompts (*CTE*). Unlike in Study 1, we intentionally chose to always provide textual explanation with code hints, since our results from Study 1, as well as our prior work [32, 44] suggested that textual explanations may improve the likelihood of learners benefiting from hints with few drawbacks. However, students' mixed reactions to self-explanation prompts in Study 1 encouraged us to create a separate condition for hints with explanation prompts (*CTE*). Our population included 63 learners in the no hint condition, 79 in the *CT* condition and 59 in the *CTE* condition[6]. As in Study 1, the programming environment provided learners with the opportunity to request a hint every two minutes, but in Study 2 the type of hints was always the same, dictated by their condition (e.g. a code hint with textual explanations), not random. Our timed approach ensured that, in a 15-min task, every learner can have from 0 to 7 hints maximum.

Second, to measure if hints can improve learners' performance in future tasks, we used Task 2 as a post-test, which did not offer hints to learners in *any* condition. We chose to use Task 2 to assess learning, rather than a traditional post-test, as we were interested in measuring learners' ability to perform a similar task without help. We were also able to use fine-grained analysis of learners' Task 2 programming logs to compare how learners performed across time in different programming objectives. Unlike in Study 1, we had only one version of Task 2, where learners had to draw a strip of triangles (the easier version).

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

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

5.2 Results

5.2.1 The Impact of Hints on Learners' Programming Performance and Learning. To measure the effect of hint condition on learners' immediate performance, we compared the number of objectives that learners completed during Task 1 in the control group (M[7]=2; SD=1.40), *CT* (M=2.53; SD=1.24) and *CTE* (M=2.8; SD=1.31). A Kruskal-Wallis test[8] shows a significant difference among hint conditions for learners' performance ($\chi^2(3) = 12.84, p = 0.001$). Afterwards, we used post-hoc non-parametric Dunn's test with Benjamini-Hochberg correction for multiple comparisons to determine pairwise significant differences across hint conditions [12, 19]. Dunn's test shows a significant difference between control group and *CT* learners ($z = 2.16$; $p = 0.045$), a significant difference between control group and *CTE* learners (z=3.56; $p = 0.001$) and non-significant difference between *CT* and *CTE* learners ($z = 1.62$; $p = 0.104$). This shows that both conditions with code hints completed significantly more objectives than the control condition. As a result, more learners completed all of Task 1 in the *CT* condition (27.8%) and *CTE* condition (45.8%) than the control condition (22.2%). Figure 2 (left) plots the mean number of objectives that had been completed by learners in each condition at different times throughout the 15 minute task. It shows that the difference between the three groups became more pronounced over time. **These results suggest that code hints with textual explanations, and code hints with both textual explanations and self-explanations prompts significantly improve performance.**

Task 2 served as our measure of learning, since learners had no hints on this task. It consists of 4 objectives. The first 2 objectives were identical to the first 2 objectives in Task 1 ("draw something" and "ask user for input X and repeat X times"), and they measured how well learners learned to repeat these steps in a new context.

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

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

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

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

The last 2 objectives measured learners' ability to apply the same programming constructs (loops and drawing) in a new way ("repeatedly draw a triangle" and "draw a strip of triangles"). For example, one solution for the third objective required nested loops.

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

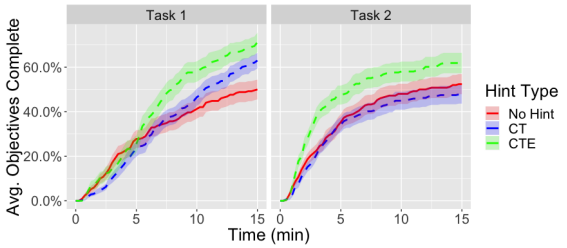

Figure 2: Learners' average completion progress (with shading indicating standard error) in each condition, measured over time, for Task 1 (left) and in Task 2 (right).

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

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

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

6 DISCUSSION AND LIMITATIONS

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

Code hints with textual explanation improve learners' immediate programming performance. In Study 2, learners in the *CT* and *CTE* conditions were able to complete 25% and 40% more of Task 1, respectively, than the control group without hints. Helping students to progress when stuck to complete a problem is one of the primary purposes of next-step hints [3], but it was not obvious that they would accomplish this. We found that students were in fact much more likely to complete all four of Task 1's objectives with hints (over twice as likely in the *CTE* group). Our findings from Study 1 suggest that students are aware of code hints' ability to improve immediate performance, and appreciate the hints' ability to help put them *"in the right direction."* [P2]. This is a similar theme in prior work on students' perceptions of code hints [43, 44].

