International Journal of Artificial Intelligence in Education (2021) 31:784–819
https://doi.org/10.1007/s40593-020-00217-z

ARTICLE

When Does Scaffolding Provide Too Much Assistance? A Code-Tracing Tutor Investigation

Jay Jennings[1] · Kasia Muldner[1]

Accepted: 11 September 2020 / Published online: 23 October 2020

Abstract
When students are first learning to program, they not only have to learn to write programs, but also how to trace them. Code tracing involves stepping through a program step-by-step, which helps to predict the output of the program and identify bugs. Students routinely struggle with this activity, as evidenced by prior work and our own experiences in the classroom. To address this, we designed a *Code Tracing* (CT)-Tutor. We varied the level of assistance provided in the tutor, based on (1) the interface scaffolding available during code tracing, and (2) instructional order, operationalized by when examples were provided, either before or after the corresponding problem was solved. We collected data by having participants use the tutor to solve code tracing problems ($N = 97$) and analyzed both learning outcomes and process data obtained by extracting features of interest from the log files. We used a multi-layered approach for the analysis, including standard inferential statistics and unsupervised learning to cluster students by their behaviors in the tutor. The results show that the optimal level of assistance for code tracing falls in the middle of the assistance spectrum included in the tutor, but also that there are individual differences in terms of optimal assistance for subgroups of individuals. Based on these results, we outline opportunities for future work around personalizing instruction for code tracing.

Keywords Code tracing · Tutoring system · Assistance · Worked examples · Programming instruction

In memory of Jim Greer

Back in the 2000s when I was in my early years of graduate school, I first met Jim at one of the AIED conferences. He was introduced to me by my close friend Andrea Bunt, a fellow graduate student who knew Jim from her time at the

✉ Kasia Muldner
kasia.muldner@carleton.ca

[1] Institute of Cognitive Science, Carleton University, Ottawa, Canada

University of Saskatchewan. A conference can be intimidating for those new to the community – everyone seems to know each other and while we are told to network and meet people, that can be challenging. I've always appreciated Jim's kindness and mentorship – since that first introduction, he took time to talk to me during the various subsequent conferences, even though he had many people to talk to as he knew everybody in the community. I got to know his work over the years, and his impact on AIED scholarship cannot be overstated. He and his students have consistently produced groundbreaking work in diverse areas, including Bayesian student modeling, programming education, peer help systems, and the evaluation of tutoring systems, to name a few. One of his areas of expertise was Bayesian student modeling, something I also worked on for my Ph.D. research. Jim was my external examiner for my Ph.D. defense. Frankly, like many candidates I was anxious about his report, especially given his extensive expertise but there was no need. Jim was constructive, kind, and fair. I held on to his external report, both as a happy memory and as a standard for how to write such reports for others. In addition to these exceptional qualities, Jim also had a brilliant sense of humour. I still remember him doing the opening remarks for AIED'07. It was the first day of the conference and it was early. While typical opening sessions are notoriously dry, Jim had us laughing from start to finish. The last time I saw Jim was at ITS in Montreal in 2018. As always, we chatted briefly, and I left with a feeling of gratitude that I had the chance to bump into him and we could share a moment. Jim had that effect on people that those rare individuals with confidence, humility, and wisdom have. The AIED community will sorely miss him.

Kasia Muldner

Introduction

Programming involves writing instructions, called code, that tell a computer how to accomplish a specific task. Because computers do not understand human languages like English, programs are written using a specialized programming language. Since the influential Wing (2006) article promoted the need for everyone to learn computational thinking skills, there has been increased recognition of the broad value of these skills in general, and programming in particular (Grover and Pea 2013). For instance, programming classes are being integrated into primary and secondary schools around the world (Brown et al. 2013; Rich et al. 2017), and some universities require that all students, not only computer science majors, take a programming class (Nelson et al. 2017).

Learning to program does come with challenges. Students frequently struggle with syntax errors (Denny et al. 2011), understanding the logical flow of a program (Lam et al. 2008), building mental models of common program elements (Hertz and Jump 2013), and tracing through programs to understand them (Vainio and Sajaniemi 2007). In general, many students find programming difficult, with the failure rate hovering around 30% in introductory university classes (Bennedsen and Caspersen 2007). Given the challenges related to learning to program, some early work focused on identifying expert programmers' mental representations (Gilmore and Green 1988), and how these representations could be incorporated into teaching activities to help novices learn (Schank et al. 1993; Soloway 1986). There was also interest in computational support for programming activities delivered through tutoring systems (Anderson et al. 1989; Bhuiyan et al. 1994). Jim Greer and colleagues championed some of this work. To illustrate, Greer and McCalla (1989) proposed that automated diagnosis of students' programs in tutoring systems could be facilitated by using hierarchical representations. These representations, referred to as granularity hierarchies, allowed the system to detect student errors at different levels of abstraction. A subsequent project involved the PETAL learning environment (Bhuiyan et al. 1994). This system scaffolded Lisp program generation through a variety of tools (e.g., syntactic support as well as high-level analytic support that helped students design and implement recursive programs). Other tutoring systems have also supported program generation by providing hints (Price et al. 2019; Price et al. 2017; Rivers and Koedinger 2017) as well self-explanation prompts (Fabic et al. 2019). For example, Rivers and Koedinger (2017) used data-driven techniques to tailor hints to a given student, so that when an impasse was encountered, students were given tailored help aimed at helping them overcome the impasse. An alternative (or complimentary) approach to tutoring systems involves providing peer support. The i-Help system pioneered by Greer and colleagues (Greer et al. 2001) assisted learners in a variety of problem-solving activities, including programming. Students enrolled in university classes could post questions to public forums and could also ask for 1-on-1 help from a peer, who would be located by an i-Help agent. There is also work outside the AIED community on supporting novices through the design of environments for programming activities (Costa and Miranda 2017; Whittall et al. 2017; Bau et al. 2017; Weintrop and Wilensky 2018).

As summarized above, there is a range of support available for program generation. While knowing how to write code is clearly an important skill, a related critical skill is code tracing. Code tracing involves the simulation of a computer program step-by-step. This activity helps identify bugs (errors in programs), predict code output, and in general improves learning outcomes as well as engagement (Cunningham et al. 2017; Lopez et al. 2008; Hertz and Jump 2013; Venables et al. 2009; Kumar 2013; Murphy et al. 2012). Despite the fact that students struggle with code tracing, to date there is little work on how to support students during this activity. This project takes a step in filling this gap. The project was inspired by our experiences teaching first-year programming. In our department of cognitive science, all students, regardless whether they are in the computer-science concentration or not, have to take a first-year programming course, currently taught in Python. As reported in prior research involving computer science majors, our teaching experiences highlighted that code tracing is challenging for students first learning to program. One reason for this may be lack of adequate practice opportunities that provide assistance and encouragement for the process. Thus, our long-term goal is to develop an intelligent tutoring system for code tracing activities and incorporate it into our course, so that students have ample opportunities to practice code tracing and receive the tutor's feedback and assistance.

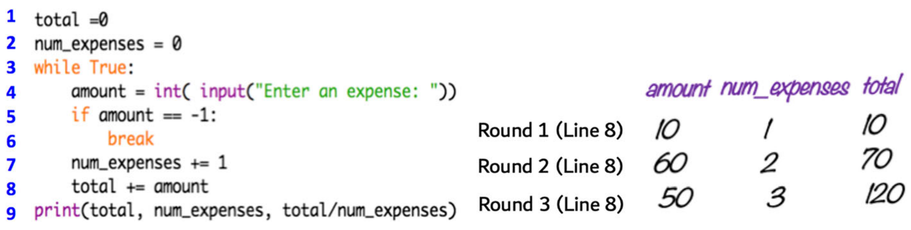
Here we describe the first phase of this project, namely the design and evaluation of a code-tracing tutor protype called Code Tracing (CT)-Tutor. The tutor scaffolds the process of code tracing through two forms of assistance: interface design and worked-out examples incorporated into the tutor. Currently, the tutor is not an intelligent tutoring system because it does not include a student model and so does not personalize instruction (beyond basic correctness feedback). While adding intelligent support is certainly a future goal, we wanted to evaluate the present prototype prior to adding functionalities to it. Before describing the tutor, we present related work.

Programming and Code Tracing: Foundations and Related Work

Code Tracing: Background

Code tracing involves the simulation of a computer program step-by-step in order to predict how the program manipulates data and what output (if any) it produces. How should code tracing be realized? One option is specially-designed debuggers that allow a user to step through a program. However, our anecdotal experience in the classroom with introducing debuggers to novice programmers was that students were not comfortable with debuggers because they presented an additional technical hurdle (learning the debugger interface in addition to learning to program). Another option for realizing code tracing is through paper and pencil activities. Such activities are more familiar to students and there is evidence they are effective for tracing short programs of the types novices initially learn to produce. To illustrate this process, Fig. 1, left, shows a short python program that tallies a user's expenses, and one way to trace this program (right), with the column labels corresponding to the program variables that need to be traced (*amount*, *net expenses, total*) – the code trace involves keeping track of the variable values during program execution. Even this brief example illustrates that keeping track of variables in working memory is not feasible, given that variable values change as a program runs (e.g., because a variable is in a loop). Indeed, Cunningham et al. (2017) found a positive association between code tracing on paper and test performance, with students who code-traced effectively scoring higher on a post-test than students who did only partial tracing, rewrote parts of the code, or did not code trace at all. The instructor also had an effect on how much students code traced and the quality of the traces, suggesting that modeling the code tracing process is beneficial. While this was not an experimental study, it provides promising indications that code tracing is associated with learning and is a skill that can be taught.

Other work, also relying on observational methods, reported a positive association between code tracing on paper and programming performance (Lopez et al. 2008; Venables et al. 2009; Kumar 2013; Lister et al. 2009). A complimentary skill that emerged from these studies is code explanation, namely describing a program in plain English rather than with programming jargon (Venables et al. 2009; Lister et al. 2009). To illustrate, one potential explanation of the program in Fig. 1 is that it asks the user



```
1 total =0
2 num_expenses = 0
3 while True:
4     amount = int( input("Enter an expense: "))
5     if amount == -1:
6         break
7     num_expenses += 1
8     total += amount
9 print(total, num_expenses, total/num_expenses)
```

	amount	num_expenses	total
Round 1 (Line 8)	10	1	10
Round 2 (Line 8)	60	2	70
Round 3 (Line 8)	50	3	120

Fig. 1 A short python program (left) and its code trace (right)

for their expenses and calculates the total amount spent as well as the number of expenses. Subsequent work replicated the finding that code explanation is positively correlated with the ability to write code (Murphy et al. 2012). We hypothesize that code explanation and code tracing are complementary skills: code explanation involves understanding the program from a high-level perspective, while code tracing identifies the low-level blocks needed to realize the high-level goal.

Support for Code Tracing

Both code tracing and code explanation are challenging for novices (Simon and Snowdon 2011; Murphy et al. 2012; Lahtinen et al. 2005), and so support is needed to help students learn these skills. One way to provide such support is through teaching practices. Hertz and Jump (2013) adopted code-tracing centered instruction after observing that students held incorrect mental models of programs. The instruction corresponded to a lesson on a code-tracing topic, followed by an application phase during which students worked in groups. Students reported that this style of teaching was helpful, and the drop/fail rate in the course was reduced from 25% to 8%. However, even after this overhaul of the pedagogy, students often did not code trace on their own, something we also found in our classrooms. This highlights the need for additional code-tracing guidance, such as that provided by educational technologies. To date, however, little technological support exists for code tracing, with a notable exception we now describe.

Nelson et al. (2017) developed the PLTutor, which provides an interface for stepping forward and backwards during the execution of a program (both at the line level and at higher levels of granularity), along with explanations at salient steps. For example, PLTutor will stop during an evaluation of an *if* statement to provide an explanation of what the program is doing, aimed to help students understand how the parts of the program relate to each other. The theory underlying the PLTutor's design is that novice programmers will learn to code trace by being exposed to the rules of how code is processed by a programming language's interpreter. Thus, PLTutor's explanations include technical terms like namespaces and stacks. This is certainly suitable for a traditional computer science audience but would need to be adapted for students from less traditional backgrounds, including non-computer science majors. This is the case for our target audience, as our students come from interdisciplinary backgrounds and many are not computer science majors. PLTutor was evaluated by comparing learning outcomes of students who worked with the tutor against outcomes of students learning from Code Academy materials. The total learning gains were greater for the PLTutor participants than the Code Academy participants. This is a positive finding, although the results have to be interpreted with caution as there were a priori differences between the groups at pre-test.

