Adaptive Immediate Feedback for Block-Based Programming: Design and Evaluation

Samiha Marwan, Bita Akram, Tiffany Barnes, and Thomas W. Price

***Abstract*—Theories on learning show that formative feedback that is immediate, specific, corrective, and positive is essential to improve novice students' motivation and learning. However, most prior work on programming feedback focuses on highlighting student's mistakes, or detecting failed test cases after they submit a solution. In this article, we present our adaptive immediate feedback (AIF) system, which uses a hybrid data-driven feedback generation algorithm to provide students with information on their progress, code correctness, and potential errors, as well as encouragement in the middle of programming. We also present an empirical controlled study using the AIF system across several programming tasks in a CS0 classroom. Our results show that the AIF system improved students' performance, and the proportion of students who fully completed the programming assignments, indicating increased persistence. Our results suggest that the AIF system has potential to scalably support students by giving them real-time formative feedback and the encouragement they need to complete assignments.**

***Index Terms*—Adaptive feedback, block-based programming, formative feedback, subgoals feedback.**

I. Introduction

NOVICE block-based programming environments like Scratch [1], Snap*!* [2], and Alice [3] were designed to foster more positive experiences with programming for novices by enabling creativity and eliminating syntax errors [2], [4], [5]. Increasingly, introductory programming classrooms in K-12 and college classrooms engage novices to create interesting programs (e.g., with interactive input and graphical output), in block-based environments to create a positive first experience with programming. However, since novices have little prior knowledge, they often face uncertainty when programming, e.g., not knowing how to start or whether their code is correct or not [6], especially during more complex and open-ended assignments, which may take 30–60 min, and have many different correct solution approaches [7]. Without confirmation that they are making progress as they work, students may lose motivation and give up early [8], or even delete correct code that they think is responsible for errors [9], [10]. Additionally, without the skills to detect errors, students may submit incorrect code without realizing it, leading to lower course performance and missed opportunities to debug their errors.

It is, therefore, especially critical that novice programmers have access to *timely, formative* feedback to address this uncertainty. In a review on effective feedback, Shute [11] argued that formative feedback can improve students' motivation and learning. Formative feedback is defined as a type of task-level feedback that provides *specific*, *timely* information to a student in response to a particular problem or task based on the student's current ability [11]. From a cognitive learning theory (CLT) perspective, formative feedback can reduce students' uncertainty about how well or poorly they are performing on a task [12], [13], and it can therefore increase students' motivation and persistence to complete tasks by revealing the progress that students have already made [14]. However, it is hard for computing instructors to provide such formative feedback for every student, especially in larger classes, or during homework outside of class. It is also difficult to develop *automated feedback*, since the interactive input and graphical output that make novice programming environments engaging also make it difficult to assess student code with traditional input/output-based test cases.

While some automated feedback approaches have been developed for block-based environments [6], [14]–[16], they have important limitations. First, some of these systems focus on negative *corrective feedback*, with less emphasis on *positive* encouragement for students that marks their progress [17]. Second, some of these systems either require extensive expert effort in hand authoring rules, which is hard to scale across assignments [14], [16], or use error-prone data-driven algorithms to generate such feedback [10], [18]. Lastly, and most importantly, few of these systems have been evaluated in authentic classroom settings to measure their impact on student outcomes.

In this article, we present an adaptive immediate feedback system (`AIF`), shown in Fig. 1, that leverages a hybrid *data-driven* model refined with *experts'* constraints to generate formative feedback on programming tasks' subgoals. This is our third version of the AIF system, which we call AIF 3.0, or for simplicity, just AIF. This system offers students immediate, specific, continuous feedback on their progress through a programming assignment in the Snap*!* environment. To do so, the algorithm behind the AIF system breaks a program down into

Manuscript received 25 October 2021; revised 11 April 2022; accepted 31 May 2022. Date of publication 8 June 2022; date of current version 5 August 2022. (*Corresponding author: Samiha Marwan.)*

This work involved human subjects or animals in its research. Approval of all ethical and experimental procedures and protocols was granted by NC State Institutional Review Board under Application No. 19082, and performed in line with the Intelligent Support for Creative, Open-ended Programming Projects.

