Step Tutor: Supporting Students through Step-by-Step Example-Based Feedback

Wengran Wang
North Carolina State University
Raleigh, NC
wwang33@ncsu.edu

ABSTRACT

Students often get stuck when programming independently, and need help to progress. Existing, automated feedback can help students progress, but it is unclear whether it ultimately leads to learning. We present Step Tutor, which helps struggling students during programming by presenting them with relevant, step-by-step examples. The goal of Step Tutor is to help students progress, and engage them in comparison, reflection, and learning. When a student requests help, Step Tutor adaptively selects an example to demonstrate the next meaningful step in the solution. It engages the student in comparing "before" and "after" code snapshots, and their corresponding visual output, and guides them to reflect on the changes. Step Tutor is a novel form of help that combines effective aspects of existing support features, such as hints and Worked Examples, to help students both progress and learn. To understand how students use Step Tutor, we asked nine undergraduate students to complete two programming tasks, with its help, and interviewed them about their experience. We present our qualitative analysis of students' experience, which shows us why and how they seek help from Step Tutor, and Step Tutor's affordances. These initial results suggest that students perceived that Step Tutor accomplished its goals of helping them to progress and learn.

ACM Reference Format:
Wengran Wang. 2020. Step Tutor: Supporting Students through Step-by-Step Example-Based Feedback. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '20), June 15–19, 2020, Trondheim, Norway.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3341525.3387411

1 INTRODUCTION

Students get stuck in many ways while programming [31], leading to frustration [25]. Ideally, a student can ask for instructor help, but this may be difficult in today's growing CS classrooms [6], where instructor availability is limited. And the student may see asking for instructor help as a threat to their competence and independence [11]. To solve this problem, researchers have developed various kinds of automated, adaptive programming feedback to help students [16, 17, 24, 38]. Like instructor feedback, adaptive feedback is context-dependent and personalized to address students' current code. Unlike instructor feedback, adaptive feedback is also scalable: it can be generated automatically using data-driven approaches (e.g., [38, 43]). This feedback can reduce teacher workload, and does not pose a social threat to students.

Adaptive feedback takes many forms, such as highlighting possibly erroneous code [17, 38], reporting completed and uncompleted objectives [52], and correcting likely misconceptions [24]. Perhaps the most common form of adaptive feedback is on-demand, next-step hints, which suggest a small change a student can make to progress or fix an error [16, 39, 43]. These hints can help students progress when stuck [39, 40], and sometimes achieve better learning [16], especially when students engage in self-explanation of the hint [35].

However, next-step hints also have limitations: they present a single edit, with little context. The hints may be difficult to interpret or fail to address students' immediate goals, leading students to ignore or avoid them [40, 41]. They do not always lead to learning [43], and students can abuse the help to complete the problem without effort [5, 32]. This suggests a need to design improved forms of adaptive feedback, that offer context, reflection and learning.

In this paper, we present Step Tutor. It is an extension of the Snap*!* block-based programming environment, which adds on-demand, example-based feedback. The goal of Step Tutor is 1) to help struggling students progress by demonstrating a correct solution step, and 2) to help them learn why the step works and how to apply it in the future. It does so by demonstrating an *example step* from the current problem, allowing the student to compare and run code representing before and after the step is completed, and reflect on the difference. Step Tutor leverages prior literature on Worked Examples, which are an effective and efficient way to learn [48]. Since Worked Examples are not designed to be used *in the middle of* problem-solving to help students progress, Step Tutor extends traditional Worked Examples by offering them as step-by-step example programs *during* programming. It shows examples step-by-step, to support students *during* problem-solving. By combining the immediate usefulness of next-step hints and the learning effectiveness of Worked Examples, Step Tutor aims to help struggling students progress, and guide them to explore and reflect on its suggested changes, so they can learn to apply it in the future.

To understand how students use the different features of Step Tutor, we asked 9 undergraduate students from introductory programming classes to use it to solve two programming problems. We conducted interviews with these students after each problem, and grouped students' responses using thematic analysis. Our results revealed a variety of student motivations for using Step Tutor (e.g. progress, assurance, independence), and strategies for using it

ITiCSE '20, June 15–19, 2020, Trondheim, Norway

ACM ISBN 978-1-4503-6874-2/20/06…$15.00
https://doi.org/10.1145/3341525.3387411

(e.g. running, comparing, copying examples), suggesting that the examples are a flexible form of support. We also found that students identified many affordances of the Step Tutor, including the comprehensiveness of its individual examples, and the "roadmap" outlined by a series of examples, which students noted as advantages over next-step hints that could promote learning.

2 RELATED WORK

Next-step hint and its limitations in promoting learning:

Hints have been traditionally valued in education theory as a tutorial tactic, which provides students with the information needed to progress or prompts them to reflect on their knowledge and problem-solving status [26]. Adaptive help systems commonly use on-demand, next-step hints that suggest the next step a student should take [4, 16, 39, 43]. In the domain of programming, next-step hints can help students program more efficiently. For example, undergraduate students spent significantly less time completing Lisp programming tasks, with three different forms of next-step hints, compared to those without [16]. Additionally, in block-based Snap*!* programming, next-step hints with explanations or self-explanation prompts allowed crowd-workers to complete more programming objectives in the same amount of time, than those without hints [35].