Self-explanation prompts changed the way that students interacted with code hints. In Study 1, our participants frequently noted that self-explanation prompts caused them to *"... think and take a step back about the whole process."* [P7]. Study 2 helps us to better understand quantitatively what impact the prompts had on learners' use of hints. We found that the median student with hints

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

Code hints only improved learning when accompanied by self-explanation prompts. We found that learners in *CTE* group on Task 1 performed 23% better overall on Task 2 than the control condition. However, learners with only code hints, but not prompts (*CT* group), performed no better than the control condition. Prior work suggested that students learn best from hints when they *spontaneously* self-explain their meaning [3, 51], and our results suggest that this can also be encouraged with prompting them to self-explain. This students' need for self-explanation support may explain why Rivers' prior evaluation of automated hints did not find a learning effect [47]. The students we interviewed in Study 1 seem to be aware of this need, as most (though not all of them) appreciated self-explanation prompts' ability to encourage them *"to interpret what... the picture [hint] mean[s].)"* – or what the KLI framework might call the sense making necessary for learning from hints [28]. We also note that we can make no claim about whether or how textual explanation contributed to learning, since we chose not to investigate this in Study 2.

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

What can we learn from these results? Our results are an important initial step in understanding the potential benefits and limitations of using automated programming hints in classrooms. Systems that offer automated hints *without* self-explanation prompts are currently in use in classrooms (e.g. [22, 41]), and our results suggest that their designers might consider adding this feature. We acknowledge that our studies focused on short (15 minute) programming assignments, and our quantitative results relied on a convenience sample of crowdworkers, so *we should be cautious in generalizing these results* to other contexts. However, we argue that given the lack of existing empirical results on the efficacy of programming hints, these initial, positive results still provide important insight. It is difficult to create large-scale, controlled classroom studies, but our results justify the need for such studies, and for identifying important hypotheses to test in these studies (e.g. the importance of self-explanation prompts). For example, if hints create a learning impact over just 15 minutes of programming, it is possible that the effect may be much larger over a whole semester, but this can only be verified empirically.

6.1 Limitations

In Study 2, our population consisted of paid crowd workers with no prior programming experience. Their motivations, and prior knowledge may differ from those of other populations of learners when programming hints are used, and we emphasize that this limits the generalizability of our results. However, such an approach is not unprecedented in computing research [31], and we argue that we can still gain valuable insight from this population, as suggested by prior work [11, 27]. Working with this population allowed us to collect a large amount of data, randomly assign participants to conditions, and collect a more gender-balanced dataset – all of which can be quite difficult in a classroom setting. In addition, our quantitative results from crowdworkers also strongly parallel our findings from novice students in Study 1

In Study 1, our participants all identified as male. Since prior work suggests that gender plays an important role in how students seek and use help [4, 14], this limits the generalizability of our results. Additionally, in both studies, we only studied users during two simple, 15-minute programming tasks, and we have begun further work to investigate if our results generalize to longer or more complex tasks in classrooms. We argue that this short duration likely made it *more* difficult to detect an effect of hints on learning. However, it likely limited the diversity of help-seeking scenarios that users encountered in Study 1.

7 CONCLUSION

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

REFERENCES

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

[2] Vincent Aleven and Kenneth R. Koedinger. 2001. Investigations into Help seeking and Learning with a Cognitive Tutor. In *Papers of the AIED 2001 Workhop 'Help Provision And Help Seeking In Interactive Learning Environments'*. 47–58.

[3] Vincent Aleven, Ido Roll, Bruce M. McLaren, and Kenneth R. Koedinger. 2016. Help Helps, But Only So Much: Research on Help Seeking with Intelligent Tutoring Systems. *International Journal of Artificial Intelligence in Education* 26, 1 (2016), 1–19. https://doi.org/10.1007/s40593-015-0089-1

[4] Vincent Aleven, Elmar Stahl, Silke Schworm, Frank Fischer, and Raven Wallace. 2003. Help Seeking and Help Design in Interactive Learning Environments Vincent. *Review of Educational Research* 73, 3 (2003), 277–320.

[5] JR Anderson. 1996. ACT: A simple theory of complex cognition. *American Psychologist* (1996). http://psycnet.apa.org/journals/amp/51/4/355/http://courses.csail.mit.edu/6.803/pdf/anderson.pdf

[6] John R. Anderson. 1993. *Rules of the Mind*. Hillsdale, New Jersey.

[7] John R Anderson, Albert T Corbett, Kenneth R Koedinger, and Ray Pelletier. 1995. Cognitive Tutors: Lessons Learned. *The Journal of the Learning Sciences* 4, 2 (1995), 167–207.

[8] Michael Ball. 2018. *Lambda: An Autograder for snap*. Technical Report. Electrical Engineering and Computer Sciences University of California at Berkeley. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-2.pdf

[9] Joseph E. Beck, Kai Min Chang, Jack Mostow, and Albert Corbett. 2008. Does help help? Introducing the Bayesian Evaluation and Assessment Methodology. In *Proceedings of the International Conference on Intelligent Tutoring Systems*. 383–394.