The authors are with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695 USA (e-mail: samarwan@ncsu.edu; bakram@ncsu.edu; tmbarnes@ncsu.edu; twprice@ncsu.edu).

This article has supplementary downloadable material available at https://doi.org/10.1109/TLT.2022.3180984, provided by the authors.

Digital Object Identifier 10.1109/TLT.2022.3180984

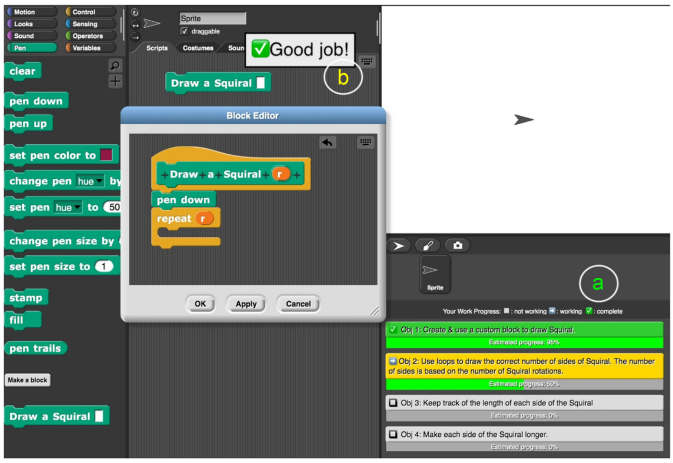

Fig. 1. AIF 3.0 system, with subgoal list (a), and pop-up message (b), augmenting the Snap*!* block-based programming environment.

a set of subgoals (i.e., programming task objectives) and calculates student progress on each from 0% to 100%, which is updated every time a student edits their code. The AIF system interface then presents this progress feedback through a subgoal list, and offers encouraging messages as students progress. Our AIF system improves over prior systems that provide formative feedback [12], [16]–[18], and extends them in three critical ways. First, the system uses a *hybrid* data-driven subgoal detector algorithm to assess students' progress on programming tasks' subgoals, described in detail in [19]. These subgoals are generated automatically from student data, and then *refined* using expert-constraints, rather than purely expert-authored rules [14], [16], or a purely data-driven model [18]. This allowed us to scale the feedback to support a multi-week programming unit, and ensure higher quality feedback. Second, our AIF system can assess students progress more granularly, providing an estimation of their progress from 0% to 100%, rather than binary correct/incorrect feedback, giving students more of a sense of progress. Third, using effective design strategies, the AIF system provides all four forms of effective formative feedback, i.e., feedback that is corrective, specific, immediate, and positive.

We also present an empirical evaluation of the AIF 3.0 system over three weeks of a university-level introductory computing course for nonmajors. This evaluation goes beyond prior evaluations of block-based feedback tools, which were limited to technical assessments [6], lab studies [20], or single-day studies [10], [14], [18]. We also leverage multiple assessment mechanisms to evaluate the system's impact, including a transfer programming task, posttest, and surveys. This evaluation helps us to understand the potential for such systems for wider adoption in classrooms.

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

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

II. Related Work

In this section, we review theoretical perceptions and empirical evidence for the design of effective, adaptive formative feedback, and how it impacts student outcomes.

A. Learning Theories and Empirical Evidence on Feedback

Research on formative feedback suggests that it is effective when it is corrective, specific, immediate, and positive [11], [21], [22], as described in more detail below.