One way to support code tracing is by giving students access to explanations, similar to hints, as the PLTutor did. An alternative approach is to supplement code tracing activities with examples of code traces to illustrate the process of code tracing. We recently incorporated code-tracing examples into instructional videos about code tracing (Lee and Muldner 2020). We found that overall the videos significantly improved learning, but in this work we did not manipulate instruction related to the examples, such as where they occurred in the instructional sequence. While to the best of our knowledge the work reported in this paper is the first to incorporate examples about code-tracing activities into a computer tutor, examples have been successfully used in tutoring systems in other domains (Conati and Vanlehn 2000; Muldner and Conati 2010; McLaren et al. 2016; Najar and Mitrovic 2013), including program construction (Hosseini and Brusilovsky 2017). In general, there is established evidence that examples are highly beneficial for novices (Chi et al. 1989; Renkl 2014; Sweller and Cooper 1985; van Gog et al. 2011) with the caveat that this positive effect diminishes as expertise develops (Sweller et al. 2003). A seminal work illustrating the benefits of examples is the Sweller and Cooper (1985) study, in which students were asked to study an example of a step-by-step solution to an algebra problem, and then solve a similar algebra problem without the presence of the example (each session included a series of example-problem pairs). Students given examples learned more compared to students who only solved problems, highlighting the benefit of worked examples. In the context of tutoring systems, however, recent findings have been mixed, with some studies not finding evidence that examples are beneficial (e.g., Salden et al. 2009; Corbett et al. 2013). A possible explanation in the discrepancy of findings is that in the original studies, all activities were done on paper, while studies not finding benefits of examples involved tutoring systems. Since tutoring systems scaffold problem solving in various ways, the effect of examples may be affected by that scaffolding.

When examples are available, learning outcomes are maximised when students invest time and effort to study the examples. Chi et al. (1989) showed that students who self-explain example solutions learn more than students who merely paraphrase example solutions. Self-explanation can involve inferences that go over what is explicitly shown in the example, for instance to identify the rules that generated the example solution steps. The self-explanation effect has been replicated through hundreds of studies (e.g., Rittle-Johnson et al. 2017; Wylie and Chi 2014; Weerasinghe and Mitrovic 2006) and various learning environments have added tools to encourage students to self-explain, in domains such as physics, mathematics, and program generation (Conati and Vanlehn 2000; Fabic et al. 2019).

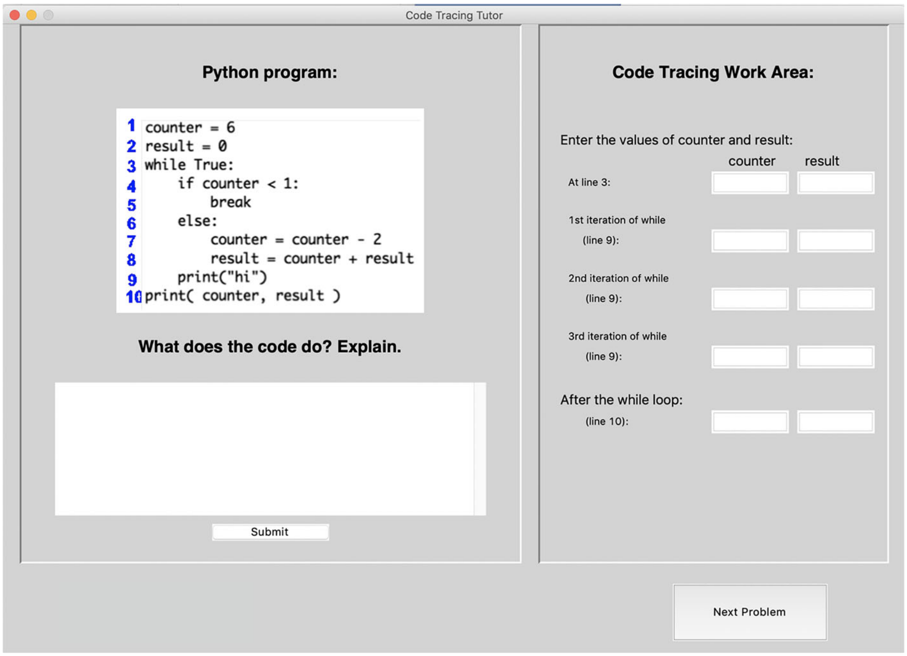
There are various ways that examples can be integrated into instructional activities. A common approach is to show the example first, followed by the problem (Sweller and Cooper 1985), or alternatively, to show the example alongside the problem (Jennings and Muldner 2020). In either case, the example can guide learning of the procedure required for solving the subsequent problem. In contrast, van Gog (2011) asked students to *first* solve a problem and then be given the corresponding example, comparing this less traditional instructional order to the standard approach of showing the example before the problem; the domain for this study was electric circuits. Intuitively, if the problem is presented first, students may struggle to solve the problem more than if the example is first. This is because an example-problem ordering provides more assistance than a problem-example ordering, as the example illustrates the procedure before the problem has to be solved, rather than after the problem is attempted. Van Gog speculated this initial struggle may actually have beneficial effects, but this was not confirmed: students in the problem-example order performing similarly to students in the example-problem order (i.e., no significant difference between instructional orders was found). A second study by van Gog et al. (2011) found a significant learning advantage of showing the example before the problem, rather than after the problem. In general, however, more work is needed before concluding that a given instructional ordering is best, given that other work, albeit using a different instructional design, has found precedent for challenging students by giving them the problem-solving activity first, before instruction. Specifically, in the productive failure paradigm (Kapur 2014; Kapur 2016; Kapur and Bielaczyc 2012) students are asked to solve open-ended problems that they lack knowledge to successfully complete and only once they have struggled with them are they provided with the canonical solution in the form of a lesson.

To summarize, little work exists on the design and evaluation of educational technologies that provide support for code tracing activities, particularly for novice programmers not enrolled in a traditional computer science stream. To fill this gap, we have been working on designing and implementing a code tracing (CT)-Tutor, and now describe the prototype that we built as a first step towards filling this goal.

Code Tracing (CT)-Tutor

As will become evident shortly, the CT-Tutor is not like a debugger, because it aims to support code-tracing activities in a manner similar to how novices trace programs on paper. Compared to paper activities, however, a computer-based tutor has several advantages because it can (1) explicitly scaffold the process by requiring that students perform the code trace in a prescriptive, principled way, (2) provide assistance such as immediate feedback for correctness, and (3) log student actions that can be subsequently mined to identify patterns of behaviours that are beneficial (or not) for learning.

Our goal was to make the tutor usable for students who are not computer science majors and who are just starting to learn to program. Thus, we avoided integrating technical terms into the design of the tutor. In general, the CT-Tutor interface was designed based on both our experience teaching programming as well as prior research



Python program:

```
1  counter = 6
2  result = 0
3  while True:
4      if counter < 1:
5          break
6      else:
7          counter = counter - 2
8          result = counter + result
9      print("hi")
10 print( counter, result )
```

What does the code do? Explain.

Code Tracing Work Area:

Enter the values of counter and result:

	counter	result
At line 3:		
1st iteration of while (line 9):		
2nd iteration of while (line 9):		
3rd iteration of while (line 9):		
After the while loop: (line 10):		

Fig. 2 The CT-Tutor interface with high scaffolding

reporting how novice students trace programs on paper (Cunningham et al. 2017). Each problem in the CT-Tutor corresponded to a short Python program (e.g., Fig. 2, top left panel) and instructions asking the student to trace the program in order to predict the program's output. We integrated two types of support for code tracing into the tutor, namely interface scaffolding and examples, as we now describe.

Assistance for Code Tracing Through Interface Scaffolding

Students struggle with code tracing and one potential reason for this is insufficient scaffolding (i.e., knowing where to start, which variables to track). To investigate the effect of scaffolding, we implemented two tutor interfaces: high vs. reduced scaffolding for code tracing.

High-Scaffolding Interface

As the name implies, the high-scaffolding interface is designed to provide a lot of guidance for the code tracing process, by informing students which variables to track and by requiring they enter values of those variables step-by-step as the variable values changed during the program. This scaffolding is realized through the *Code Tracing Work Area* panel, top right, Fig. 2. The design of the *Code Tracing Work Area* is based on the table format used by effective learners using paper and pencil materials (Cunningham et al. 2017). To illustrate, for the problem in Fig. 2, students are guided to keep track of the *counter* and *result* variables – since the program involves a loop, the table includes multiple rows to reflect that the variable values change for each loop iteration. The tutor provides immediate feedback for correctness on each entry (correct entries are coloured green; incorrect entries are coloured red).

To guide the process of code tracing, students are not allowed to skip steps and their entries in a given row in the code-tracing table have to be correct before they can move on to the next row. Thus, initially all the rows in the code-tracing table are locked (and grayed out slightly), with the exception of the first row. Once the entries for the first row are correctly generated, the tutor unlocks the second row, and so on. Based on piloting, the tutor generates a warning message when a locked row is clicked, telling the user they have to first produce the prior rows' entries.

This form of scaffolding is fairly strict. While it is an open and empirical question as to whether allowing more flexibility in terms of solution generation would be beneficial, there is some precedent for strict tutor guidance in other domains (e.g., Anderson et al. 1995). In the present domain, one argument for the benefits of this restrictive interface design is that it might encourage code tracing. This conjecture is based on prior work showing that if students do not know which variables to track or how to track them, they tend to avoid code tracing altogether or make mistakes because they skip steps (Cunningham et al. 2017). However, the alternative possibility is that this design will make students overly reliant on producing the correct answer, without self-explaining how the answer was obtained. We address this question through an empirical study we describe below.

In addition to guiding students in the code tracing work area and providing feedback for correctness, a third form of scaffolding for code tracing comes in the form of encouraging students to enter a high level, plain English explanation of what the code does (see the panel under the "*What does the code do? Explain*" label, bottom left, Fig.

2). If students try to move on to the next problem without providing an explanation (by clicking the *Next Problem* button, not shown in Fig. 2), the tutor reminds them to provide one. The tutor does not currently check the content of what is typed in the explanation box, although this is something we plan to integrate in future versions.

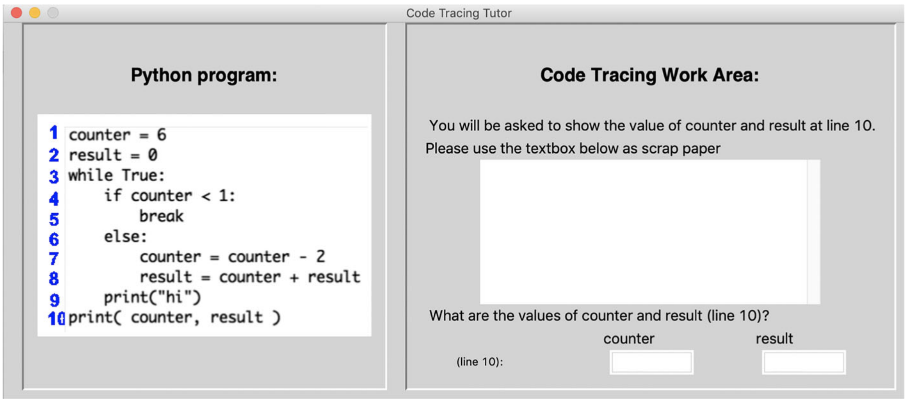
Reduced-Scaffolding Interface

To evaluate the effect of interface scaffolding for code tracing, we created a second version of the CT-Tutor interface that provided only reduced scaffolding for code tracing. An example of the reduced-scaffolding interface is shown in Fig. 3. As was the case for the high-scaffolding interface, the reduced version also requires students to predict the output of the target program and to type their answer into the provided entry boxes (see boxes labelled *counter* and *result*, bottom right, Fig. 3); also as in the high-scaffolding interface, this version provided immediate feedback for correctness on the entries related to the these final values.

However, there are several key differences between the reduced- and high-scaffolding interfaces. The reduced-scaffolding interface does not enforce a specific strategy for code tracing, instead providing a free-form textbox that students could use (but are not required to), see *Code Tracing Work Area*, middle right, Fig. 3. Because the entries produced in this textbox are free form and not guided in any way, and thus challenging to automatically grade, the tutor does not evaluate their correctness. Thus, in this interface, the feedback for correctness is provided only for the final answer, but not the intermediate steps. Moreover, the reduced-scaffolding interface does not explicitly ask students to generate a plain English explanation of the code, although students are free to do so in the code-tracing work area.