However, while next-step hints have been widely shown to enable progression [16, 43], they don't always lead to improved learning [43, 53]. For example, Rivers et al. evaluated 15 undergraduate students' programming experience using next-step hints in the ITAP python tutor, and found learning gains to be the same with and without next-step hints, although learners with hints spent less time to complete the task [43]. This missed learning can occur because hints fail to address students' needs [40, 41], for hints show only one edit at a time, and may lack context and details needed to interpret the suggestion [36]. Additionally, students often misuse hints (e.g., hint abuse) [4], allowing them to progress without understanding each step. Aleven et al. reviewed hints from various domains, and concluded that learning from hints only happens occasionally, under certain circumstances, and its effect is small [4].

How do students *learn* programming strategies? Schön explained the cognitive process of learning procedural knowledge [4] through reflection-in-action, emphasizing that new knowledge is gained through *self-reflection*, during which the learner repeatedly questions herself while *actively* working and testing on the learning material [45]. This emphasis on self-reflection and activity is echoed by several learning theories, such as the sense-making process highlighted by the KLI framework [2]), and the emphasis on active interaction with the learning material suggested by the ICAP framework [12]. For a next-step hint to offer students meaningful content to promote sense-making and self-reflection, its "next step" may involve more than one single edit. And for the next-step hint to enable an active learning experience, its feedback window should go beyond just allowing students to passively view the content. These indicate ways to improve upon next-step hints, to offer feedback that gives a clear step, and enables students to interact with the feedback and reflect on why and how the step works.

Worked Examples:

Worked Examples (WEs) are a form of instructional support, which give students a demonstration of how to solve the problem [13]. Unlike next-step hints, WEs are traditionally offered in lieu of problem solving, usually "before" or "after" a student solves a distinct but related programming task [10, 50]. The effectiveness of WEs is primarily grounded in Cognitive Load Theory, which argues that learners have a finite amount of mental resources during problem-solving (called cognitive load), and when problems impose an unnecessary burden on those resources (intrinsic load), the student has fewer resources left for processing and learning the material (germane load) [47]. WEs support learning by providing support for "borrowing" knowledge, reducing the unnecessary intrinsic load [48]. Programming learning environments use WEs widely. For example, WebEx provides web-based self-explaining code examples for students [10]. Such programming WEs could help students learn the problem-solving schema [19] and transfer it to another task [50]. Trafton et al. evaluated 40 undergraduate students' post-test scores after programming in BATBook, a Lisp programming learning environment, and found that those with alternating WE and problem solving (PS) pairs performed better than those with PS pairs [50]. However, another group, who saw all WE problems, followed by all PS problems (not in pairs), solved problems significantly slower, and achieved significantly lower post-test scores. This study shows that students seek for *immediate relevance* when studying an example. And even being provided with the most relevant WE right before solving a problem, students still need help when they are stuck in the middle of problem-solving.

Worked Examples during problem-solving:

WEs are an effective learning support, but students still need help *during* programming when they are stuck. Looking Glass provides students with annotated WEs from *another* similar task during block-based programming [27, 28]. However, learners had difficulties understanding these examples in Looking Glass, encountering "example comprehension hurdles" while trying to connect example code to their own code [27, 28].

Novice students are usually not able to spontaneously transfer knowledge they learn from one problem to another isomorphic problem [21], so they can benefit more if WEs are offered from the *same* programming problem. Peer Code Helper offers such step-by-step WEs from the same task, during block-based programming [56]. An evaluation on 22 high school novice students showed that students using these WEs solved tasks quicker than those without, without hindering their learning [56]. The FIT Java Tutor [23] provides such step-by-step WEs for Java programming. Investigation on five students' programming experience showed that students occasionally followed the feedback and improved their program over time [22]. But, example steps in these programming problems are non-adaptive [56], or coarse-grained [23, 56], and lack the necessary scaffolding for students to make sense of them. In a study evaluating 23 students' experience with step-by-step WEs offered during Java programming, students barely followed the examples, reporting them being "unspecific and misleading" [14]. Therefore, more work is needed to design new forms of example feedback, to promote reflection through fine-grained example steps, and to enable progression through an interactive experience [15].

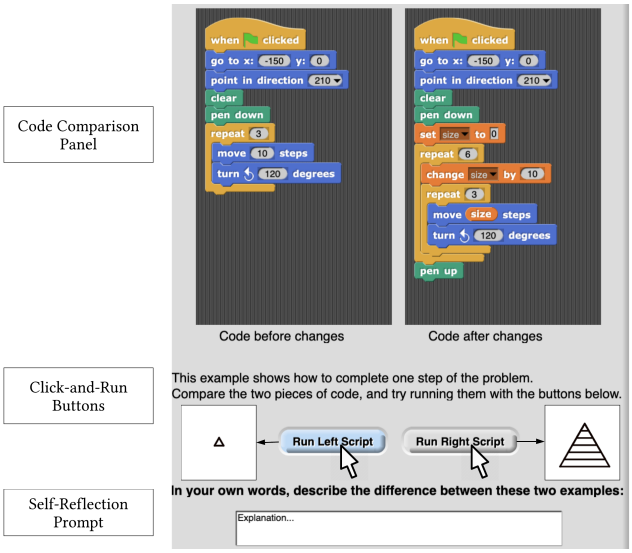

Figure 1: An example step given by Step Tutor, which includes a Code Comparison Panel, two Click-and-Run Buttons, and a Self-Explanation Prompt.