Corrective feedback tells students not only whether their answer is right or wrong, but also provides information to help them achieve a more correct response (e.g., a hint about the solution or a corrective action) [11]. While many automated feedback systems for programming highlight incorrect behavior (e.g., failed test cases [15], syntax errors [23]), few provide clear, actionable information to help a student address these issues, perhaps because such feedback is difficult to design [24]. From a theoretical perspective, such corrective feedback would be more effective because it makes learners aware of their errors and guides them to provide the right answer [21]. In practice, Gusukuma *et al.*[17] conducted an empirical study showing that adding corrective information on top of detecting students' programming misconceptions (i.e., misconception feedback), improved students' performance in a posttest.

Feedback is specific when it provides information about *how* and *where* a student's work does or does not meet assignment goals, such as a feedback that tells the student *what* is missing in their code, not just that their code is incomplete [11], [25].

From a theoretical perspective, Shute [11] and Thurlings *et al.* [21] argued that formative feedback is effective when it is specific and clear. This is because feedback lacking specificity can increase students' uncertainty and the cognitive load needed to understand the feedback or respond to it, which can lead to decreased learning [11], [26]. Several tutoring systems for programming have been shown to improve student performance with specific feedback, such as error highlights and detection of failed test cases [15], [17]. However, there is still a lack of research on how to generate automated specific feedback in programming due to the multiple approaches students can take to reach a correct solution.

Feedback is immediate when it is provided right after a student has responded to an item (or made an action) [11]. While studies have shown benefits for both delayed and immediate feedback [11], some suggest that immediate feedback is more helpful for students with less prior knowledge or less motivation [11], [27]. From a theoretical perspective, immediate, specific feedback can help students to focus their attention on the error at the time that the error occurs, where students can actively correct it before moving on and, therefore, errors do not compound and the salience of the feedback is increased [21]. This is reflected by empirical studies that found an increase in novices' programming performance and learning when they receive immediate feedback [17], [20], [28].

Positive feedback *praises* students when they achieve a step (or a task) appropriately (e.g., by saying: "Good Move!") [23], [29]. Studies in human tutoring dialogs and cognitive learning theories (CLT) show that positive feedback can increase students' confidence in their abilities, decrease their uncertainty about their answer steps, and motivate them to learn [12], [30]–[32]. Such positive feedback can also improve students' affective outcomes [14]. However, positive feedback is not always present in intelligent tutoring systems because such systems were primarily designed to intervene when they detect *incorrect* steps or solutions [12], [23], [33]. As a result, few feedback systems can provide positive feedback proactively in a manner similar to human tutors [12], [23].

Overall, these feedback characteristics can help students close the gap between their actual and the desired outcomes and can have long-lasting positive impacts on student behavior [21], [22]. While there are several other potential characteristics of effective feedback [21], we focus on corrective, specific, immediate, and positive feedback because they are not usually available in programming environments.

B. Adaptive Formative Feedback in Practice

A growing body of work has developed and evaluated computer-based tutors that provide automated programming feedback [34]–[37]. However, most tutoring systems in computing education are only capable of providing feedback with a *subset* of Scheeler *et al.*'s [22] desired qualities—immediate, specific, positive, and corrective. For example, assessment feedback systems, such as autograders, can provide corrective feedback when student code passes or fails a test case; however, this feedback lacks positivity, and it is not immediate since it is offered only when the student submits their code [15], [37]. Other tutoring systems provide hints to help students reach the correct solution [20]. While these hints can be specific and immediate; they do not confirm students' correct steps. An exception is the work of Gusukuma *et al.* [16], who developed a specification language that allows human experts to author formative feedback to novice programmers in a block-based programming environment, supporting various forms of formative feedback, e.g., immediate, corrective, or elaborative feedback. However, their specification language was tested on only one programming task, and its impact on students' outcomes was not evaluated [16].

1) Benefits of Formative Feedback in Practice: There are few tutoring systems that investigated the benefits of formative feedback on students in real classrooms. For example, the most recent version of the iList tutor (iList-5), a tutoring system for linked lists, has been shown to be as effective as human tutors by providing feedback, that is corrective, immediate, positive, and also proactively anticipate students' future moves [23]. The SQL tutor, a database tutor, has been shown to improve student performance through corrective, delayed feedback. Furthermore, adding positive feedback to the SQL tutor has been shown to help students complete problems much faster, (i.e., achieved mastery in learning), compared to those who only only received negative feedback [12].