Assistance for Code Tracing Through Examples

Beyond interface scaffolding, the second form of assistance for code tracing in the CT-Tutor is in the form of worked examples. In the context of code tracing, a worked example shows all the intermediate solution steps related to code tracing, as well as the final solution. Since students naturally use a table design when code



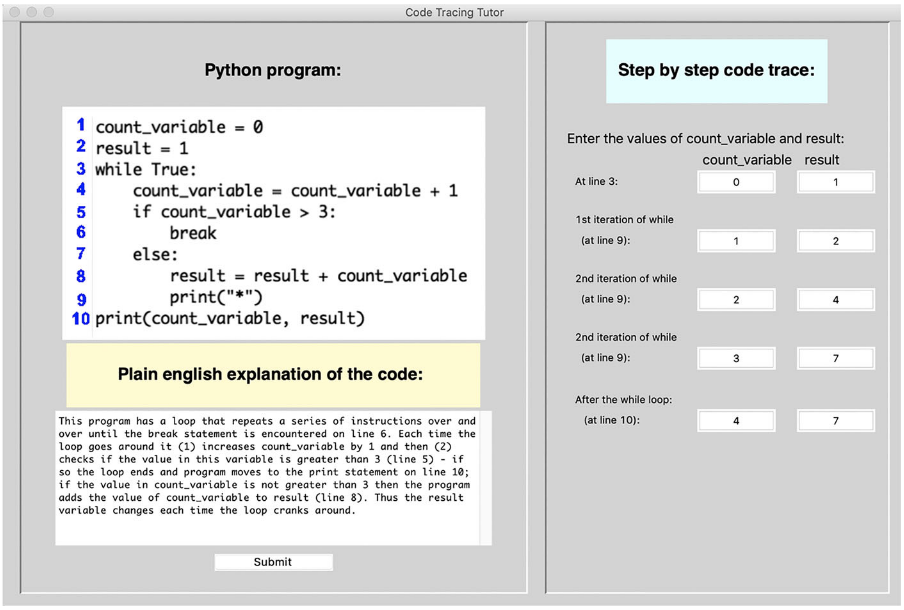
```
1 counter = 6
2 result = 0
3 while True:
4     if counter < 1:
5         break
6     else:
7         counter = counter - 2
8         result = counter + result
9     print("hi")
10 print( counter, result )
```

You will be asked to show the value of counter and result at line 10.
Please use the textbox below as scrap paper

What are the values of counter and result (line 10)?

	counter	result
(line 10):		

Fig. 3 The CT-Tutor interface with reduced scaffolding



```
1  count_variable = 0
2  result = 1
3  while True:
4      count_variable = count_variable + 1
5      if count_variable > 3:
6          break
7      else:
8          result = result + count_variable
9          print("*")
10 print(count_variable, result)
```

This program has a loop that repeats a series of instructions over and over until the break statement is encountered on line 6. Each time the loop goes around it (1) increases count_variable by 1 and then (2) checks if the value in this variable is greater than 3 (line 5) - if so the loop ends and program moves to the print statement on line 10; if the value in count_variable is not greater than 3 then the program adds the value of count_variable to result (line 8). Thus the result variable changes each time the loop cranks around.

Enter the values of count_variable and result:

	count_variable	result
At line 3:	0	1
1st iteration of while (at line 9):	1	2
2nd iteration of while (at line 9):	2	4
2nd iteration of while (at line 9):	3	7
After the while loop: (at line 10):	4	7

Fig. 4 An example showing a step by step solution to a code-tracing problem, including the variable values as the loop executes in the code trace panel (see right panel), as well as a plain English explanation (bottom left)

tracing (Cunningham et al. 2017), like the one used in the high scaffolding interface, the format of the examples in the CT-Tutor followed this approach. Fig. 4 shows an example in the CT-Tutor. While the example appears similar to a problem in the high-scaffolding interface, in the example the explanation and the code-tracing entries are filled in with correct answers, and all text boxes are locked (i.e., cannot be edited).

As described above, the examples in CT-Tutor were shown separately from the problems - either before the problem was presented or after (instructional order was one of the factors we varied in our study because we wanted to analyze its effect on learning). If the example came before the problem, then it offered higher assistance than if the example came after the student attempted to solve the problem (van Gog et al. 2011).

CT-Tutor Logistics

While we have benefited from the CTAT tutoring framework for tutor construction for several projects (Borracci et al. 2020; Sale and Muldner 2019), for this project we needed more flexibility in terms of the tutor functionalities (e.g., rapid problem integration). Thus, we implemented the tutor from scratch, without relying on any tutor-building frameworks.

In the present tutor prototype, a human author provides a text file that specifies the necessary parameters: (1) the problem file name, which contains a screenshot of the python program shown in the interface; (2) a corresponding explanation of the program; (3) a specification for the code tracing interface (i.e., how many rows and columns to include in the table); (4) a solution for each of the entries in the interface. The CT-Tutor automatically builds the tutor interface based on the author specification. This latter feature allows the program to flexibly adjust the look and feel of the interface based on the problem, since different Python programs may require different code-tracing layouts. This flexibility was important to us because, when we do integrate the tutor into our programming class, we will have the ability to easily add and modify problems in the tutor.

Evaluation of CT-Tutor: Methods

The CT-Tutor protype described above provides feedback for correctness and two types of assistance (interface scaffolding, worked examples). Before we integrated more complex forms of help and AI methods to enable personalization, we evaluated the tutor through a study that provided data on what was working (and not) in terms of the tutor's design.

In the present work, our goal was to investigate the impact of the two types of assistance in the CT-Tutor on learning and performance. We hypothesized that compared to reduced-scaffolding, the high-scaffolding interface will increase learning because it provides assistance to code tracing (H1). We did not have a hypothesis for the effect of instructional order on learning or interactions between it and scaffolding because there is not yet sufficient prior research related to these aspects. As far as performance, we had a variety of within-tutor variables related to time and correctness of entries (described below), some of which were exploratory and so did not have corresponding hypotheses. For the variables that did, we hypothesized that the number of problems finished will be greater in the high-scaffolding conditions and the example-first conditions (H2), because these provide higher assistance than reduced scaffolding and problem-first conditions. Similarly, we hypothesized students would spent less time solving problems in the high-scaffolding and example-first conditions (H3). As far as error rate based on entries the tutor flagged as incorrect, we did not have specific hypotheses. Error rate, operationalized here errors flagged by the tutor, is affected by the number of opportunities to receive error feedback, and this is higher in the high-scaffolding interface (recall that the reduced-scaffolding interface only provides feedback on the final solution). While we did normalize this variable, we could not fully control for the inherent difference in the two interfaces.

Participants

The study participants were undergraduate students in a medium-sized Canadian University who were enrolled in a first-year class that presented a broad overview of cognitive science ($N = 97$, 67 female). To avoid ceiling effects, participants were not eligible for the study if they had taken any university-level computer science courses. Compensation was 2% bonus course credit for the class, which was added as a percent to their final grade in the course. The study was reviewed and approved by the university ethics board.

Materials

Python Lesson

To provide background knowledge for coding with Python, participants were given a tutorial on the basics of programming and code tracing, including variables, assignment, input and output statements, basic conditional statements, and basic while loops. The tutorial was organized using a slide-deck PowerPoint format with 11 slides. Based on prior work showing that questions in instructional materials can foster learning (Craig et al. 2006; Driscoll et al. 2003), each slide included “speech bubbles” from hypothetical students asking questions, along with an answer (see Fig. 5). The slide deck included two brief videos (approximately two minutes each) demonstrating how to code trace, including how to keep track of variables and their values.

Problems and Examples Used in the CT-Tutor

We populated CT-Tutor with four code-tracing problems and four corresponding examples. The problems that participants had to code trace were short Python programs that involved while loops and conditional statements (for an example see Fig. 3, left, *Python program*). These types of short programs are included in introductory programming classes to familiarize students with programming basics. However, the use of loops makes tracing even these basic programs challenging due to the need to keep track of variables and how their values repeatedly change inside a loop (all problems involved *while* loops).

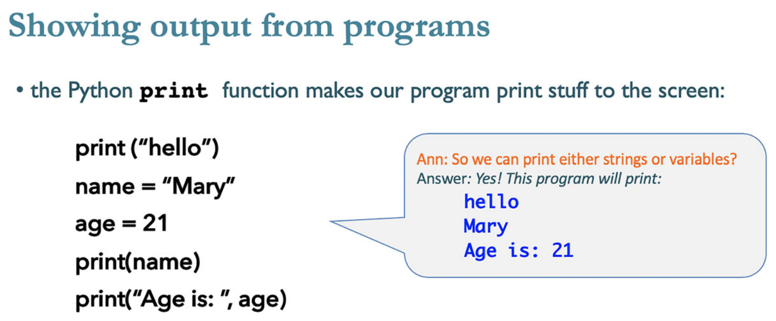
For each problem, we created a corresponding example that showed a step-by-step worked out solution to a similar problem (but with different variable names and values). The examples were shown either before the problem or after.

Pre-Test and Post-Test

The pre-test and post-test were identical except that variable names and values were varied between the two (the pre-test is in Appendix 1). The test had two parts: code tracing and code explanation. The code tracing questions were worth 10 points: four brief 1-point questions related to programs that did not include loops and three 2-point questions related to programs with loops. The two explanation questions related to the loop programs and were worth 2 points each. Thus, the test was scored out of 10 points for code tracing and out of 4 for explanation. We kept these scores separate in our analysis to see how CT-Tutor affected each skill, with a focus on the code-tracing score as this is currently the focus of CT-Tutor’s assistance.

Experimental Design and Study Conditions

We used a between subjects design with two factors: *interface scaffolding* (high, reduced) and *instructional order* (example first, where CT-Tutor showed the example before the


Showing output from programs

• the Python **print** function makes our program print stuff to the screen:

```
print ("hello")
name = "Mary"
age = 21
print(name)
print("Age is: ", age)
```

Ann: So we can print either strings or variables?
Answer: *Yes! This program will print:*

```
hello
Mary
Age is: 21
```

Fig. 5 A snippet of the tutorial showing hypothetical student questions and subsequent answers

corresponding problem, and problem-first, where the example was provided after the corresponding problem). Thus, there were four conditions: reduced-scaffolding problem-first, reduced-scaffolding example-first, high-scaffolding problem-first, and high-scaffolding example-first. Given the between-subjects design, each participant experienced only one condition. Note that the example interface in each of the four conditions was always the same (see Fig. 4 for an example). The four conditions captured three levels of assistance when considering both instructional design and level of scaffolding: (1) the reduced-scaffolding problem-first condition provided low assistance (2) the reduced-scaffolding example-first and high-scaffolding problem-first conditions provided intermediate assistance, and (3) the high-scaffolding example-first condition provided high assistance.

Procedure

Each experimental session was conducted in a quiet room (most sessions were run individually, but several included 2–3 participants, who sat at opposite ends of a large room). Participants were assigned to conditions in a round robin fashion and the procedure for the four conditions was the same. After participants signed an informed consent form, they completed an online survey which gathered demographics data and additional questionnaires (10 min; the questionnaires included items on mindset and related factors – we do not present analysis from these here and so do not describe them). Participants were then asked to read and study the python lesson (20 min). Once they signalled they were done, the materials were taken away and participants completed a paper and pencil pre-test to evaluate their programming knowledge (15 min). The pre-test was done after the instructional materials rather than before because we wanted to have a clean assessment of how much the CT-Tutor influenced learning (we used a similar design in prior studies for the same reason, Muldner et al. 2014).

After the pre-test, participants were introduced to the CT-Tutor and were given 40 min to work on code tracing with the CT-Tutor. The programs they had to trace and the corresponding examples were identical in the four conditions (same four problems and same four examples). CT-Tutor imposed minimal time thresholds, as follows: (1) an example had to be studied for at least one minute (and if a participant tried to move on before that, CT-Tutor informed them they had to study the example more), and (2) a problem had to be worked on until it was either correct or a minimum amount of time had passed. Specifically, if the solution was incomplete or had errors, participants could not move on to the next item (problem or example depending on the condition) unless three minutes had passed (if the solution was complete and correct, they could move on whenever they chose). If participants tried to leave a problem after the three minute threshold but before they had a complete solution, the tutor asked them if they were sure they want to move on and pointed out the missing components (e.g., incomplete solution, missing explanation in the high scaffolding conditions). Participants could choose to say "yes" to move on or could choose to stay (to move on, participants used a button in the interface). Once participants left a problem or example they were not allowed to go back (they were informed about this prior to starting their work). We chose relatively low time thresholds for viewing the example and solving the problem (e.g., one minute may not be sufficient to fully study an example). We did not impose longer thresholds because we were concerned this might induce negative emotions in some students (e.g., frustration over being blocked from moving on).

Once participants completed the tutoring session or time was up, they were given a post-test (as well as a questionnaire to measure mindset etc., not analyzed in the present study). The entire experiment took no more than 2 hr.

Evaluation of the CT-Tutor: Results

As stated above, the broad goal of our work was to investigate the impact of different types of assistance in the CT-Tutor on learning and performance. Thus, in our analysis, we had two categories of outcome measures: learning and performance in the tutor. To operationalize learning, we used the standard approach of calculating the gain from pre-test to post-test (post-test – pre-test).[1] We had two types of questions on the test: code tracing and plain explanation. Our primary interest was in the code tracing outcomes because the interface scaffolding component focused on this, but for the sake of completeness, we also analyzed learning related to the plain-English explanations for the primary analyses.

We operationalized performance through behaviors in the CT-Tutor, based on information extracted from the log files.[2] We focused on a set of features prior work has found informative (e.g., Baker et al. 2009; Gobert et al. 2015; Sao Pedro et al. 2013), including:

- **Number of problems finished:** the number of problems correctly solved (recall that students could choose to leave a problem before finishing it after three minutes had passed).
- **Number of attempts:** number of attempts on average per problem needed to produce the correct answer for a given step (this measure was normalized by taking the total number of attempts and dividing by the number of text boxes that had to be used to enter the solution, to account for the fact that the high-scaffolding CT-Tutor included more entry boxes per problem than the reduced-scaffolding tutor).
- **Problem time:** average time spent on a problem (calculated by dividing total time spent on the problems by the number of problems attempted).
- **Time on correct entries:** average median time per problem spent generating correct entries, calculated by obtaining the median time spent on correct entries for a given problem, and then finding the average value by dividing by the number of problems attempted. Median was used rather than mean because the logger failed to record when participants were writing in the explanation area for the two corresponding conditions in the high-scaffolding interface. Thus, for these two conditions, the explanation time was lumped together with the subsequent step time, which could inflate the overall step time. By using the median step time, this should exclude the explanation time as participants tended to produce the explanation as a "one-shot-deal" at the end, based on a review of the log files; however,

[1] Alternative ways of assessing learning include using normalized gain scores and using ANCOVA with pretest as the covariate (Bonate 2000). While the former method is not advocated (Bonate 2000), the ANCOVA is a good alternative. For the sake of completeness we re-ran the analysis for learning for each of these two alternatives and the same pattern of results emerged.