3 STEP TUTOR

Step Tutor is an extension of the Snap*!* programming environment. Its goal is to help students progress and learn when stuck, by teaching them a meaningful step when requested. Step Tutor helps students by showing them a concrete example of how the step could be completed, including both the changes in the code and the corresponding changes in the program's output, and prompting the student to reflect on these differences. In our context, an "example step" should be self-contained, and large enough to meaningfully change the program's output, but small enough to be easily digested. This feedback serves the dual purpose of helping the student: 1) progress when stuck (as with a hint), and also 2) critically engage with and reflect on the example code to learn generalizable programming concepts. We designed Step Tutor to achieve this goal through a feedback window that facilitates comparison, code running, and self-explanation.

Because Step Tutor extends Snap*!* , instructors can easily integrate it into widely-used Snap*!* programming curricula (e.g., the Beauty and Joy of Computing [18]). Although Step Tutor currently only supports Snap*!*-based programming, the design and algorithm we used to create Step Tutor and its feedback is language-agnostic.

3.1 The Step Tutor Feedback Window

Consider a student who gets stuck when working on a homework problem in Snap*!* , perhaps due to a bug in her code, a misconception, or uncertainty about how to proceed. Rather than waiting for office hours or a response on a forum, the student can click on the "Show Example" button that's displayed on the programming interface to ask for help. Step Tutor added this "Show Example" button to the original Snap*!* interface to remind students about this added option to view a step. It flashes every 90 seconds, inspired by prior work [35], which suggests that students can become too engaged in solving a challenging problem to notice or act on their own need for help [40]. When the student clicks on the button, she sees the Step Tutor feedback window (Figure 1), which shows a carefully-selected example step (explained in Section 3.2). The feedback window guides the student through learning the step in three ways, designed to promote deliberate comparison and self-reflection, to help students learn the step, and learn how to apply it again in the future [46]:

Comparing and running the code: At the top of the feedback window, the student sees two code snapshots, which together give a meaningful, interpretable step that a student can take to proceed towards the solution, selected by the example selection algorithm in Section 3.2. The left "before" code is similar to the student's code, representing "before" the step is completed, and the right "end" code shows the changes needed to complete the step. The student can inspect the example step by comparing the left and right code. We want to encourage learning the step through comparison, because prior work in programming education suggests that comparison is a powerful way for a student to learn from examples and generalize domain principles [19, 37, 42]. The student is also encouraged to run the code to understand the step. The click-and-run feature prompts students to actively engage with and reflect on the example code, which is an essential element of active learning defined by the ICAP framework [12].

Writing self-explanation: After the student has viewed an example step, she can answer the self-explanation prompt: "In your own words, describe the difference between the two examples". Although answering the prompt takes time, prior work suggests that self-explanation is critical for learning from feedback [7, 36, 46, 49], and such self-explanation does not emerge spontaneously without carefully-designed prompts [4, 20]. After writing self-explanation, she can either close the Step Tutor feedback window, or leave it open as a reference when she continues to write code.

When the student continues to make progress, but gets stuck again, she can use the "Show Example" button to ask for another help, seeing a new example step adapted to her current code. Instructors could easily add a limit of total request to prevent help abuse [4]. To explore a student's natural use of the system, we did not impose the limit in our study. Additionally, even if a student does abuse help by repeatedly asking for examples, she has still experienced a step-by-step Worked Example, which prior work shows leads to more efficient learning outcomes than solving the problem from scratch [50].

3.2 The Example Selection Algorithm

An important feature of Step Tutor is that it adaptively selects examples, tailoring them to a student's current code. It does so through an *example selection* algorithm, which searches for an example step. To help students achieve optimal learning, the example step should be one that the student has not completed but is ready to complete with some help – meaning one in the student's Zone of Proximal Development [51]. Our approach extends our prior work [54], and consists of the following steps:

An instructor creates a database of example steps: An instructor can first create a set of example steps for a problem, each consisting of "before" and "after" code (as shown in Figure 1), representing

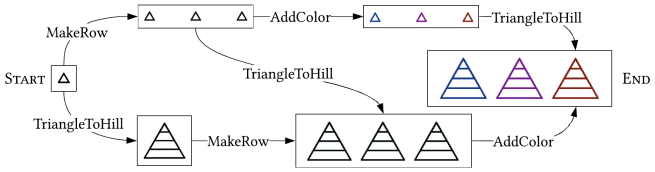

Figure 2: Three possible solution paths for Task 2, made up of three example steps: (a) MakeRow: drawing a row of triangles; (b) TriangleToHill: changing the triangle to a stripped hill; (c) AddColor: adding color for each triangle.

one meaningful step in completing the solution which ideally alters the program's output. In many problems, solution steps can be completed in various orders [55]. As is shown in Figure 2, there are multiple ways to reach the solution. To illustrate each possible path to the solution, it is helpful to author an example pair (i.e., the "before" code and the "after" code) for each transition (i.e., each arrow in Figure 2).

Example pairs are easy for instructors to author, consisting of two code snapshots: the "before" code and the "after" code. The example pairs can then be loaded into a database, adding labels for its location in different solution paths, and then Step Tutor does the rest, as discussed below. Our prior work has demonstrated that we can even automatically generate these example steps by extracting solution features from previous students' submissions, and that they match the quality of expert examples [54]. However, in this work, we used hand-authored examples, since we did not have access to prior successful student submissions for our programming tasks.

The algorithm selects "before" and "after" code snapshots: When a student requests an example step, the selection algorithm attempts to select one with "before" code very similar to the student's, so it is easy to understand. The algorithm transfers students' code to an abstract syntax tree (AST), and then calculates the distance between the student's code AST and the "before" code AST of each of the expert examples using the SourceCheck code distance function [38]. Among the two examples that have the smallest distance to the student's current code, we select the example with the fewest completed steps, so err on the side of giving away less of the solution. The algorithm then attempts to select "after" code that demonstrates a step the student can understand. Since all selected example pairs will complete at least one step beyond the student's code, the algorithm selects the example step with "after" code closest to the students' current code, which is most likely to be the step the student is currently working on.