2) Methods of Generating Formative Feedback in Block-Based Environments: Recent work by Gusukuma *et al.* [17] developed misconception feedback for block-based, interactive programs based on instructors' analyses of prior student work, and found that it improved students' performance in a classroom study. Wang *et al.* [37] created expert-authored rules to provide adaptive feedback that is positive and corrective, finding *qualitatively* that such feedback engaged students in solving short programming tasks faster. Both of these methods require extensive expert effort, making them hard to scale across tasks. In contrast, the `AIF` system we present here achieves similar characteristics of effective feedback, using a method that is more scalable to new tasks [19].

III. Adaptive Immediate Feedback System Design

In this section, we present the `AIF 3.0` system that uses a hybrid data-driven model to provide students with effective formative feedback. The main idea of the `AIF` system is to simplify the students' learning process, and motivate them to complete their programming tasks, leading to an increase in students' performance and learning. To do so, we designed the `AIF 3.0` system to provide feedback that is *corrective*, *specific*, *immediate*, and *positive*, which are key aspects of effective formative feedback for learning as specified in the literature [11], [22]. The `AIF 3.0` system augments the Snap*!* block-based programming environment with the following three main components.

1) A subgoal list, which breaks down the current programming task requirements into 3–5 smaller, more manageable task objectives/subgoals [shown in Fig. 1(a)].

2) Pop-up messages, which provide students with encouraging messages based on their progress [shown in Fig. 1(b)].
3) An algorithm for data-driven subgoal detection, proposed and evaluated in prior work [19], which we used in the AIF 3.0 system to generate the list of subgoals, and detects students' progress on them in real time, driving the feedback in the subgoal list and the pop-up messages.

The first two components represent the interface, described in Section III-A, while the third component presents the back end of the AIF 3.0 system, described in Section III-B.

A. AIF 3.0 System Interface

Our current AIF design is a product of years of prototyping, deployment, and refinement [10], [14], [18]. Below we describe AIF 3.0's current interface components in detail.

1) Subgoal List: The primary feature of our AIF system is a list of subgoals/objectives, and feedback on student progress on completing each subgoal, as shown in Fig. 1(a). The subgoal list breaks down a given programming task into a set of smaller objectives (i.e., subgoals) that students can attempt one at a time, which has been shown to be effective in improving students' performance in programming [38]. In particular, we created the subgoal list for three reasons. First, many novices struggle with how to *start* an assignment [8], [32], so we provide a concrete set of steps (a plan) to get them started. Second, for complex programming tasks, especially open-ended ones, students must navigate a large search space, which can lead to high cognitive load [39]. Our intuition was that breaking a problem into smaller subgoals, which are smaller structural parts of the overall programming assignment [38], should mitigate this challenge and increase students' ability to solve the task. The process we used to generate subgoals is detailed in Section III-B. Third, for subgoals to be effective, students need to know when they have completed one subgoal, so they can focus on the next. To reduce this uncertainty about subgoal (and assignment) completion, AIF 3.0 continuously updates each subgoal with a progress bar after each edit to show the student's estimated progress on each subgoal. This progress is calculated by an algorithm that analyzes student code after each edit, as described in Section III-B. Fig. 1(a) shows the subgoal list and progress feedback for the *Squiral* assignment, whose output is shown in Fig. 4. For example, one subgoal is to *"Make each side of the Squiral longer."*