[2] Two log files were lost

since we cannot guarantee that some may not have, the *time on correct* measure needs to be interpreted with caution.

- **Time on incorrect entries:** average median time per problem spent on incorrect entries, calculated using the method for correct entries described above.
- **Example time:** average time spent on an example (calculated by dividing total time spent on the examples by the number of examples opened).

We present the results at multiple levels of granularity, starting with aggregate outcomes for all participants, followed by outcomes for participants who gained from pre-test to post-test, followed by groups of participants identified by cluster analysis. For the first two sets of analyses, we used null hypothesis significance testing – since our study design had two factors, *interface scaffolding* and *instructional order*, the primary analyses were run as two-way between subjects ANOVAs with these factors as the independent variables and the learning and performance measures as the dependent variables. We report results as significant when $p < .05$ and as marginal when p is between .05 and .10; we also report effect sizes. Outliers flagged as outside 3* interquartile range were not included in the analysis (so *df* might vary slightly). To check for normality of the data, we calculated the residuals for each dependent variable and obtained normality parameters (skewness and kurtosis) for each residual variable (see Appendix 2 – note that obtaining residuals within the context of the full model, as we did, accounts for the experimental conditions in the analysis). George and Mallery (2019) indicate that skewness and kurtosis values less than |1| can be considered excellent and less than |2| as acceptable – the majority of the normality values fell into the excellent category and the remaining fell into the acceptable category. In general, perfectly-normal distributions are not typical in real data samples (Blanca et al. 2013) and the ANOVA is robust to violations of normality (Blanca et al. 2017).

Effect of CT-Tutor on Learning and Performance (All Participants)

We begin with the results for all participants, regardless of whether they learned from their interaction with the CT-Tutor, followed by analysis for participants who did learn. Learning is operationalized by our primary learning variable, namely code tracing gains (below, unless otherwise stated, when we refer to learning outcomes, we mean ones for code-tracing). The interaction between *interface scaffolding* and *instructional order* was not significant for any of the dependent variables. Thus, we focus on reporting the main effects for each factor, starting with the results for interface scaffolding.

Effect of Interface Scaffolding

Learning Descriptives for the effect of interface scaffolding on learning are in Table 1; the learning gains in each condition are also shown graphically in Fig. 6. Overall, collapsed across conditions, participants improved significantly from pre-test to post-test, $t(96) = 9.8$, $p < .001$. However, in contrast to our hypothesis (H1), participants learned marginally less from interacting with the high-scaffolding version of the CT-Tutor as compared to the reduced-scaffolding version, $F(1, 93) = 3.4$, $p = .069$, $\eta_p^2 = .04$. This result was not biased by a priori differences between groups (i.e., as expected given the assignment method, there were no differences between the four conditions on

Table 1 Descriptives (mean and SD) for the *code tracing* pre-test, post-test, and pre- to post-test gains shown as percentages for each condition as well as marginal means for each factor collapsing by the other factor (all participants included in the analysis)

		Reduced Scaffolding	*High Scaffolding*	*Marginal (instructional order)*
pre- test % $N = 97$	*Problem-first*	38.4% (16.6), $n = 28$	34.3% (23.2), $n = 28$	36.3% (20.6), $n = 56$
	Example-first	33.5% (20.1), $n = 17$	34.0% (15.5), $n = 24$	33.8% (17.3), $n = 41$
	Marginal (scaffolding)	36.6% (17.9), $n = 45$	34.1% (19.8), $n = 52$	
post-test % $N = 97$	*Problem-first*	57.1% (23.6), $n = 28$	50.0% (30.6), $n = 28$	53.6% (27.3), $n = 56$
	Example-first	58.5% (26.0), $n = 17$	48.3% (20.2), $n = 24$	52.6% (23.0), $n = 41$
	Marginal (scaffolding)	57.7% (24.2), $n = 45$	49.2% (26.1), $n = 52$	
gain % $N = 97$	*Problem-first*	18.8% (15.7), $n = 28$	15.7% (21.5), $n = 28$	17.2% (18.7), $n = 56$
	Example-first	25.0% (20.9)), $n = 17$	14.4% (12.9), $n = 24$	18.8% (17.3), $n = 41$
	Marginal (scaffolding)	**21.1% (17.9)**, $n = 45$	**15.1% (17.8**), $n = 52$	

Effect of scaffolding for gain scores shown in bold

code tracing pre-test scores, verified with an ANOVA on the pre-test scores). One potential explanation for this result is that the high-scaffolding CT-Tutor was less effective at promoting constructive behaviors needed for learning as compared to the reduced scaffolding interface. We have some evidence from the log files for this proposal that we now outline.

First, as shown in Table 2 (bottom row), the reduced-scaffolding group spent more time attending to the examples, $F(1, 87) = 7.5$, $p = .008$, $\eta_p^2 = .07$. Time spent on examples could be indicative of constructive behaviors like self-explanation of the example solution. While we do not have evidence that students were self-explaining, self-explanation from examples takes time (Muldner and Conati 2010), and so there is indirect evidence of self-explanation occurring more in the reduced-scaffolding group. In addition to the self-explanation in the context of an example, self-explanation can also take place in the context of solving a problem (Aleven and Koedinger 2002; Loibl et al. 2017. Since the reduced-scaffolding interface did provide less assistance, it may

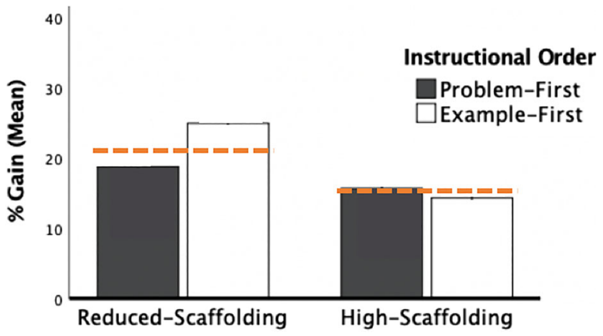


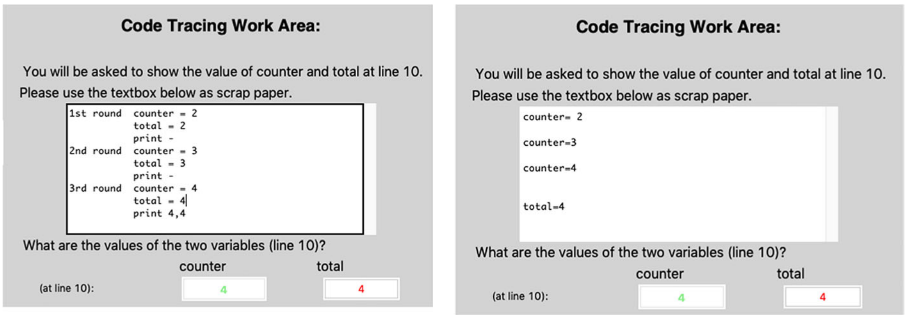
Fig. 6 Graphical depiction of code tracing pre- to post-test gains; dashed line shows main effect of scaffolding (all participants included in the analysis)

Table 2 Descriptives (mean and SD) for the performance measures for each condition as well as marginal means for each factor collapsing by the other factor (all participants included in the analysis)

		Reduced Scaffolding	*High Scaffolding*	*Marginal (instructional order)*
# problems finished	*Problem-first*	1.9 (1.4), $n = 28$	2.4 (1.3), $n = 27$	2.2 (1.4), $n = 55$
	Example-first	2.4 (1.6), $n = 17$	2.8 (1.3), $n = 23$	2.7 (1.4), $n = 40$
	Marginal (scaffolding)	2.1 (1.5), $n = 45$	2.62 (1.3), $n = 50$	
# attempts	*Problem-first*	1.8 (0.9), $n = 26$	1.5 (0.4), $n = 26$	1.7 (0.7), $n = 55$
	Example-first	1.5 (0.7), $n = 15$	1.3 (0.3), $n = 21$	1.4 (0.5), $n = 38$
	Marginal (scaffolding)	1.7 (0.8), $n = 43$	1.4 (0.4), $n = 50$	
time on problem (sec)	*Problem-first*	326.9 (100.3), $n = 28$	353.3 (102.6), $n = 27$	339.9 (101.4), $n = 55$
	Example-first	241.5 (136.8), $n = 17$	290.4 (115.4), $n = 23$	269.6 (125.6), $n = 40$
	Marginal (scaffolding)	294.7 (121.4), $n = 45$	324.4 (112.1), $n = 50$	
time on correct (sec)	*Problem-first*	4.3 (1.8), $n = 24$	9.26 (3.4), $n = 27$	6.91 (3.7), $n = 51$
	Example-first	4.1 (1.8), $n = 14$	8.17 (3.5), $n = 23$	6.64 (3.6), $n = 37$
	Marginal (scaffolding)	4.2 (1.8), $n = 38$	8.76 (3.5), $n = 50$	
time on incorrect (sec)	*Problem-first*	6.27 (3.6), $n = 22$	12.0 (6.2), $n = 27$	9.5 (5.9), $n = 49$
	Example-first	7.1 (4.0), $n = 12$	10.0 (6.9), $n = 21$	8.9 (6.1), $n = 33$
	Marginal (scaffolding)	6.6 (3.7), $n = 34$	11.1 (6.6), $n = 48$	
time on example (sec)	*Problem-first*	96.8 (25.3), $n = 28$	84.98 (20.6), $n = 27$	91.2 (23.7), $n = 51$
	Example-first	136.2 (31.9), $n = 17$	118.2 (25.6), $n = 23$	125.8 (29.4), $n = 40$
	Marginal (scaffolding)	112.0 (33.8), $n = 44$	101.2 (28.4), $n = 47$	

have encouraged more constructive behaviors like self-explanation, because it put more responsibility on the student to generate the solution.

A second piece of evidence regarding constructive behaviors comes from the free-form text box in the interfaces (used to generate explanations and take notes, see Fig. 7 and Fig. 8). While participants in both interface versions had a textbox for taking notes, they used these textboxes quite differently. In the high-scaffolding interface, participants tended to follow the instructions to provide a high-level explanation of what the program did (see Fig. 8) and did not use this area for additional code tracing notes (this is not surprising since the code tracing table in this interface required them to code trace step-by-step in the provided table). In contrast, in the reduced-scaffolding interface, participants were not given scaffolding from the tutor and so they used this textbox to code trace (see Fig. 7) and to make explicit how variables changed in the program. By sketching how these values changed, reduced-scaffolding participants were constructive because they produced traces that went well over and beyond the instructional materials (Chi and Wylie 2014). This construction may have boosted their code tracing scores on the post-test. In general, however, as illustrated in Figs. 7 and 8, in both interfaces there was variability in terms of how effectively participants used the textboxes, highlighting that some students did not use these tools as well as others.


Code Tracing Work Area:

You will be asked to show the value of counter and total at line 10.
Please use the textbox below as scrap paper.

```
1st round  counter = 2
           total = 2
           print -
2nd round  counter = 3
           total = 3
           print -
3rd round  counter = 4
           total = 4
           print 4,4
```

What are the values of the two variables (line 10)?

	counter	total
(at line 10):	4	4

Code Tracing Work Area:

You will be asked to show the value of counter and total at line 10.
Please use the textbox below as scrap paper.

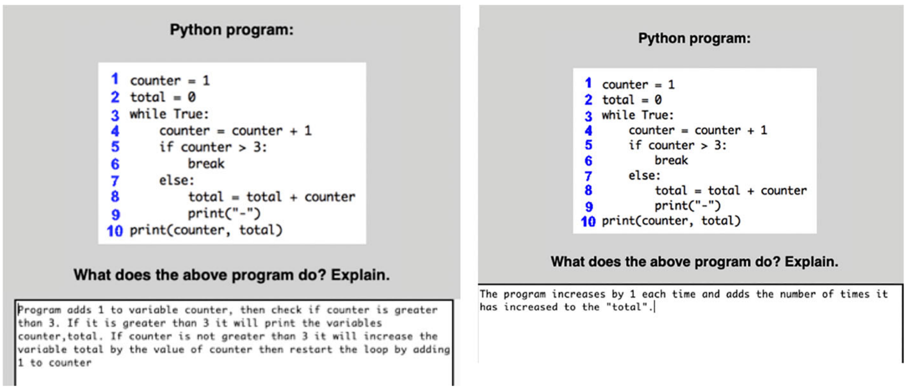
```
counter= 2

counter=3

counter=4


total=4
```

What are the values of the two variables (line 10)?

	counter	total
(at line 10):	4	4

Fig. 7 Two examples of participants' code traces in the reduced scaffolding interface, with the example on the left showing more constructive tracing than the example on the right (green = correct, red = incorrect entries)

Performance Descriptives for the effect of interface scaffolding on the target performance measures are in Table 2. The high-scaffolding interface improved correctness of entries compared to the reduced-scaffolding version. Specifically, the high-scaffolding group needed significantly fewer attempts to produce the required answer for a given text box in the interface, $F(1, 84) = 4.0$, $p = .048$, $\eta_p^2 = .05$. We predicted that this group would finish more problems correctly (H2), and while descriptively this was the case, this trend did not reach significance, $F(1, 91) = 2.6$, $p = .11$, $\eta_p^2 = .03$. Contrary to our prediction (H3), high scaffolding did not, however, improve performance in terms of time. The high-scaffolding group spent longer than the reduced-scaffolding group on the problem overall, $F(1, 91) = 2.6$, $p = .11$, $\eta_p^2 = .028$; this was also the case for correct and incorrect entries (correct entries: $F(1, 84) = 51.0$, $p < .001$, $\eta_p^2 = .4$; incorrect entries, $F(1, 78) = 11.4$, $p = .001$, $\eta_p^2 = .13$). We acknowledge it is challenging to compare the results for the number of attempts, and time on correct and incorrect entries given that there were more entry boxes in the high-scaffolding condition than the reduced-scaffolding condition (see Fig. 2 vs. Figure 3). To attempt to account for this difference, as noted above, we normalized these variables by dividing the number of attempts and time spent by the number of entries to achieve an average time for each entry box.