4 USER STUDY

Before deploying Step Tutor in a classroom, we wanted to collect formative data on how and why students used Step Tutor, to gain a better understanding of the strengths and weaknesses of the system, and whether it is likely to achieve its goals of supporting progress and learning. So, we recruited a small group of students to use Step Tutor while solving two programming problems, and conducted interviews to better understand their experiences.

Participants: We recruited nine undergraduate students from two introductory programming courses in a large, public university in the Southeast United States. In both classes, students had been taught Snap*!* programming for one to two months. The students included five males and four females, with two identifying as Hispanic/Latino, three as Asian, three as White, and one undisclosed. Each student received a $25 gift card as compensation at the end of the study.

Procedure: Students completed the procedures one at a time. To start with, they read a short tutorial on Snap*!* to refresh their memories. They were then given up to 30 minutes to complete each of two programming tasks: Stairway and Row of Hills (the latter shown in Figure 2). Both tasks required students to use variables, loops, and nested loops. These tasks were appropriate for our study, because they contain decomposable steps, visual output, and concepts that were beyond what students learned in their coursework (e.g., three nested loops). The researcher did not offer help to students except to confirm when the programming task was completed. After completing each programming task, the researcher conducted a semi-structured interview to ask about the student's experience in the task. Each of the two interview sessions lasted between five to 15 minutes.

Each interview includes a retrospective think-aloud protocol [34], during which we asked students to watch the video we recorded when they interacted with Step Tutor. While watching the video, they were asked to explain their thoughts and activities. The interviewer also asked students' general experience with Step Tutor. Since one of the goals of Step Tutor was to provide more helpful feedback than next-step hints, and students had had access to next-step hints [39] in their classrooms, we also included questions asking students to compare Step Tutor to next-step hints. To encourage impartial feedback, they were not informed that Step Tutor was designed by our research team, and the questions were designed to evoke open-ended responses (e.g., "Here's an example you have requested; how do you feel about it?").

5 ANALYSIS & RESULTS

Rather than a summative evaluation, we conducted a formative evaluation to understand students' experience with Step Tutor. As a pilot study, we used qualitative analysis to capture the various ways students interacted with Step Tutor. For this initial analysis, We focused on analyzing the interview data, because it offers a comprehensive understanding of students' experience.

5.1 Thematic Analysis

We used thematic analysis to summarize and identify central themes from our interviews [9]. Using the six-phase thematic analysis method outlined by Braun and Clarke [9], two researchers each independently read the transcripts thoroughly (Phase 1), open-coded conversation sentences with labels of interest, then met to discuss and refine the codes, producing 125 initial codes (Phase 2). The two researchers then iteratively analyzed and categorized codes to generate themes, i.e., general ideas that emerged in codes (Phase 3). They then revisited the original data to refine the initial themes into main themes, each including several sub-themes (Phase 4). The two researchers then discussed and defined the themes (Phase 5). We present the results (Phase 6) in Section 5.2.

Themes	H (3)	M (2)	L (4)	Total
1. WHY I chose to use or not use Step Tutor				
1.1. High-level goals				
1.1.1. Progress	3	2	4	9
1.1.2. Assurance	1	0	0	1
1.1.3. Expedience	1	0	0	1
1.1.4. Independence	1	2	4	7
1.2. Low-level goals				
1.2.1. Find next step	3	0	0	3
1.2.2. Find how to do a step	1	2	1	4
1.2.3. Fix a problem in my code	1	0	4	5
2. HOW I used Step Tutor				
2.1. Run example code	3	1	3	7
2.2. Comparing example code	3	2	2	7
2.3. Write self-explanation	2	1	2	5
2.4. Locate the change	0	0	4	4
2.5. Copy example code	1	0	0	1
3. WHAT affordances Step Tutor offers				
3.1. Comparison with hints	3	2	3	8
3.2. Roadmapping				
3.2.1. Connect the roadmap	2	0	0	2
3.2.2. Roadmap transfer	1	0	0	1

Table 1: Themes discussed by students from high-use (H), medium-use (M), and low-use (L) groups.

5.2 Findings

Our thematic analysis has revealed three main themes: why I chose to use or not use Step Tutor (WHY); how I used Step Tutor (HOW); what affordances Step Tutor offers (WHAT) (Table 1). During the Phase 4 of our thematic analysis, when we refined initial themes into these three main themes and their sub-themes, we revisited the data and found a meaningful correspondence between the WHY, HOW, WHAT, and the frequency students asked for examples. This correspondence revealed three groups of students, with different levels of reliance on examples: **high-use group** (**H** - P1, P3, P6; 2+ per task) [1]; **medium-use group** (**M** - P4, P8; 0-2 per task), and **low-use group** (**L** - P2, P5, P7, P9; 0-1 per task). Rather than providing a definitive categorization of student behaviors, we used these groups to draw descriptive connections between the WHY, HOW, and WHAT of each individual student. Table 1 provides the count of students in each group who discussed each theme.