[10] Brett A Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. 2016. Effective compiler error message enhancement for novice programming students. *Computer Science Education* 26, 2-3 (2016), 148–175.

[11] Tara S. Behrend, David J. Sharek, Adam W. Meade, and Eric N. Wiebe. 2011. The viability of crowdsourcing for survey research. *Behavior Research Methods* 43, 3 (25 Mar 2011), 800.

[12] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society* 57, 1 (1995), 289–300.

[13] Jens Bennedsen and Michael E. Caspersen. 2007. Failure rates in introductory programming. *ACM SIGCSE Bulletin* 39, 2 (2007), 32. https://doi.org/10.1145/1272848.1272879

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

[15] Xianglei Chen and Matthew Soldner. 2013. *STEM Attrition: College Students' Paths Into and Out of STEM Fields*. Technical Report. National Center for Education Statistics, Institute of Education Sciences, U.S. Department of Education. http://nces.ed.gov/pubs2014/2014001rev.pdf

[16] Rohan Roy Choudhury, Hezheng Yin, and Armando Fox. 2016. Scale-driven automatic hint generation for coding style. In *Proceedings of the International Conference on Intelligent Tutoring Systems*. 122–132. https://doi.org/10.1007/978-3-319-39583-8_12

[17] Albert Corbett and John R. Anderson. 2001. 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*. 245–252. http://dl.acm.org/citation.cfm?id=365111

[18] Dan Davis, Claudia Hauff, and Geert-Jan Houben. 2018. Evaluating Crowdworkers as a Proxy for Online Learners in Video-Based Learning Contexts. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 42.

[19] Olive Jean Dunn. 1964. Multiple comparisons using rank sums. *Technometrics* 6, 3 (1964), 241–252.

[20] Anna Espasa and Julio Meneses. 2010. Analysing feedback processes in an online teaching and learning environment: an exploratory study. *Higher education* 59, 3 (2010), 277–292.

[21] Davide Fossati, Barbara Di Eugenio, Stellan Ohlsson, Christopher Brown, and Lin Chen. 2015. Data Driven Automatic Feedback Generation in the iList Intelligent Tutoring System. *Technology, Instruction, Cognition and Learning* 10, 1 (2015), 5–26.

[22] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L. Thomas van Binsbergen. 2016. Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback. *International Journal of Artificial Intelligence in Education* 27, 1 (2016), 1–36.

[23] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common Programming Errors by Deep Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 1. 1345–1351.

[24] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, and Jeremy Ernst. 2018. Misconception-driven feedback: Results from an experimental study. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, 160–168.

[25] Björn Hartmann, Daniel Macdougall, Joel Brandt, and Scott R Klemmer. 2010. What Would Other Programmers Do? Suggesting Solutions to Error Messages. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 1019–1028. https://doi.org/10.1145/1753326.1753478

[26] Andrew Hicks, Barry Peddycord III, and Tiffany Barnes. 2014. Building Games to Learn from Their Players: Generating Hints in a Serious Game. In *Proceedings of the International Conference on Intelligent Tutoring Systems*. 312–317.

[27] Aniket Kittur, Ed H Chi, and Bongwon Suh. 2008. Crowdsourcing user studies with Mechanical Turk. In *Proceedings of the SIGCHI conference on human factors in computing systems*. ACM, 453–456.

[28] KR Koedinger and JC Stamper. 2013. Using data-driven discovery of better student models to improve student learning. In *Proceedings of the International Conference on Artificial Intelligence in Education*.

[29] Timotej Lazar and Ivan Bratko. 2014. Data-Driven Program Synthesis for Hint Generation in Programming Tutors. In *Proceedings of the International Conference on Intelligent Tutoring Systems*. Springer, 306–311.

[30] Timotej Lazar, Martin Možina, and Ivan Bratko. 2017. Automatic Extraction of AST Patterns for Debugging Student Programs. In *Proceedings of the International Conference on Artificial Intelligence in Education*. 162–174.

[31] Michael J Lee and Andrew J Ko. 2015. Comparing the effectiveness of online learning approaches on CS1 learning outcomes. In *Proceedings of the eleventh annual international conference on international computing education research*. ACM, 237–246.

[32] S. Marwan, N. Lytle, J. J. Williams, and T. W. Price. 2019. The Impact of Adding Textual Explanations to Next-step Hints in a Novice Programming Environment. In *Proceedings of the 24th Annual ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE19 (forthcoming)*.

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

[34] Antonija Mitrovic, Pramuditha Suraweera, Brent Martin, and Amali Weerasinghe. 2004. DB-suite: Experiences with three intelligent, web-based database tutors. *Journal of Interactive Learning Research* 15, 4 (2004), 409–432.