a) *Subgoal list design:* When students open Snap*!* and choose which task to work on, they will see a list of subgoals shown as objectives, each with a short, hand-authored label.[1] The subgoal list is placed within the programming environment to align with the multimedia learning principle of contiguity, which states that information needed to perform a task is most useful when it is placed next to where it is needed [40]. Initially, each subgoal background is colored grey to indicate that none of the subgoals have been attempted. Students can interact with the subgoal list in two ways. First, they can click on a subgoal to highlight it in yellow and get more information, as shown in "subgoal 2" (i.e., obj2) in Fig. 2. This information consists of expert-authored explanations derived from the task instructions, since research suggests that students prefer feedback augmented with explanatory text [41], and find it more interpretable [42]. Second, students can double-click on a subgoal to mark it as complete (as shown in "obj 1" in Fig. 2), aligning with self-regulated learning principles, which state that learning is improved when students track their own progress and self-assess their learning process [43]. This feature was also recommended by students in evaluations of earlier versions of the system [14], [32].

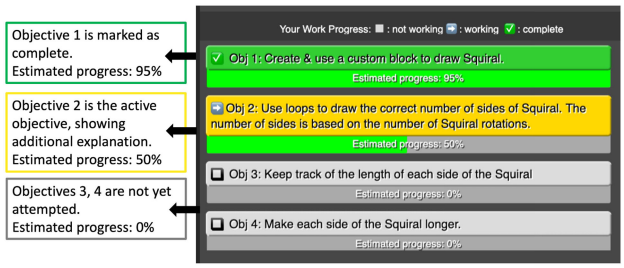

Fig. 2. Subgoal list in Squiral assignment, with four objectives (i.e., subgoals), and explanation of each on the right.

b) *Adaptive progress feedback display:* The subgoal list adapts to students' code edits by providing feedback on students' progress on each subgoal in real time, aligned with the learning design principles of specific, immediate feedback that enables students to understand what specific actions they have just taken, that are leading to mistakes or progress. Rather than telling students only whether a subgoal is complete or not, as in typical programming autograders [14], [15], [37], AIF 3.0 presents a progress bar underneath each subgoal to show how far students have progressed toward completing that subgoal, as shown in Fig. 2. This provides students with *specific, corrective* feedback on each edit they make to their code, by showing how that code edit increased or decreased the estimated progress toward a correct solution.

c) *Promoting self-assessment:* A key aspect to improve novice programmers' learning is to promote them to self-assess whether their program is correctly solving the given problem [32]. Therefore, we have purposefully designed two important features to encourage self-assessment and self-regulated learning in the AIF 3.0 system. First, we provide transparency into how the system works, explaining to students that the algorithm that tracks their progress may sometimes fail to recognize correct solutions, as described below. Second, we set the maximum progress on each subgoal to 95%, instead of 100%, to remind students that it is up to them to self-assess and double-click on each subgoal when *they* decide it is complete, which is suggested by prior work [32].

2) Pop-Up Messages: We developed pop-up messages because prior work shows that positive feedback that praises students' accomplishment and perseverance can promote students' confidence [44], which may increase their persistence to complete tasks. Therefore, the AIF 3.0 system provides positive feedback in the form of *congratulating* messages to praise students

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

TABLE I
EXAMPLES OF POP-UP MESSAGES IN THE AIF SYSTEM

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

when they complete a subgoal or fix a broken subgoal. It also provides *encouraging* messages when students persevere by spending a long time without any progress, as shown in Fig. 1(b). To design these messages, one researcher asked undergraduate students to construct messages to praise a friend's achievement when they complete a subgoal, or motivate a friend when they are struggling or losing progress. While students are programming, the AIF 3.0 system selects a personalized pop-up message based on students' code and actions. For example if the system detects the completion of a subgoal, it will pop up a congratulating message like: "Good job!," or an encouraging message like "Don't give up" when a student has spent more than 4 min without making any progress.[2] Table I shows examples of pop-up messages.