5.2.1 WHY I chose to use or not use Step Tutor? Understanding users' needs is essential in user-centered design [1]. Students' discussions of "WHY" can tell us how we may adjust our design goal to align with students' expectations. The theme "why students choose to use or not use the Step Tutor" discusses the motivations for students to ask or not ask for help. Students talked about this theme when they were discussing reasons why they asked certain help, such as "*I looked at the example because ...*"P2/L, or "*my main goal was to ...*"P1/H, or answering the question: "why did you click the "show example" button?". We grouped the sub-themes into *high-level goals*, such as affective and achievement goals, that are not problem-specific, and *low-level goals* that describe specific outcomes students want in relation to their current code, including find next step, find blocks I need to achieve my goal, and fix a problem in my code.

High-level goals. A primary high-level motivation expressed by all students was the desire to **progress** in the assignment. When asked why they asked for an example, one student stated "*I don't wanna like, keep getting stuck on this one little piece*"[P1/H] [2]. This aligns with one of the primary goals of Step Tutor: to help students progress. We also saw other motivations that we had not anticipated. For example, one student noted her desire for **assurance**: "*students like me who are not like pretty confident in programming, having an example makes us feel ... like, this is how you need to do it.*"[P3/H]. Step Tutor may address this need for assurance with the "before" code snapshot, which allows them to confirm the correctness of what they have already written. Another student's motivation for using Step Tutor was an **expedience** goal [30], to reduce their own effort: "*coz it's easy I guess, it just shows you what to do?*"[P6/H]. These two goals were unique among the high-use group, who may have lower self-efficacy [8], suggesting that Step Tutor needs to address their particular needs for reassurance, and discourage expedient help use. Students also expressed motivations for *not* using Step Tutor, such as a desire for **independence.** As in prior work [40], students avoided using help to maintain independence: "*I figure things out on my own, so I learn more thoroughly.*"[P8/M]. But unlike prior work, another student noted that Step Tutor actually gave her a sense of *having* independence, because it allows students to continue working without the need for other forms of help: "*even though [Step Tutor is] helping, they are doing it independently, they are doing it by themselves, instead of calling [teaching assistant] every other time.*"[P3/H].

Low-level goals. We discovered three low-level goals, which includes 1) needs to **find the next step**, because they "*still don't have a clue*"[P1/H] how to progress; 2) needs to **find how to achieve their intended step**: "*I was just trying to figure out how to draw one triangle.*"[P9/L]; and 3) needs to **fix a problem in their code**: "*my triangle was drawing it upside down, and I wasn't sure what was going on.*"[P2/H]. These low-level goals are consistent with the learning barriers discovered by Ko et al. [31]. For example, 1) the "find the next step" goal aligns with the *design* barrier - "I don't know what I want my computer to do"; 2) the goal to "find how to achieve their intended step" aligns with the *selection*, *coordination*, and *use* barriers, which describes barriers to select what programming interfaces to use, to coordinate a set of interfaces, and to use the interfaces; 3) the goal to "fix a problem in their code" aligns with the *understanding* and *information* barriers, describing a student not knowing why the program goes wrong, or not knowing how to check. We found that while high-use groups had all these three low-level goals, medium- and low- use groups' low-level goals were more focused on finding how to achieve their intended step or to fix a problem in their code. We later also found interesting alignment between different groups' low-level goals and their interactions with Step Tutor, discussed in Section 5.2.2.

[1] high-use group includes participants P1, P3 and P6. Each of them were observed to have requested at least two example steps per task.

[2] A quotation from participant P1 within the high-use (H) group.

5.2.2 HOW I used Step Tutor? We collected the "HOW" theme through the retrospective think-aloud protocols during the interviews. We found various ways students interacted with Step Tutor, including those we intended (run, compare, self-explain), and some we did not:

Run, compare, and self-explain. Students discussed interleaved activities of running, comparing and self-explain the examples, which aligned with our design goals. Students found running the examples to be useful, since it shows "*the difference between these two codes*"[P1/H], and how it leads to "*the difference in the outputs.*"[P1/H]. Then, seeing the output triggers them to think more: "*the output of the examples… make me wonder for a sec like, why it gives me example [sic] like that.*"[P1/H]. Students also noted engaging in comparison, not only by running the left and right example snapshot to "*see what's going on*"[P4/M], but also by comparing their own code with the left or right code snapshot to see if their code "*matches up*"[P4/M] with the example code. We found students had mixed feelings about writing self-explanation, as in prior work [35]. Some expressed that self-explanation was distracting: "*when I can look at what's going on, and understand, [writing self-explanation] kind of gets in the way.*"[P5/L]. In contrast, other students expressed that it helped them reflect and think more: "*if you write it down as a reflection, … it would just get into your head that there were these differences and that's what I have to do next.*"[P9/L]. One student also appreciated the chance for expression, explaining that the self-explanation prompt "*gives a place where I can explain what I'm feeling*"[P3/H]. The interleaved activities of running, comparing, and reflection show that our tool offers an active and engaged learning experience, and suggest us to design the self-explanation prompts carefully, to help students without frustrating them.

Locate the change and copy example code. We noticed the low-use group generally did not run the example code "*because I felt like I already knew what [the snapshots] are going to do.*"[P5/L]. Instead, they used the example code to locate where they needed to update their code. For example, a student who asked for an example to find how to use a "turn" block explained: "*I look for whichever one that already have [sic] a turn degrees.*"[P5/L]. These students also expressed their low-level goals as only using the example steps when they knew *what* to do next, but were unclear *how*, or needed to fix a problem with their code. We also observed another behavior: copying the example code, which all three high-use group students employed (based on our observations). These students also expressed all three different low-level goals, including to "find the next step", indicating that they were unclear about what they wanted the program to do [31]. While two students seemed to have interleaved the code-copying experience with running, comparing, and critically reflecting on the examples, one expressed "*when I read the example, I was just copying it.*"[P6/H], and critically commented that it made him "*think a lot less*"[P6/H]. While this student seemed to have abused Step Tutor's help, suggesting a limitation of our tool, we discuss the positive impact of example-copying for the other two students below.