Python program:

```
1  counter = 1
2  total = 0
3  while True:
4      counter = counter + 1
5      if counter > 3:
6          break
7      else:
8          total = total + counter
9          print("-")
10 print(counter, total)
```

What does the above program do? Explain.

```
Program adds 1 to variable counter, then check if counter is greater
than 3. If it is greater than 3 it will print the variables
counter,total. If counter is not greater than 3 it will increase the
variable total by the value of counter then restart the loop by adding
1 to counter
```

Python program:

```
1  counter = 1
2  total = 0
3  while True:
4      counter = counter + 1
5      if counter > 3:
6          break
7      else:
8          total = total + counter
9          print("-")
10 print(counter, total)
```

What does the above program do? Explain.

```
The program increases by 1 each time and adds the number of times it
has increased to the "total".
```

Fig. 8 Two examples of participants' explanations in the high scaffolding interface, with the example on the left showing a better explanation than the one on the right

Effect of Instructional Order (Problem-First vs. Example First)

Above, we presented results related to the main effect of interface scaffolding – we now report on instructional order (descriptives are in Table 1 and Table 2). Recall that the example either came first in the instructional order (and was followed by the problem), or came second (and so was preceded by the problem); the example-first order provides more guidance than the problem-first order, because in the former, participants were able to study the example prior to solving the problem. We did not find evidence that instructional ordering affected learning, $F(1, 93) = .4$, $p = .511$, $\eta_p^2 < .01$. Ordering did, however, affect performance, as follows. The example-first group spent significantly more time on average per example, $F(1, 87) = 44.5$, $p < .001$, $\eta_p^2 = .34$, and, as predicted (H3), less time on average per problem, $F(1, 91) = 10.1$, $p = .002$, $\eta_p^2 = .02$. While as hypothesized the example-first group did finish more problems correctly, this did not reach significance, $F(1, 91) = 2.2$, $p = .138$, $\eta_p^2 = .02$. As far as the number of attempts, the example-first group required fewer attempts to produce a correct answer, $F(1, 84) = 5.1$, $p = .027$, $\eta_p^2 = .06$. Thus, as hypothesized, there is some evidence that having the example first in the instructional order improved performance, but not learning (as there were no learning differences between the two instructional orders).

Explanation-Related Results

While code tracing scores are the primary learning variable of interest, for the sake of completeness we now report on the explanation-related outcomes (recall there were two questions on the test asking students to explain code using plain English). The explanations were graded using a rubric we developed assigning points to the key explanation components. To illustrate, in question 5 on the pre-test (see Appendix 1), the explanation was graded out of a total of 2 points using the following rubric: *the program loops around, each time (0.5); counter increases by 1 each iteration (0.5); the previous value of result is increased by the value of counter each iteration (0.5); the loop stops when the value of counter becomes bigger than 2 (0.5).*

The explanation results are in Table 3. The explanation scores did improve overall from pre-test to post-test but there were no significant effects of either level of scaffolding, $F(1, 93) = .5$, $p = .50$, $\eta_p^2 < .01$, or instructional order, $F(1, 93) = .3$, $p = .57$, $\eta_p^2 < .01$. As predicted, learning code-tracing gains and explanations were positively correlated, Pearson's $r(95) = .2$, $p = .044$, although the effect was weak.

Effect of CT-Tutor on Learning and Performance (Participants Who Gained From Pre-Test to Post-Test)

The above analyses included data from all participants, including ones who did not gain from pre-test to post-test or got worse (referred to as non-gainers below, $N = 23$). No participants were at ceiling at pre-test, so there are two reasons why students might not have learned: (1) they were not motivated to learn and/or complete the post-test to the best of their ability, and/or (2) the instructional materials were confusing. Since the majority of students did gain from pre- to post-test ($N = 74$), this suggests the instructional materials are not the culprit. However, it may be the case that certain conditions in the tutor were particularly unsuitable for some participants. There were twice as many non-gainers in the

Table 3 Mean (SD) *explanation* pre-test, post-test, and pre- to post-test gains shown as percentages for each condition as well as marginal means for each factor collapsing by the other factor (all participants included in the analysis)

		Reduced Scaffolding	*High Scaffolding*	*Marginal (instructional order)*
pre-test % $N = 97$	*Problem-first*	32.6% (27.7), $n = 28$	21.9% (26.3), $n = 28$	27.2% (27.3), $n = 56$
	Example-first	20.6% (24.2), $n = 17$	33.3% (30.8), $n = 24$	28.0% (28.6), $n = 41$
	Marginal (scaffolding)	28.1% (26.8), $n = 45$	27.2% (28.7), $n = 52$	
post-test % $N = 97$	*Problem-first*	52.2% (38.5), $n = 28$	44.6% (39.7), $n = 28$	48.44% (39.0), $n = 56$
	Example-first	44.1% (32.5), $n = 17$	43.8% (33.4), $n = 24$	43.9% (32.6), $n = 41$
	Marginal (scaffolding)	49.2% (36.2), $n = 45$	44.2% (36.6), $n = 52$	
gain % $N = 97$	*Problem-first*	19.6% (32.7), $n = 28$	22.8% (31.5), $n = 28$	21.2% (31.9), $n = 56$
	Example-first	23.5% (35.1), $n = 27$	10.4% (42.5), $n = 24$	15.9% (39.6), $n = 41$
	Marginal (scaffolding)	21.1% (33.3), $n = 45$	17.1% (37.1), $n = 52$	

high-scaffolding conditions ($N = 15$) than the reduced-scaffolding conditions ($N = 8$), which suggests the high-scaffolding interface was not suitable for this subgroup. Since the non-gainer sample size is too small to reliably run inferential stats on, we focus on participants who did gain from pre-test to post-test, referred to as *gainers* below.

Learning Table 4 shows the descriptives related to learning for the gainers; the gains in each condition are also shown graphically in Fig. 9. There were no a priori differences between the four conditions (verified with an ANOVA on the code tracing pre-test scores). Compared to the analysis that included all participants, the effect of condition on learning became more complicated for the gainers. Specifically, there was a significant interaction between interface scaffolding and instructional order, $F(1, 70) = 6.1$, $p = .016$, $\eta_p^2 = .08$.[3] Participants in the reduced-scaffolding condition learned more if they were in the example-first instructional order, whereas participants in the high-scaffolding condition learned more if they were in the problem-first instructional order (Fig. 9 illustrates this interaction). Thus, for the gainers, our hypothesis (H1) regarding learning was partially supported: scaffolding improved learning, but only under certain conditions.

A potential explanation for these results is that participants in the high-scaffolding condition who were given an example first before solving a problem may have had too much assistance, an argument similar to the one made above when describing the learning results that included all participants (not just ones who gained). In this case, assistance from examples and assistance from the scaffolding may have led to more passive engagement with the instructional materials. In particular, when it came time to solve the problem, the example-first high-scaffolding combination may have failed to encourage participants to apply what they learned from the example, as the scaffolding provided a lot of assistance to solve the problem. In contrast, participants in the example-first reduced-scaffolding needed to actively infer information from the example so they could solve the following corresponding problem, as that problem did not provide scaffolding. Participants in the example-

[3] Main effect of scaffolding and instructional order were not significant

Table 4 Descriptives (mean and SD) for the code tracing pre-test, post-test, and pre- to post-test gains shown as percentages for each condition as well as marginal means for each factor (collapsing by the other factor) (gainers only)

		Reduced Scaffolding	*High Scaffolding*	*Marginal (instructional order)*
pre-test % *N* = *74*	*Problem-first*	39.1% (17.4), *n* = 23	32.5% (17.1), *n* = 18	36.2% (17.4), *n* = 41
	Example-first	34.29% (17.1), *n* = 14	35.0% (16.8), *n* = 19	34.7% (16.6), *n* = 33
	Marginal (scaffolding)	37.3% (17.1), *n* = 37	33.8% (16.7), *n* = 37	
post-test % *N* = *74*	*Problem-first*	62.4% (22.0), *n* = 23	60.8% (24.5), *n* = 18	61.7% (22.8), *n* = 41
	Example-first	64.6% (20.0), *n* = 14	53.7% (18.6), *n* = 19	58.3% (19.7), *n* = 33
	Marginal (scaffolding)	63.2% (21.0), *n* = 37	57.2% (21.7), *n* = 37	
gain % *N* = *74*	*Problem-first*	**23.3% (13.4),** *n* = 23	**28.3% (15.1),** *n* = 18	25.5% (14.2), *n* = 41
	Example-first	**30.4% (19.1),** *n* = 14	**18.7% (10.7),** *n* = 19	23.6% (15.7), *n* = 33
	Marginal (scaffolding)	26.0% (15.9), *n* = 37	23.4% (13.7), *n* = 37	

Significant interaction between scaffolding and instructional order for the gain scores in bold

first reduced-scaffolding condition did indeed study the example the longest (see Table 5, bottom row) and had the highest learning gains on the post-test (see Table 4).[4]

Performance Overall, the pattern of results related to the performance measures was the same for the gainers (see Table 5) as for the analysis that included all the participants presented above. Like the analysis that included all participants, there was a main effect of instructional order for the same target variables: the high-scaffolding group needed significantly fewer attempts to produce the correct answer, $F(1, 64) = 2.7$, $p = .10$, $\eta_p^2 = .04$, and as predicted finished more problems correctly, $F(1, 69) = 4.2$, $p = .044$, $\eta_p^2 = .06$. Also like for the analysis including all participants, the example-first group spent significantly more time on average per example, $F(1, 67) = 41.8$, $p < .001$, $\eta_p^2 = .38$, and as predicted less time on average per problem, $F(1, 69) = 13.6$, $p < .001$, $\eta_p^2 = .16$). The only difference between the current analysis and the prior analysis that included all participants is that for the gainers, there was a significant interaction between the level of scaffolding and instructional order for the median time to produce correct answers, $F(1, 64) = 4.9$, $p = .031$, $\eta_p^2 = .07$, and incorrect answers, $F(1, 58) = 6.0$, $p = .018$, $\eta_p^2 = .09$. The pattern was the same for both correct and incorrect entries, with the high-scaffolding condition in the problem-first order having the longest response times and the reduced-scaffolding conditions taking approximately the same amount of time to produce the answers.

Clustering Based on Behaviours in CT-Tutor

The above analysis provided information on how assistance in the CT-Tutor affected performance and outcomes using aggregate statistics. An alternative analysis strategy

[4] As was the case for the analysis that included all the participants, conditional differences related to explanation gains were not significant and since they are not the focus, they are not reported here.

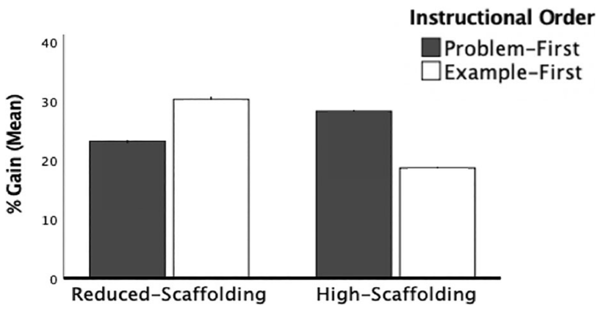


Fig. 9 Graphical depiction of code tracing pre- to post-test gains showing the interaction between interface scaffolding and instructional order (gainers only)

involves using unsupervised learning to cluster students according to behaviors within the tutor. Clustering was done in R using k-means, a basic but effective clustering algorithm commonly used to analyze educational data (Shovon and Haque 2012; Bhise et al. 2013).

This analysis was run on using data from all participants as we wanted to investigate how participants interacted with the tutor and whether certain combinations of behaviours were related to learning. As input to k-means we used the features we extracted from the log files shown in Table 2 and listed in the introduction to the Results section – these comprise student behaviors within CT-Tutor. We did not include test data or condition as features, because we wanted to have the tutor behaviors drive the cluster results and not bias them with test or conditional information.