B. Data-Driven Subgoal Detection Algorithm

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

In prior work, there are two common approaches to generate and assess subgoals: 1) expert-authored; and 2) data-driven approaches. For expert-authored approaches, human experts define the subgoals of a correct solution, and create autograders for each objective to detect if it is complete or incomplete, for example using static code analysis [14], [16]. While expert-authored models are capable of providing highly accurate feedback, creating them is time consuming, requires extensive expert effort, and is hard to implement for open-ended programming tasks due to the large number of possible solutions. The second common approach uses data-driven models, which can detect subgoal completion in the current student's code based on *features* learned from historical student data [18]. While data-driven approaches are highly scalable across programming tasks, and require less expert effort, they are dependent on the quality of prior students' solutions, leading to a possibility of providing inaccurate feedback [10].

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

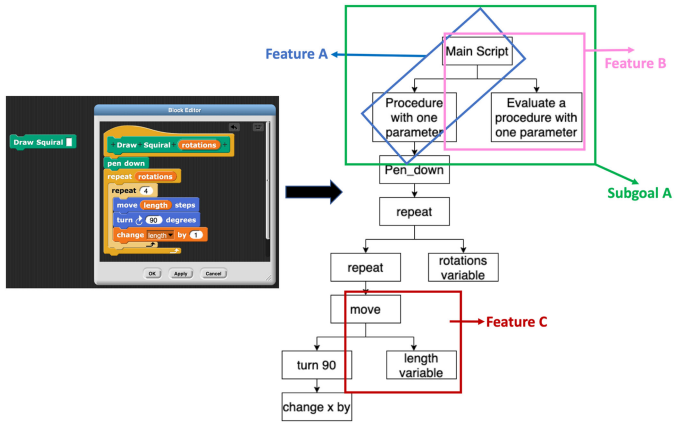

Fig. 3. Example of an abstract syntax tree (AST) of Squiral task code. Features A, B, and C are examples of Squiral data-driven features.

In this article in the AIF 3.0 system, we applied a hybrid model that combines both a data-driven model refined with expert constraints to detect subgoals [19]. As evaluated in prior work [19], the hybrid model can achieve the best of both the expert and data-driven models—building a system that can intelligently address the diverse but correct ways that students solve problems, while benefiting from human expertise in filtering, combining and communicating a high quality data-driven subgoals. In the following paragraphs, we describe the methodology of applying the hybrid model for multiple programming tasks in the AIF 3.0 system; however, the technical details of the hybrid model, including the data-driven detectors and expert-authored modifications, and their accuracy evaluation are more completely described in our prior work [19], [45].

The AIF 3.0's hybrid data-driven model is responsible for subgoal generation, progress assessment, and feedback generation. This hybrid model extends a data-driven feature detector (DDFD) algorithm developed by Zhi *et al.* [45]. The DDFD algorithm is designed to extract common features present in prior students' correct solutions (e.g., from prior semesters) [45]. In brief, the DDFD algorithm works as follows. First, it translates students' correct solutions into abstract syntax trees (ASTs). For example, Fig. 3 shows an example of students' code, and its corresponding AST. Second, it extracts common code shapes (i.e., subtrees of ASTs); such that a group of code shapes can represent a feature of a correct solution. More precisely, a feature describes a distinct property for a correct solution, such as: using a 'procedure with one parameter' in the code, or using a "move" block nested with a variable are two distinct features of a correct solution as shown in Fig. 3(a) and (c), respectively. The DDFD then filters redundant code shapes, and performs hierarchical clustering of frequently co-occurring code shapes to generate more coherent features.

In addition, the DDFD algorithm also defines *disjunction* shapes that identify when there are multiple, distinct ways of solving a subgoal, i.e., a set of code shapes where *one* is

present in most solutions, but not *others*. For example, a student can draw a square using a "loop" or a set of redundant blocks, and a correct solution must have *only one* of these disjunction shapes. The ability to detect disjunction shapes enables the DDFD algorithm to detect subgoals from *multiple* different implementation strategies, allowing it to help students with diverse code. However, it is limited to the implementations learned from prior students' correct solutions. Once the DDFD algorithm learns features of correct solutions, it can be applied to new students' solutions to detect the completion or absence of these features. Since the DDFD algorithm works regardless of whether student code is complete or not, it can be used to provide *immediate* feedback on feature completion.