5.2.3 WHAT affordances Step Tutor offers? Other than the specific interactions students experienced with Step Tutor, they have also expressed perceptions of Step Tutor as a whole. They discussed this theme 1) when they were prompted to compare the next-step hints in Snap*!*, that they received in their course projects, and 2) when they were prompted to comment on the usabilities of Step Tutor, such as "How was the example useful to you, or not useful to you?", and also 3) during the retrospective think-aloud protocols.

Comparison with hints. During the interviews, among eight of our interviewees who recalled using next-step hints in class, five preferred Step Tutor, one hints, and two didn't express a strong preference. Students commented that a "*hint is like, no information*"[P6/H], and they "*don't really do anything*"[P6/H] because they tell "*something I already know*"[P1/H]. In contrast, they appreciated the interpretability of Step Tutor - comparing to next-step hints, an example step "*gives the entire coding [sic]*"[P3/H], and tells "*how to combine the things I already know*"[P1/H]. They also appreciated the demonstration offered by Step Tutor, since it "*teaches better*"[P6/H] by "*[showing] how the stuff runs*"[P6/H]. But regarding the amount of information that's given, three students believed that next-step hints have advantages over Step Tutor. They discussed that a hint "*doesn't give me as much of the answer as the example does*"[P5/L], so it "*makes you think more on your own*"[P5/L], indicating that students need more control over how much information they see.

Roadmapping. Unlike experience with one example step, the "roadmapping" theme describes how the series of examples steps together help students understand the high-level structure of the solution: "*I can see how the example [given] is... evolving, from one single square to all these squares and then increase the thickness.*"[P1/H]. A student in the high-use group expressed that the series of examples in one task (e.g., the Step Tutor tutorial) helped them learn the task structure: "*in the example before, it was drawing one circle first, and then... it was drawing many circles...*"[P3/H]. She then was able to apply this pattern to a subsequent task which used a similar solution structure: "*I was expecting the same thing …*"[P3/H], showing an effort to transfer knowledge from one task to another.

6 DISCUSSION

We here compare students' experience with Step Tutor with help-seeking and learning behaviors highlighted by prior work, and examine whether Step Tutor combines the benefits of promoting progression offered by next-step hints, and the benefits of enabling learning and transfer provided by Worked Examples.

Can Step Tutor help students progress?

Eight out of nine students successfully completed the tasks within 30 minutes in our study. Like findings in students' help-seeking behaviors with *next-step hints* [2, 40], students actively employed help-seeking as a problem-solving strategy to progress when stuck [40], indicating Step Tutor may offer similar benefits to next-step hints. Unlike challenges observed in students' experience with *Worked Examples*, such as difficulties in understanding examples [27], we found many students followed the steps suggested by Step Tutor, which shows Step Tutor offers clear information for students to trust the step [40], and use it to progress.

Can Step Tutor help students learn?

One concern instructors may have is that of "Assistance Dilemma" [32] - Step Tutor may give away too much information, allowing students to progress, but without understanding how, as in expedient help-seeking or help-abuse, problems commonly seen among students' interactions with next-step hints [5, 30, 44]. However,

unlike those next-step hints that were perceived as not addressing their needs [40, 41], students described Step Tutor to offer clear and interpretable information. We also saw that most students, including two of the three students who copied examples, engaged in running, comparing, and self-explaining, showing a deliberate attempt to make sense of the step, which is critical for learning [15, 33, 45]. Not only did we see reflection on individual examples, but also on a larger problem structure. Even when a student copied all the examples, she was able to connect the series of examples to create a problem-solving schema [19], and was able to transfer this schema to another task [21], indicating a learning process through reflection-in-action [45].

We also saw that Step Tutor supports different levels of prior knowledge. When a student is unclear about what to do next, she takes a longer time to interact with an example, and uses various activities to make sense of the example step, such as running, comparing, copying the code, and reflecting on the step. When a student only needs a quick debugging tip to move forward, she simply searches through the example code and finds the change she needs to make. This finding aligns with the expertise reversal effect [29], indicating the more expertise one has, the less information she needs when asking for help. We believe the variety of ways of interactions in Step Tutor provides different students with tailored support.

7 LIMITATIONS AND FUTURE WORK

Our user study includes several limitations. 1) With only a small sample of students and two short programming tasks, our study may not generalize to other student groups. But instead of widely gathering data to conclude the benefits of the system, we conducted an in-depth analysis by closely analyzing each individual student's experience. 2) As an initial pilot study, our analysis did not measure learning or progress quantitatively. Therefore, we cannot make strong claims about our system's ability to support progress and learning. But, we collected rich data that depicts a variety of experiences that informs us of Step Tutor's affordances.

Findings in our study also reveal a research space to measure and provide personalization. While no student reported inherent deficiencies in the system, our result from students' interactions with Step Tutor revealed to us a variety of different levels of interactions with the system. For example, 1) students in the high-use group requested help frequently, and one of them expressed their high-level goal as expediently solving the task, which is an maladaptive help-seeking behavior. On the other hand, we also observed that 2) some students in the low-use group didn't request any help. We have closely analyzed the data, and concluded that 1) Instead of abusing help, students who frequently requested help have still experienced a step-by-step demonstration of a Worked Example, as in the Worked Example Effect [48], and discussed having connected the series of examples and transferred to another task; and 2) students who did not request help also did not get stuck. So they did not deliberately avoid help-seeking when they could have benefited from requesting the help[3]. These behavioral patterns of seeking and not seeking help is "preferred" [2] and doesn't statistically decrease learning outcomes [44, 50]. However, from a limited amount of samples, this conclusion may not be generalizable. In the future, we should use quantitative measures to evaluate the learning benefits of these specific types of behaviors, and also for different types of students.