As two log files were lost, the initial number of participants for this analysis was $N = 95$. Any missing data points were imputed with that variable's mean, allowing us to include all 95 participants in the analysis. Given that k-means is sensitive to outliers, we checked and removed them as follows. We performed an initial clustering, and then calculated the relative distance of data points to their respective clusters. We excluded from the data five outliers that were far removed from their respective clusters and one data point that occupied a single cluster (for a total of 6 removed data points).[5] This left us with 89 data points corresponding to 89 participants (including gainers and non-gainers). After removal of these cases, we re-ran k-means. We used the elbow method to determine the number of clusters (Kassambara 2017), which we set at $k = 4$. Due to k-means sensitivity to initial cluster positions, we executed k-means with 50 different random starting points and k-means subsequently selected the result with the lowest within-cluster variation.

Clustering Results

Table 6 shows the mean feature value for each cluster (top), and the code tracing and explanation learning outcomes for each cluster (bottom; note that these were not part of the features used to train the model and were calculated after clustering was complete). To analyze how much (or not) the clusters aligned with the four conditions, we also

[5] Four of the removed data points were from the reduced-scaffolding problem-first condition and two were from the high-scaffolding problem-first condition; one of the high-scaffolding problem-first participants was a non-gainer, while the other outliers were gainers.

Table 5 Descriptives (mean and SD) for the performance measures for each condition and marginal means for each factor collapsing by the other factor (gainers only)

		Reduced Scaffolding	*High Scaffolding*	*Marginal (instructional order)*
# problems finished	*Problem-first*	2.0 (1.4), *n* = 23	2.7 (1.3), *n* = 18	2.3 (1.4), *n* = 41
	Example-first	2.5 (1.6), *n* = 14	3.1 (1.2), *n* = 18	2.9 (1.4), *n* = 32
	Marginal (scaffolding)	2.2 (1.5), *n* = 37	2.9 (1.2), *n* = 36	
# attempts	*Problem-first*	1.9 (.8), *n* = 21	1.6 (.3), *n* = 18	1.7 (.6), *n* = 39
	Example-first	1.4 (.5), *n* = 13	1.3 (.3), *n* = 16	1.3 (.4), *n* = *29*
	Marginal (scaffolding)	1.7 (.7), *n* = 34	1.4 (.3), *n* = 34	
time on problem (sec)	*Problem-first*	324.1 (90.0), *n* = 23	364.6 (73.5), *n* = 18	341.9 (84.6), *n* = *41*
	Example-first	243.4 (138.1), *n* = 14	269.1 (102.2), *n* = 18	257.9 (117.9), *n* = *32*
	Marginal (scaffolding)	293.6 (115.8), *n* = *37*	316.8 (100.2), *n* = 36	
time on correct (sec)	*Problem-first*	4.4 (1.9), *n* = 20	10.1 (3.2), *n* = 18	7.1 (3.9), *n* = 38
	Example-first	4.2 (1.8), *n* = 12	7.3 (2.5), *n* = 18	6.0 (2.7), *n* = 30
	Marginal (scaffolding)	4.31 (1.86), *n* = 32	8.68 (3.2), *n* = 36	
time on incorrect (sec)	*Problem-first*	5.8 (2.7), *n* = 18	12.6 (6.1), *n* = 18	9.2 (5.8), *n* = 36
	Example-first	6.8 (3.8), *n* = 10	7.9 (3.9), *n* = 16	7.5 (3.8), *n* = 26
	Marginal (scaffolding)	6.18 (3.1), *n* = 28	10.38 (5.7), *n* = 34	
time on example (sec)	*Problem-first*	97.5 (24.7), *n* = 22	85.7 (22.0), *n* = 17	92.3 (24.0), *n* = 39
	Example-first	138.4 (33.3), *n* = 14	125.0 (23.9), *n* = 18	130.9 (28.7), *n* = 32
	Marginal (scaffolding)	113.4 (34.5), *n* = 36	105.9 (30.2), *n* = 35	

extracted the number of participants in each condition, see Fig. 10 (note that condition and learning were not features used during training of the model). We now present the results of our analysis. We focus on descriptives in our presentation but also include inferential statistics to highlight key results inferential statistics were conducted using t-tests between cluster groups; if homogeneity of variance was detected, corrected values are reported.

In general, cluster 4 had the lowest overall code-tracing learning gains, while learning related to code tracing was similar between clusters 1–3; clusters 4 and 3 each had a similar number of individuals from each condition. We labelled cluster 4 as the *struggling unproductively* cluster. Overall, participants in this cluster finished the fewest number of problems ($p = .10$); half to four times fewer than the other clusters, see Table 6, and their code-tracing gains were lower than the other clusters ($p < .10$). Participants in this cluster may have been uncomfortable making mistakes. Evidence for this conjecture comes from the high time on both correct and incorrect entries, substantially higher than for the other clusters ($p < .05$). This suggests cluster 4 participants were putting in effort (also evidenced by the time spent on examples, which was higher than for the other clusters, significantly more so than clusters 2 and 3, $p < .05$), but this effort was not paying off.

We characterized individuals in Cluster 3 as *productive*, based on the fact that they finished more problems than the other clusters ($p < .001$), spent the least amount of time to generate correct entries (significantly faster than cluster 4, $p = .003$, and cluster 1, $p < .001$), and needed fewer attempts to get the correct answer. Of note is that the cluster 3 pre-test scores were highest overall. While the scores were still relatively low at 50%, this may have given a boost to cluster 3 participants. The pre-test was given after the instructional materials were read, and so one possibility is that cluster 3 individuals attended more to these materials and so learned more from them. An alternative possibility is that they had higher knowledge coming in, because, for instance, they programmed in high school. As we mentioned above, we did not include pre-test scores (nor any learning outcomes) as features used to train k-means, and so the model picked up on the higher knowledge of this group solely based on their behaviors in the tutor.

The other two clusters, cluster 2 and 1, did reflect some conditional effects in that they had substantially more individuals from one condition. We labelled cluster 2 as the *productively struggling* cluster. Individuals in this cluster needed the most attempts to produce a correct entry (about twice as many as the other clusters, $p < .001$) and compared to cluster 3, finished about half as many problems, and spent longer on a given problem. This cluster contained five times as many individuals in the reduced-scaffolding problem-first group as the other three conditions. As mentioned above, the reduced-scaffolding problem-first condition was the lowest assistance condition, thus explaining why participants may have struggled while working with this version of CT-Tutor. However, we labelled this struggle as productive because cluster 2 had the highest learning gains (both for code tracing and explanation, at least from a descriptive standpoint), despite having low pre-test scores, see Table 6, bottom. This is an example of how clustering can shed further light on the effect of assistance. Recall that the aggregate statistics above indicated that moderate assistance was beneficial for learning (i.e., the reduced-scaffolding example-first and high-scaffolding problem-first conditions).

Table 6 Mean feature values for each cluster (SD shown in brackets, top, and mean gains from pre-test to post-test (followed by pre-test and post-test scores, respectively, shown in brackets), bottom

Features	Cluster 1 *n* = 34	Cluster 2 *n* = 16	Cluster 3 *n* = 31	Cluster 4 *n* = 8
# problems finished	1.7 (1.1)	2.19 (1.1)	3.74 (.5)	.9 (1.0)
# attempts	1.4 (.3)	2.75 (.4)	1.28 (.3)	1.2 (.2)
time on problem	354.6 (111.9)	322.7 (101.8)	232.3 (90.8)	357.9 (102.5)
time on correct entries	14.0 (7.5)	9.0 (5.3)	8.0 (4.3)	32.2 (15.6)
time on incorrect entries	18.2 (6.8)	11.5 (7.2)	14.70(8.1)	48.2 (11.9)
time on example	112.8 (44.7)	93.90 (24.1)	110.2 (31.5)	142.4 (52.5)
code tracing % gains (pre-test%, post-test%)	18.8% (28.1%, 46.9%)	22.5% (27.2%, 49.7%)	19.5% (50.0%, 69.5%)	9.4% (25.6%, 35.0%)
explanation % gains (pre-test%, post-test%)	13.3% (21.8%, 35.0%)	32.8% (14.0%, 47.0%)	23.5% (41.3%, 64.5%)	7.8% (.22.0%, 29.8%)

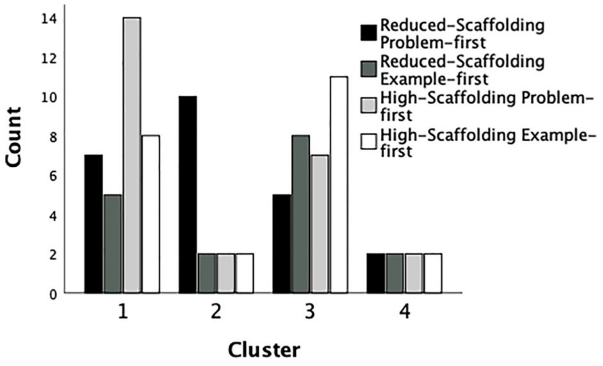


Fig. 10 Number of participants in each condition per cluster (conditions are shown ordered from lowest assistance to highest assistance)

The clustering analysis shows that some individuals can succeed at learning the materials with low assistance, and indeed benefit from it if they persevere.

We characterized cluster 1 individuals as *dedicated problem solvers*. Participants in this cluster spent more time generating entries than clusters 2 and 3 and accordingly had higher problem time. Their code-tracing learning gains are very similar to cluster 3 and slightly lower than cluster 2. This cluster has almost twice as many high-scaffolding problem-first participants in it as compared to the other three conditions. Thus, this cluster reflects at least in part the aggregate results that some assistance is beneficial, at least for code tracing (this cluster did particularly poorly on the explanation outcomes).

Discussion

In this paper we described the design and evaluation of the CT-Tutor. The tutor was designed based on prior research on how novice programmers trace programs using paper and pencil activities (Cunningham et al. 2017) and provided assistance in two different ways: through worked examples and interface scaffolding. We implemented four versions of the tutor to test four conditions corresponding to different ways of combining assistance. The highest form of assistance was in the high-scaffolding example-first condition, while the lowest was in the reduced-scaffolding problem-first. The other two conditions fell in the middle of the assistance spectrum as they each provided one type of assistance (either through high scaffolding in the high-scaffolding problem-first condition, or through an example presented before each problem in the reduced-scaffolding example-first condition).

How much assistance was optimal? The answer depends on the outcome variable (performance vs. learning) and individual differences in how participants interacted with the tutor. The results using null hypothesis significance testing based on the data from all the participants, regardless of whether they learned or not, showed that both types of assistance (high-interface scaffolding, example first instructional order) increased the number of problems solved and time needed to solve them. However, for some combinations of assistance, this came at the cost of learning. For the analysis considering all participants, we did not find evidence of instructional order effects on learning, but we did find that high-interface scaffolding resulted in marginally *less* learning than reduced scaffolding. Thus, our hypothesis that the high-scaffolding group would learn more was not supported. We provided a number of explanations for this unanticipated finding above, centered around the proposal that high-interface scaffolding made students less constructive because it provided too much assistance. However, it would be premature to conclude that the interface scaffolding was detrimental to all participants as this result was influenced by a subset of participants we referred to as non-gainers (individuals who did not gain from pre-test to post-test). The participants who did not gain fared especially poorly when given interface scaffolding: there were twice as many non-gainers in the high-scaffolding condition ($N = 15$) as in the reduced-scaffolding condition ($N = 8$). One explanation for this result is that the scaffolding may have facilitated shallow behaviors like guessing and checking, a form of gaming (Baker et al. 2009). There are many reasons why students game (Baker et al. 2010; Muldner et al. 2011; Baker et al. 2009), some of which are motivational (Baker et al. 2008). The non-gainers may have been particularly unmotivated, as suggested by the fact they did not learn from the tutor (and many got worse from pre-test to post-test), and that may explain why the high scaffolding interface was not suitable for them. When these individuals were removed from analysis, a more nuanced story emerged on the effect of assistance.

For participants who gained from pre-test to post-test, learning was maximized when the tutor provided one type of assistance (example-first or high-scaffolding). If both types of assistance were provided, or if neither was provided, then learning was reduced. Thus, our results suggest that novice students benefit from a specific level of assistance when learning how to code trace, neither too much, nor too little. While the "too little" aspect is unsurprising, there is precedent for the "too much" aspect, i.e., that too much assistance is not helpful. Earlier work in the context of human tutoring proposed that high assistance from tutors, for instance in the form of explanations, may supress tutee learning because it reduces opportunities for tutees to make constructive responses (Chi et al. 2001). To test this conjecture, Chi et al. (2001) compared student learning from a high-assistance condition where tutors offered many explanations to one where tutors were instructed to prompt instead (thus offering reduced assistance as the onus was on the students to generate the answer). While there was no significant difference between the two groups, possibly due to power issues, there was some promising indications that reduced assistance did promote better learning. More recently, in a motor learning task, high assistance during the learning phase resulted in poorer performance than when assistance was faded out (Tullis et al. 2015). As a second example, Lee et al. (2015) tested the effect of problem-example similarity in a problem solving task where examples were used as assistance. Examples that were very similar to the problem (offered high assistance) resulted in less learning than lower similarity examples (which offered reduced assistance). Our recent work with an algebra CTAT tutor echoed this result (Borracci et al. 2020). In general, prior work has shown that help in tutoring systems is beneficial, but "help helps only so much" (Aleven et al. 2016), highlighting the key role the student has in processing the assistance provided.