[35] Briana B Morrison, Lauren E Margulieux, and Cherry Street. 2015. Subgoals , Context , and Worked Examples in Learning Computing Problem Solving. In *Proceedings of the International Computing Education Research Conference*. 21–29. https://doi.org/10.1145/2787622.2787733

[36] Pete Nordquist. 2007. Providing accurate and timely feedback by automatically grading student programming labs. *Journal of Computing Sciences in Colleges* 23, 2 (2007), 16–23.

[37] Benjamin Paaßen, Barbara Hammer, Thomas W. Price, Tiffany Barnes, Sebastian Gross, and Niels Pinkwart. 2018. The Continuous Hint Factory -Providing Hints in Vast and Sparsely Populated Edit Distance Spaces. *Journal of Educational Data Mining* (2018), 1–50.

[38] Daniel Perelman, Sumit Gulwani, and Dan Grossman. 2014. Test-Driven Synthesis for Automated Feedback for Introductory Computer Science Assignments. In *Proceedings of the Workshop on Data Mining for Educational Assessment and Feedback*.

[39] Chris Piech, Mehran Sahami, Joh Huang, and Leo Guibas. 2015. Autonomously Generating Hints by Inferring Problem Solving Policies. In *Proceedings of the ACM Conference on Learning @ Scale*. 1–10.

[40] T.W. Price, R. Zhi, Y. Dong, N. Lytle, and T. Barnes. 2018. 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*. https://doi.org/10.1007/978-3-319-93843-1_35

[41] Thomas W. Price, Yihuan Dong, and Dragan Lipovac. 2017. iSnap: Towards Intelligent Tutoring in Novice Programming Environments. In *Proceedings of the ACM Technical Symposium on Computer Science Education*.

[42] Thomas W Price, Yihuan Dong, Rui Zhi, Benjamin Paaßen, Nicholas Lytle, Veronica Cateté, and Tiffany Barnes. 2019. A Comparison of the Quality of Data-driven Programming Hint Generation Algorithms. *International Journal of Artificial Intelligence in Education* (2019).

[43] Thomas W. Price, Zhongxiu Liu, Veronica Catete, and Tiffany Barnes. 2017. Factors Influencing Students' Help-Seeking Behavior while Programming with Human and Computer Tutors. In *Proceedings of the International Computing Education Research Conference*.

[44] T. W. Price, J. J. Williams, and S. Marwan. 2019. A Comparison of Two Designs for Automated Programming Hints. (2019).

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

[46] Thomas W. Price, Rui Zhi, and Tiffany Barnes. 2017. Hint Generation Under Uncertainty: The Effect of Hint Quality on Help-Seeking Behavior. In *Proceedings of the International Conference on Artificial Intelligence in Education*.

[47] Kelly Rivers. 2016. *Automated Data-Driven Hint Generation for Learning Programming*. Ph.D. Dissertation. Carnegie Mellon University.

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

[49] Marguerite Roy and Michelene TH Chi. 2005. The self-explanation principle in multimedia learning. *The Cambridge handbook of multimedia learning* (2005), 271–286.

[50] Silke Schworm and Alexander Renkl. 2006. Computer-supported Example-based Learning: When Instructional Explanations Reduce Self-explanations. *Computers & Education* 46, 4 (2006), 426–445.

[51] Benjamin Shih, Kenneth Koedinger, and Richard Scheines. 2008. A Response Time Model for Bottom-Out Hints as Worked Examples. In *Proceedings of the International Conference on Educational Data Mining*. 117 – 126.

[52] Hyungyu Shin, Eun-Young Ko, Joseph Jay Williams, and Juho Kim. 2018. Understanding the Effect of In-Video Prompting on Learners and Instructors. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 319, 12 pages.

[53] Arto Vihavainen, Craig S. Miller, and Amber Settle. 2015. Benefits of Self-explanation in Introductory Programming. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education - SIGCSE '15* 68 (2015), 284–289.

[54] Ke Wang, Benjamin Lin, Bjorn Rettig, Paul Pardi, and Rishabh Singh. 2017. Data-Driven Feedback Generator for Online Programing Courses. In *Proceedings of the ACM Conference on Learning @ Scale*. 257–260.

[55] Christopher Watson and Frederick W B Li. 2014. Failure rates in introductory programming revisited. In *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 39–44.

[56] Christopher Watson, Frederick W B Li, and Jamie L. Godwin. 2012. BlueFix: Using crowd-sourced feedback to support programming students in error diagnosis and repair. In *Proceedings of the International Conference on Web-based Learning*. 228–239.

[57] Joseph Jay Williams, Tania Lombrozo, Anne Hsu, Bernd Huber, and Juho Kim. 2016. Revising Learner Misconceptions Without Feedback: Prompting for Reflection on Anomalies. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 470–474.

[58] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 740–751. http://dl.acm.org/citation.cfm?doid=3106237.3106262