The different levels of interactions with Step Tutor also indicate that students can bring different predispositions (e.g., prior knowledge, goal orientation,programming preference) into their programming experience, and so should benefit from different levels of programming support. While we were able to offer such personalization through choices of interactions in Step Tutor, we should also discover the contextual factors that lead to these different levels of interactions, and create more personalized and flexible forms of feedback to students in the future.

8 CONCLUSION

In this paper, we presented Step Tutor, that provides a combination of the immediate relevance offered by next-step hints, as well as the learning benefits provided by Worked Examples. Our user study shows the wide variety of interactions students employed with Step Tutor, and also suggests the need to personalize support for each individual student.

9 ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1917885.

REFERENCES

[1] Chadia Abras, Diane Maloney-Krichmar, Jenny Preece, et al. 2004. User-centered design. *Bainbridge, W. Encyclopedia of Human-Computer Interaction. Thousand Oaks: Sage Publications* 37, 4 (2004), 445–456.
[2] Vincent Aleven. 2013. Help seeking and intelligent tutoring systems: Theoretical perspectives and a step towards theoretical integration. In *International handbook of metacognition and learning technologies*. Springer, 311–335.
[3] Vincent Aleven, Bruce M McLaren, and Kenneth R Koedinger. 2006. Toward computer-based tutoring of help-seeking skills. *Help seeking in academic settings: Goals, groups, and contexts* (2006), 259–296.
[4] 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), 205–223.
[5] Vincent Aleven, Elmar Stahl, Silke Schworm, Frank Fischer, and Raven Wallace. 2003. Help seeking and help design in interactive learning environments. *Review of educational research* 73, 3, 277–320.
[6] Computing Research Association et al. 2017. Generation CS: Computer science undergraduate enrollments surge since 2006. *Retrieved March* 20 (2017), 2017.
[7] Robert K Atkinson, Alexander Renkl, and Mary Margaret Merrill. 2003. Transitioning from studying examples to solving problems: Effects of self-explanation prompts and fading worked-out steps. *Journal of educational psychology* 95, 4 (2003), 774.
[8] Albert Bandura. 1977. Self-efficacy: toward a unifying theory of behavioral change. *Psychological review* 84, 2 (1977), 191.
[9] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.
[10] Peter Brusilovsky. 2001. WebEx: Learning from Examples in a Programming Course.. In *WebNet*, Vol. 1. 124–129.
[11] 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.
[12] Michelene TH Chi and Ruth Wylie. 2014. The ICAP framework: Linking cognitive engagement to active learning outcomes. *Educational psychologist* 49, 4 (2014), 219–243.
[13] Ruth C Clark, Frank Nguyen, and John Sweller. 2011. *Efficiency in learning: Evidence-based guidelines to manage cognitive load.* John Wiley & Sons.
[14] Jarno Coenen, Sebastian Gross, and Niels Pinkwart. 2017. Comparison of Feedback Strategies for Supporting Programming Learning in Integrated Development Environments (IDEs). In *International Conference on Computer Science, Applied Mathematics and Applications*. Springer, 72–83.

[15] Allan Collins, John Seely Brown, and Susan E Newman. 1988. Cognitive apprenticeship: Teaching the craft of reading, writing and mathematics. *Thinking: The Journal of Philosophy for Children* 8, 1 (1988), 2–10.

[16] Albert T 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 factors in computing systems*. ACM, 245–252.

[17] Bob Edmison, Stephen H. Edwards, and Manuel A. Pérez-Quiñones. 2017. Using Spectrum-Based Fault Location and Heatmaps to Express Debugging Suggestions to Student Programmers *(ACE '17)*. Association for Computing Machinery, New York, NY, USA, 48–54.

[18] Dan Garcia, Brian Harvey, and Tiffany Barnes. 2015. The beauty and joy of computing. *ACM Inroads* 6, 4 (2015), 71–79.

[19] Dedre Gentner, Jeffrey Loewenstein, and Leigh Thompson. 2003. Learning and transfer: A general role for analogical encoding. *Journal of Educational Psychology* 95, 2 (2003), 393.

[20] Peter Gerjets, Katharina Scheiter, and Richard Catrambone. 2004. Designing instructional examples to reduce intrinsic cognitive load: Molar versus modular presentation of solution procedures. *Instructional Science* 32, 1-2 (2004), 33–58.

[21] Mary L Gick and Keith J Holyoak. 1983. Schema induction and analogical transfer. *Cognitive psychology* 15, 1 (1983), 1–38.

[22] Sebastian Gross, Bassam Mokbel, Benjamin Paassen, Barbara Hammer, and Niels Pinkwart. 2014. Example-based feedback provision using structured solution spaces. *International Journal of Learning Technology 10* 9, 3 (2014), 248–280.

[23] Sebastian Gross and Niels Pinkwart. 2015. Towards an integrative learning environment for java programming. In *2015 IEEE 15th International Conference on Advanced Learning Technologies*. IEEE, 24–28.