To recap, the analysis using aggregate methods indicated that on average, some assistance is better than no assistance or too much assistance. An even more detailed story emerged, however, when we clustered participants based on their behaviors in CT-tutor, using features extracted from the log files. The cluster we labelled as the productively-struggling group had the highest learning gains and had more reduced-scaffolding problem-first participants, suggesting that low assistance was beneficial for some, if they persevered (this cluster had a greater number of attempts to solve the problem that eventually resulted in a correct response). We also found that the lowest learning group, labelled the struggling unproductively cluster, had participants evenly spread across conditions, suggesting that low learning gains were not always the result of the experimental manipulations (level of scaffolding, instructional order). The clustering analysis also identified a group we labelled as the productive group due to their performance within the tutor. These participants had the highest pre-test scores (which were not included in the clustering algorithm), highlighting the ability of clustering to identify students with higher prior knowledge based solely on their behaviours within the tutor.

Limitations, Opportunities for Adaptation, and Implications

This paper describes only the first step in a long-term project centered around the design and evaluation of support for the skill of code tracing. The goal of the present work was to gain preliminary insight into the design of the CT-Tutor. While we met this goal, we acknowledge that the relatively modest sample size in the evaluation means further work and replication is needed. Another limitation is that the CT-Tutor prototype does not currently personalize its assistance, because we wanted to evaluate the initial prototype to understand what kind of adaptation would be beneficial. This iterative design approach is suitable given there is very little work on supporting code tracing and so it is an open question of how to best scaffold this activity. Our results highlight a number of opportunities to incorporate adaptation, which we now describe.

Interface Scaffolding

One opportunity for personalization relates to the design of the code-tracing interface (e.g., see Figs. 2 and 3). The interface was inspired by our experiences in the classroom as well as prior work showing that students do not code trace effectively and so we wanted to see if explicitly scaffolding that process would help. We did not find strong evidence that explicit scaffolding in the CT-Tutor was beneficial (although if we consider students who gained, it did promote learning, but only if the example followed the problem). Moreover, this scaffolding was particularly problematic for some individuals, namely ones who did not gain from pre- to post-test – we speculated this was because they were not motivated to learn, and this interface may have facilitated shallow processing. These observations open up opportunities for future work, related to the design of code-tracing scaffolding that does promote constructive behaviors. One possibility is to keep the scaffolding but incorporate prompts for self-explanation during problem solving. Prior work has found self-explanation prompts do increase learning (Aleven and Koedinger 2002). Another possibility is to fade the scaffolding (start high, transition to low). Faded worked-examples have been shown to be effective in conjunction with tutored problem solving (Salden et al. 2009).

An alternative solution pertains to removing scaffolding altogether, giving students a free-form area to code trace in any manner they like. This is already the case for the reduced-scaffolding interface in the CT-Tutor, but currently there is no feedback provided on the typed entries. As Figs. 7 and 8 highlight, some students were more constructive in their use of this interface than others. Thus, a future step could involve adaptively providing feedback on the code traces using natural language processing of the typed code traces. Natural language processing is already used in some tutoring systems to parse student written entries (Graesser 2016) although to date it has not been applied to the type of text that free-form code tracing produces (see Fig. 7 – the content here looks like pseudocode). This type of functionality would also be beneficial for assessing students' high-level explanations of what a program does. In addition to potentially making students more constructive, removing the scaffolding may increase student agency and in turn student motivation (Wang et al. 2019).

Example Design

In addition to including adaptive prompting for self-explanation during the solution of a code tracing problem, another opportunity for self-explanation prompts comes in the context of the worked example of a code trace. While we found that participants did read the examples provided by CT-Tutor as evidenced by time on task and took longer doing so if the interface scaffolding was reduced, the overall reading time was not high (less than two minutes on average). This suggests that students may need encouragement to self-explain the worked-out solution in the example. There are various successful examples of self-explanation prompts in intelligent tutors (Conati and Vanlehn 2000), but to date none in the domain of code tracing. To give us insight for what self-explanation might look like during code tracing, we are currently in the process of designing a talk aloud study where students self-explain worked examples of code traces, which will help to identify what types of prompts would be beneficial. Additionally, another way that students could be encouraged to study the examples effectively is by making the student an active participant in the example presentation, for instance, by including examples with errors that students need to identify and correct. Research on erroneous examples in an algebra tutor showed that the examples improved student learning above that of problem solving alone (Adams et al. 2014). Incorporating methods that increase participant interaction and processing of the example are valuable areas to target in the next iterations of the tutor.

Initial Lesson

Our instructional materials included a tutorial on code tracing. This tutorial came in the form of a short PowerPoint slideshow. While it was aimed to provide a brief introduction to code tracing, we acknowledge that learning to code-trace often takes place over a semester(s) in classrooms and labs where students frequently interact with instructors and teaching assistants. Therefore, the effect of the CT-Tutor on students' code tracing skills may be different in a classroom context, due to the ongoing human interaction between the educators and the students. Incorporating the tutor into standard classrooms is part of our long-term goal.

Adaptive Sequencing

Our results show that instructional order affected learning (if we consider the students who did gain from pre-test to post-test). In the CT-Tutor, however, the instructional order was fixed within a given condition (e.g., example always came first and was followed by a code tracing problem in the two example-first conditions). Recently, Najar et al. (2016) tested the effect of adapting the type of assistance in a tutor. Specifically, participants were given either a problem to solve, an example to study, or a faded example that had some but not all steps provided, in the domain of data base query knowledge. Each participant saw a series of these items (i.e., problems, examples, faded examples). The decision on which item to present was based on the student model's assessment of cognitive efficiency, which was derived from a self-reported measure of mental effort and performance in the tutor. The group with the adapted sequencing did better than the control group that had a fixed sequence consisting of an example always followed by a problem. Our study included conditions with different combinations of assistance, but the assistance was fixed. Thus, in the future we plan to add adaptivity related to instructional sequencing to see how it affects the learning of code tracing. Our finding that on average, too much assistance is not optimal suggests that the adaptation strategy should take this into account (e.g., if a student saw an example, providing a problem in the high-scaffolding interface may not be a good idea).

It is an open question if and how much adding these functionalities will improve learning related to code-tracing activities, but we are cautiously optimistic. There is certainly room for improvement – while there were significant gains from pre-test to post-test, the average post-test score hovered around 55%. While it may be tempting to attribute this to poor tutor design, the very reason we started this project is because of our teaching experiences in a first-year programming classroom. Test scores on code tracing questions are notoriously lower than on code writing questions in our introductory programming class – with many students scoring below 50%. The hope is that by using a data-driven approach to adapt to student behaviours within the tutor, we can improve learning more.

Implications

To date, very little work exists on how to support the process of code tracing for novices first learning to program. In this paper, we described a prototype for the Code Tracing-Tutor, as well as an evaluation of four versions of the tutor that varied levels of assistance. The key implications of our work include: (1) considering all individuals, learning was higher when interface scaffolding was reduced, (2) considering only the subset of individuals who improved on their code-tracing scores, moderate assistance was best (either reduced-scaffolding example-first or high-scaffolding problem-first), as compared with too much and too little assistance (i.e., high-scaffolding example-first and reduced-scaffolding problem-first), (3) the difference between points (1) and (2) suggests a potential future direction, namely personalized assistance, as the difference in outcomes between (1) and (2) was driven by individuals who did not gain from their interaction with the tutor. Further individual differences emerged from the unsupervised clustering, highlighting, for example, that some individuals do fine even in low assistance conditions; (4) code tracing is challenging for students and more work is needed to help them do it effectively.

Acknowledgements This research was funded by an NSERC Discovery grant and an Ontario Graduate Scholarship. We thank Julia Donahue and Emma Pavan, who did part of the data collection on the project.

Appendix 1: Pre-test

Participant ID: ______________________ Date: ___

Please answer the questions on the front and back of these. If you don't know, put IDK. For each of the following short programs, given the program (left) show what is printed (right) [1 point each for questions 1-4]:

1	`result = 1`
	`result = result + 10`
	`print( result )`
		What does the program print?

2	`char = "K"`
	`letter = input("Enter a letter: ")`
	`res = letter + char`
	`print ( res )`
		User types **Z** - what does the program print?

3	`answer = input("Enter name: ")`
	`if answer == "Bob":`
	`    print(" hello ")`
	`else:`
	`    print(" hi ")`
		User types **Bob**. What does the program print?

4	`counter = 1`
	`while True:`
	`    if counter > 2:`
	`        break`
	`    else:`
	`        print(" hi ")`
	`    counter = counter + 1`
		What does the program print?

5 **Given this program:**

```
counter = 1
result = 0
while True:
    counter = counter + 1
    if counter > 2:
        break
    else:
        result = result + counter
print(counter, result)
```

What does the program print? Please <u>show your work</u>. [2 points]

What is a high level plain English explanation of the above program? [2 points]

6 **Given this program:**

```
input1 = "-"
final = "$"
while True:
    input1 = input('next letter: ')
    final = input1 + final
    if input1 == "z":
        break
    else:
      print("hi")
print( final )
```

Suppose the user enters:

k
a
t
z

In the very last line (see `print(final))`, **what does the program print ? <u>Show your work</u>. [2 points]**

What is a high level plain English explanation of this program? [2 points]

7 **Given this program:**

```
val = "9"
counter = 1
result = "0"
while True:
    user_input = input('guess the value?')
    if user_input == val:
        print( 'good!' )
        break
    else:
        counter = counter + 1
    result = result + user_input
print( counter, result )
```

Suppose the user enters:

7
3
9

In the very last line (`print (counter, result)`), what does the above program print? Please show your work. [2 points]

Appendix 2

Table 7 Skew and kurtosis for primary variables

	All Participants (Skew, Kurtosis)	Gainers (Skew, Kurtosis)
# problems finished	−0.24, −1.0	−.30, −.99
# attempts	1.5, 1.9	.53, .11
time on problem (s)	.74, .42	.9, 1.3
time on correct (s)	.84, 1.1	.6, .04
time on incorrect (s)	1.1, 1.3	.83, .24
time on example (s)	.63, .25	.58, .37
learning (pure gain)	.33, −.25	.57, .28

References

Adams, D. M., McLaren, B. M., Durkin, K., Mayer, R. E., Rittle-Johnson, B., Isotani, S., & Van Velsen, M. (2014). Using erroneous examples to improve mathematics learning with a web-based tutoring system. *Computers in Human Behavior, 36*, 401–411.

Aleven, V., & Koedinger, K. (2002). An effective metacognitive strategy: Learning by doing and explaining with a computer-based cognitive tutor. *Cognitive Science, 26*, 147–179.

Aleven, V., Roll, I., McLaren, B. M., & Koedinger, K. R. (2016). Help helps, but only so much: Research on help seeking with intelligent tutoring systems. *International Journal of Artificial Intelligence in Education, 26*(1), 205–223.

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP tutor. *Cognitive Science, 13*(4), 467–505.

Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive tutors: Lessons learned. *The Journal of the Learning Sciences, 4*(2), 167–207.

Baker, R. S., D'Mello, S. K., Rodrigo, M. M. T., & Graesser, A. C. (2010). Better to be frustrated than bored: The incidence, persistence, and impact of learners' cognitive–affective states during interactions with three different computer-based learning environments. *International Journal of Human-Computer Studies, 68*(4), 223–241.

Baker, R. S., De Carvalho, A. M. J. A., Raspat, J., Aleven, V., Corbett, A. T., & Koedinger, K. R. (2009). Educational software features that encourage and discourage "gaming the system". In *Proceedings of the 14th International Conference On Artificial Intelligence In Education*, 475–482.

Baker, R. S., Walonoski, J., Heffernan, N., Roll, I., Corbett, A., & Koedinger, K. (2008). Why students engage in "gaming the system" behavior in interactive learning environments. *Journal of Interactive Learning Research, 19*(2), 185–224.

Bau, D., Gray, J., Kelleher, C., Sheldon, J., & Turbak, F. (2017). Learnable programming: Blocks and beyond. *Communications of the ACM, 60*(6), 72–80.

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *ACM SIGCSE Bulletin, 39*(2), 32–36.

Bhise, R. B., Thorat, S. S., & Supekar, A. K. (2013). Importance of data mining in higher education system. *IOSR Journal Of Humanities And Social Science (IOSR-JHSS), 6*(6), 18.

Bhuiyan, S., Greer, J. E., & McCalla, G. I. (1994). Supporting the learning of recursive problem solving. *Interactive Learning Environments, 4*(2), 115–139.

Blanca, M. J., Alarcón, R., Arnau, J., Bono, R., & Bendayan, R. (2017). Non-normal data: Is ANOVA still a valid option? *Psicothema, 29*(4), 552–557.

Blanca, M. J., Arnau, J., López-Montiel, D., Bono, R., & Bendayan, R. (2013). Skewness and kurtosis in real data samples. *Methodology, 9*, 78–84.

Bonate, P. (2000). Analysis of pretest-posttest designs. CRC Press.

Borracci, G., Gauthier, E., Jennings, J., Sale, K., & Muldner, K. (2020). The effect of assistance on learning and affect in an algebra tutor. *Journal of Educational Computing Research, 57*(8), 2032–2052.

Brown, N. C. C., Kölling, M., Crick, T., Peyton Jones, S., Humphreys, S., & Sentance, S. (2013). Bringing computer science back into schools: Lessons from the UK. In *Proceeding Of The 44th* ACM Technical Symposium on Computer Science Education, 269-274.

Chi, M. T. H., Bassok, M., Lewis, M. W., Reimann, P., & Glaser, R. (1989). Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science, 13*, 145–182.

Chi, M. T., & Wylie, R. (2014). The ICAP framework: Linking cognitive engagement to active learning outcomes. *Educational Psychologist, 49*(4), 219–243.

Chi, M. T., Siler, S. A., Jeong, H., Yamauchi, T., & Hausmann, R. G. (2001). Learning from human tutoring. *Cognitive Science, 25*(4), 471–533.

Conati, C., & Vanlehn, K. (2000). Toward computer-based support of meta-cognitive skills: A computational framework to coach self-explanation. *International Journal of Artificial Intelligence in Education, 11*, 389–415.

Corbett, A., MacLaren, B., Wagner, A., Kauffman, L., Mitchell, A., & Baker, R. S. (2013). Differential impact of learning activities designed to support robust learning in the genetics cognitive tutor. In proceedings of the international conference on artificial intelligence in education, 319–328.

Costa, J. M., & Miranda, G. L. (2017). Relation between Alice software and programming learning: A systematic review of the literature and meta-analysis. *British Journal of Educational Technology, 48*(6), 1464–1474.

Craig, S. D., Sullins, J., Witherspoon, A., & Gholson, B. (2006). The deep-level-reasoning-question effect: The role of dialogue and deep-level-reasoning questions during vicarious learning. *Cognition and Instruction, 24*(4), 565–591.

Cunningham, K., Blanchard, S., Ericson, B., & Guzdial, M. (2017). Using tracing and sketching to solve programming problems: Replicating and extending an analysis of what students draw. In *Proceedings of the 2017 ACM Conference on International Computing Education Research,* 164-172.

Denny, P., Luxton-Reilly, A., Tempero, E., & Hendrickx, J. (2011). Understanding the syntax barrier for novices. In *Proceedings of the* 16th conference on innovation and Technology in Computer Science Education, 208–212.

Driscoll, D. M., Craig, S. D., Gholson, B., Ventura, M., Hu, X., & Graesser, A. C. (2003). Vicarious learning: Effects of overhearing dialog and monologue-like discourse in a virtual tutoring session. *Journal of Educational Computing Research, 29*(4), 431–450.

Fabic, G. V. F., Mitrovic, A., & Neshatian, K. (2019). Evaluation of parsons problems with menu-based self-explanation prompts in a mobile python tutor. *International Journal of Artificial Intelligence in Education, 29*(4), 507–535.

George, D., & Mallery, P. (2019). IBM SPSS statistics 26 step by step: A simple guide and reference. Routledge.

Gilmore, D. J., & Green, T. R. G. (1988). Programming plans and programming expertise. *The Quarterly Journal of Experimental Psychology, 40*(3), 423–442.

Gobert, J. D., Baker, R. S., & Wixon, M. B. (2015). Operationalizing and detecting disengagement within online science microworlds. *Educational Psychologist, 50*(1), 43–57.

Graesser, A. C. (2016). Conversations with AutoTutor help students learn. *International Journal of Artificial Intelligence in Education, 26*(1), 124–132.

Greer, J. E., & McCalla, G. I. (1989). A computational framework for granularity and its application to educational diagnosis. *In Proceedings of the International Joint Conference on Artificial Intelligence*, 477-482.

Greer, J., McCalla, G., Vassileva, J., Deters, R., Bull, S., & Kettel, L. (2001). Lessons learned in deploying a multi-agent learning support system: The I-help experience. *In Proceedings of Artificial Intelligence in Education*, 410-421.

Grover, S., & Pea, R. (2013). Computational thinking in K–12: A review of the state of the field. *Educational Researcher, 42*(1), 38–43.

Hertz, M., & Jump, M. (2013, March). Trace-based teaching in early programming courses. In *Proceeding of the 44th* ACM Technical Symposium on Computer Science Education, 561-566.

Hosseini, R., & Brusilovsky, P. (2017). A study of concept-based similarity approaches for recommending program examples. *New Review of Hypermedia and Multimedia, 23*(3), 161–188.

Jennings, J., & Muldner, K. (2020). Assistance that fades in improves learning better than assistance that fades out. *Instructional Science, 48*(4), 371–394.

Kapur, M. (2014). Productive failure in learning math. *Cognitive Science, 38*(5), 1008–1022.

Kapur, M. (2016). Examining productive failure, productive success, unproductive failure, and unproductive success in learning. *Educational Psychologist, 51*(2), 289–299.

Kapur, M., & Bielaczyc, K. (2012). Designing for productive failure. *Journal of the Learning Sciences, 21*(1), 45–83.

Kassambara, A. (2017). *Practical Guide To Cluster Analysis in R: Unsupervised Machine Learning* (Vol. 1). STHDA.

Kumar, A. N. (2013). A study of the influence of code-tracing problems on code-writing skills. In *Proceedings Of The 18th ACM Conference On Innovation And Technology In Computer Science Education*, 183-188.

Lahtinen, E., Ala-Mutka, K., & Järvinen, H. M. (2005). A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin, 37*(3), 1418.

Lam, M. S., Chan, E. Y., Lee, V. C., & Yu, Y. T. (2008). Designing an automatic debugging assistant for improving the learning of computer programming. In *Proceedings of the International Conference on Hybrid Learning and Education*, 359–370.

Lee, H. S., Betts, S., & Anderson, J. R. (2015). Not taking the easy road: When similarity hurts learning. *Memory & Cognition, 43*(6), 939–952.

Lee, B. & Muldner, K. (2020). Instructional video design: Investigating the impact of monologue- and dialogue-style presentations. *In Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*, 1–12.

Lister, R., Fidge, C., & Teague, D. (2009). Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *ACM SIGSE Bulletin, 41*(3), 161–165.

Loibl, K., Roll, I., & Rummel, N. (2017). Towards a theory of when and how problem solving followed by instruction supports learning. *Educational Psychology Review, 29*(4), 693–715.

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. *In Proceedings of the Fourth International Workshop on Computing Education Research*, 101-112.

McLaren, B. M., van Gog, T., Ganoe, C., Karabinos, M., & Yaron, D. (2016). The efficiency of worked examples compared to erroneous examples, tutored problem solving, and problem solving in computer-based learning environments. *Computers in Human Behavior, 55*, 87–99.

Muldner, K., & Conati, C. (2010). Scaffolding meta-cognitive skills for effective analogical problem solving via tailored example selection. *International Journal of Artificial Intelligence in Education, 20*(2), 99–136.

Muldner, K., Burleson, W., Van de Sande, B., & VanLehn, K. (2011). An analysis of students' gaming behaviors in an intelligent tutoring system: Predictors and impacts. *User Modeling and User-Adapted Interaction, 21*(1–2), 99–135.

Muldner, K., Lam, R., & Chi, M. T. (2014). Comparing learning from observing and from human tutoring. *Journal of Educational Psychology, 106*(1), 69–85.

Murphy, L., Fitzgerald, S., Lister, R., & McCauley, R. (2012). Ability to 'explain in plain English' linked to proficiency in computer-based programming. In proceedings of the ninth annual international conference on international computing education research, 111–118.

Najar, A. S., & Mitrovic, A. (2013). Examples and tutored problems: How can self-explanation make a difference to learning?. In proceedings of the international conference on artificial intelligence in education, 339–348.

Najar, A. S., Mitrovic, A., & McLaren, B. M. (2016). Learning with intelligent tutors and worked examples: Selecting learning activities adaptively leads to better learning outcomes than a fixed curriculum. *User Modeling and User-Adapted Interaction, 26*(5), 459–491.

Nelson, G. L., Xie, B., & Ko, A. J. (2017). Comprehension first. *In Proceedings of the 2017 ACM Conference on International Computing Education Research*, 2–11.

Price, T. W., Dong, Y., & Lipovac, D. (2017). iSnap: Towards intelligent tutoring in novice programming environments. *In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, 483-488.

Price, T. W., Dong, Y., Zhi, R., Paaßen, B., Lytle, N., Cateté, V., & Barnes, T. (2019). A comparison of the quality of data-driven programming hint generation algorithms. *International Journal of Artificial Intelligence in Education, 29*(3), 368–395.

Renkl, A. (2014). Toward an instructionally oriented theory of example-based learning. *Cognitive Science, 38*(1), 1–37.

Rich, P. J., Jones, B. L., Belikov, O., Yoshikawa, E., & Perkins, M. (2017). Computing and engineering in elementary school: The effect of yearlong training on elementary teacher self-efficacy and beliefs about teaching computing and engineering. *International Journal of Computer Science Education in Schools., 1*(1), 1–20.

Rittle-Johnson, B., Loehr, A. M., & Durkin, K. (2017). Promoting self-explanation to improve mathematics learning: A meta-analysis and instructional design principles. *ZDM, 49*(4), 599–611.

Rivers, K., & Koedinger, K. R. (2017). Data-driven hint generation in vast solution spaces: A self-improving python programming tutor. *International Journal of Artificial Intelligence in Education, 27*(1), 37–64.

Salden, R. J., Aleven, V. A., Renkl, A., & Schwonke, R. (2009). Worked examples and tutored problem solving: Redundant or synergistic forms of support? *Topics in Cognitive Science, 1*(1), 203–213.

Sale, K., & Muldner, K. (2019). Learning with an algebra computer tutor: What type of hint is best? *In Proceedings of Cognitive Science Conference,* 2708–2714.

Sao Pedro, M. A., Baker, R. S., Gobert, J. D., Montalvo, O., & Nakama, A. (2013). Leveraging machine-learned detectors of systematic inquiry behavior to estimate and predict transfer of inquiry skill. *User Modeling and User-Adapted Interaction, 23*(1), 1–39.

Schank, P. K., Linn, M. C., & Clancy, M. J. (1993). Supporting Pascal programming with an on-line template library and case studies. *International Journal of Man-Machine Studies, 38*(6), 1031–1048.

Shovon, M. H. I., & Haque, M. (2012). Prediction of student academic performance by an application of k-means clustering algorithm. *International Journal of Advanced Research in Computer Science and Software Engineering, 2*(7), 353–355.

Simon, & Snowdon, S. (2011). Explaining program code: Giving students the answer helps but only just. *In Proceedings of the 7th Annual International ACM Conference on International Computing Education Research*, 93–99.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM, 29*(9), 850–858.

Sweller, J., & Cooper, G. A. (1985). The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction, 2*(1), 59–89.

Sweller, J., Ayres, P. L., Kalyuga, S., & Chandler, P. A. (2003). The expertise reversal effect. *Educational Psychologist, 38*(1), 23–31.

Tullis, J. G., Goldstone, R. L., & Hanson, A. J. (2015). Scheduling scaffolding: The extent and arrangement of assistance during training impacts test performance. *Journal of Motor Behavior, 47*(5), 442–452.

Vainio, V., & Sajaniemi, J. (2007). Factors in novice programmers' poor tracing skills. *ACM SIGCSE Bulletin, 39*(3), 236–240.

van Gog, T. (2011). Effects of worked examples, example-problem, and problem-example pairs on novices' learning. *Computers & Education, 57*(2), 1775–1779.

van Gog, T., Kester, L., & Paas, F. (2011). Effects of worked examples, example-problem, and problem-example pairs on novices' learning. *Contemporary Educational Psychology, 36*(3), 212–218.

Venables, A., Tan, G., & Lister, R. (2009). A closer look at tracing, explaining and code writing skills in the novice programmer. *In Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, 117–128.

Wang, Y., Nguyen, H., Harpstead, E., Stamper, J., & McLaren, B. M. (2019). How does order of gameplay impact learning and enjoyment in a digital learning game?. *In Proceedings of* the *International Conference on Artificial Intelligence in Education*, 518–531.

Weerasinghe, A., & Mitrovic, A. (2006). Facilitating deep learning through self-explanation in an open-ended domain. *International Journal of Knowledge-based and Intelligent Engineering Systems, 10*(1), 3–19.

Weintrop, D., & Wilensky, U. (2018). How block-based, text-based, and hybrid block/text modalities shape novice programming practices. *International Journal of Child-Computer Interaction, 17*, 83–92.

Whittall, S. J., Prashandi, W. A. C., Himasha, G. L. S., De Silva, D. I., & Suriyawansa, T. K. (2017). CodeMage: Educational programming environment for beginners. In proceedings of the 9th international conference on knowledge and smart technology, 311–316.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33–35.

Wylie, R., & Chi, M. T. (2014). The self-explanation principle in multimedia learning. *The Cambridge Handbook Of Multimedia Learning,* 413-432.S.