[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] Stuart Hansen and Erica Eddy. 2007. Engagement and frustration in programming projects. *ACM SIGCSE Bulletin* 39, 1 (2007), 271–275.

[26] Gregory Hume, Joel Michael, Allen Rovick, and Martha Evens. 1996. Hinting as a tactic in one-on-one tutoring. *The Journal of the Learning Sciences* 5, 1 (1996), 23–47.

[27] Michelle Ichinco, Kyle J Harms, and Caitlin Kelleher. 2017. Towards Understanding Successful Novice Example User in Blocks-Based Programming. *Journal of Visual Languages and Sentient Systems* 3 (2017), 101–118.

[28] Michelle Ichinco and Caitlin Kelleher. 2015. Exploring novice programmer example use. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 63–71.

[29] Slava Kalyuga. 2009. The expertise reversal effect. IGI Global.

[30] Stuart A Karabenick. 2004. Perceived achievement goal structure and college student help seeking. *Journal of educational psychology* 96, 3 (2004), 569.

[31] Andrew J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206.

[32] Kenneth R Koedinger and Vincent Aleven. 2007. Exploring the assistance dilemma in experiments with cognitive tutors. *Educational Psychology Review* 19, 3 (2007), 239–264.

[33] Kenneth R Koedinger, Albert T Corbett, and Charles Perfetti. 2012. The Knowledge-Learning-Instruction framework: Bridging the science-practice chasm to enhance robust student learning. *Cognitive science* 36, 5 (2012), 757–798.

[34] Hannu Kuusela and Paul Pallab. 2000. A comparison of concurrent and retrospective verbal protocol analysis. *The American journal of psychology* 113, 3 (2000), 387.

[35] Samiha Marwan, Joseph Jay Williams, and Thomas Price. 2019. 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*. 61–70.

[36] Samiha Marwan, Nicholas Lytle, Joseph Jay Williams, and Thomas Price. 2019. 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*. 520–526.

[37] Elizabeth Patitsas, Michelle Craig, and Steve Easterbrook. 2013. Comparing and contrasting different algorithms leads to increased student learning. In *Proceedings of the ninth annual international ACM conference on International computing education research*. ACM, 145–152.

[38] Thomas Price, Rui Zhi, and Tiffany Barnes. 2017. Evaluation of a Data-driven Feedback Algorithm for Open-ended Programming. *International Educational Data Mining Society* (2017).

[39] Thomas W Price, Yihuan Dong, and Dragan Lipovac. 2017. iSnap: towards intelligent tutoring in novice programming environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, 483–488.

[40] Thomas W Price, Zhongxiu Liu, Veronica Cateté, and Tiffany Barnes. 2017. Factors Influencing Students' Help-Seeking Behavior while Programming with Human and Computer Tutors. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, 127–135.

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

[42] Bethany Rittle-Johnson and Jon R Star. 2007. Does comparing solution methods facilitate conceptual and procedural knowledge? An experimental study on learning to solve equations. *Journal of Educational Psychology* 99, 3 (2007), 561.

[43] Kelly Rivers. 2017. Automated Data-Driven Hint Generation for Learning Programming. (2017).

[44] Ido Roll, Ryan SJ d Baker, Vincent Aleven, and Kenneth R Koedinger. 2014. On the benefits of seeking (and avoiding) help in online problem-solving environments. *Journal of the Learning Sciences* 23, 4 (2014), 537–560.

[45] Donald A Schön. 1987. Teaching artistry through reflection in action: Educating the reflective practitioner. (1987).

[46] Benjamin Shih, Kenneth R Koedinger, and Richard Scheines. 2011. A response time model for bottom-out hints as worked examples. *Handbook of educational data mining* (2011), 201–212.

[47] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive science* 12, 2 (1988), 257–285.

[48] John Sweller. 2006. The worked example effect and human cognition. *Learning and instruction* (2006).

[49] W. Price Thomas, Jay Williams Joseph, Solyst Jaemarie, and Marwan Samiha. 2020. Engaging Students with Instructor Solutions in Online Programming Homework. In *To be published in the 2020 Association for Computing Machinery's Special Interest Group on Computer Human Interaction (ACM SIGCHI '20)*.

[50] John Gregory Trafton and Brian J Reiser. 1994. *The contributions of studying examples and solving problems to skill acquisition*. Ph.D. Dissertation. Citeseer.

[51] Lev Vygotsky. 1978. Interaction between learning and development. *Readings on the development of children* 23, 3 (1978), 34–41.

[52] Wengran Wang, Rui Zhi, Alexandra Milliken, Nicholas Lytle, and Thomas W. Price. 2020. Crescendo: Engaging Students to Self-Paced Programming Practices *(SIGCSE '20)*. ACM, 859–865.

[53] 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 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 740–751.

[54] Rui Zhi, Samiha Marwan, Yihuan Dong, Nicholas Lytle, Thomas W Price, and Tiffany Barnes. 2019. Toward Data-Driven Example Feedback for Novice Programming. *Proceedings of the International Conference on Educational Data Mining* (2019), 218–227.

[55] Rui Zhi, Thomas W Price, Nicholas Lytle, Yihuan Dong, and Tiffany Barnes. 2018. Reducing the State Space of Programming Problems through Data-Driven Feature Detection. In *Educational Data Mining in Computer Science Education (CSEDM) Workshop@ EDM*.

[56] Rui Zhi, Thomas W Price, Samiha Marwan, Alexandra Milliken, Tiffany Barnes, and Min Chi. 2019. Exploring the Impact of Worked Examples in a Novice Programming Environment. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, 98–104.