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

To tackle these challenges, we made several improvements to the DDFD algorithm to detect *subgoal* completion, instead of *feature* completion, in a more *interpretable* and *concrete* way. We call our modified algorithm a hybrid subgoal detector, and its overall algorithm consists of five steps, using both automated and expert insight, which we explain below (with more details here [19]).

First, the hybrid subgoal detector algorithm applies the DDFD algorithm to students' correct solutions from several prior semesters to generate clusters of data-driven features. Second, human experts manually group the generated clusters into more meaningful clusters that are interpretable and reflect discrete assignment requirements. For instance, a programming task that includes 7–10 lines of code is broken down into three to four subgoals which reflect objectives found in the task instructions provided to students. To clarify the difference between a feature and a subgoal, consider the following example in Fig. 3. Assume that the DDFD algorithm generates two features: one requires that student code includes a procedure with one parameter (Fig. 3, Feature A); the second feature requires the evaluation of a procedure with one parameter (i.e., calling a procedure), Fig. 3, Feature B. A meaningful subgoal in this case can be the combination of these two features, which means that a correct solution must have a procedure with one parameter which is called (i.e., evaluated) in the main script, as shown in Fig. 3, subgoal A.

Third, we developed a percent progress estimate that reflects student code's *progress* on each subgoal. For example, if the algorithm generates four data-driven subgoals for a given exercise, and for a given student code the algorithm outputs `{50%, 0, 0, 80%}`, it means that the student code has completed 50% of subgoal 1, 80% of subgoal 4, and 0% of subgoals 2 and 3. The algorithm calculates this percentage by calculating how many code features needed for a given subgoal are present in students' code. This new progress tracking feature developed for `AIF 3.0` allowed it to provide more *specific* progress feedback.

Fourth, we modified the generated data-driven features by adding human constraints to improve the quality of subgoal detection using the same validated method in our prior work [19]. First, to understand where the purely data-driven subgoal detectors were *not accurate*, we applied them to prior students' code snapshots (an average of 150 code edit by each student for a given programming task) and logged the output of the subgoal detectors (e.g., for a given student's code edit, the subgoal detection output might be [1, 0, 0, 1], meaning that subgoals 1 and 4 are detected). Then, a group of three human experts (researchers in block-based programming) searched for false detections—instances where the algorithm detects the completion or incompleteness of a subgoal but the human expert disagrees. Considering these false detections, we modified the detectors by adding, editing or deleting code features that made up the subgoal detector. This is possible because the data-driven subgoal detectors search for simple, human-interpretable code shapes in students' code, and the experts were able to modify those code shapes directly with simple changes, without having to author them from scratch. For example, in the *Squiral* task (shown in Fig. 3), one of the required elements of a correct solution is using the 'pen down' block, needed for drawing. This element is detected by the subgoal 2 detector (shown in Fig. 2); however, the original, data-driven subgoal 2 detector detects this element only if the "pen down" block is inside of a procedure. This works most of the time, but not if a student adds the "pen down" block *outside* a procedure, making a false negative detection. To fix this false detection, human experts modified the feature to detect the presence of the "pen down" block inside *or* outside a procedure. After updating all detectors, we ran the hybrid subgoal detectors (i.e., the data-driven subgoal detectors with experts' modifications) on prior students' data, ensuring that the false detections were ∼100% corrected. In our prior work we evaluated this process on the *Squiral* task, where we found the accuracy of the data-driven subgoal detection increased from 74% to 88% overall when we added three expert modifications to create hybrid detectors, with individual subgoals improving by up to 34% points in accuracy [19]. In this article, we applied the hybrid subgoal detectors on two additional tasks (*PolygonMaker and Daisy*) in the `AIF` 3.0 system, described in Section IV-B, to allow for evaluation of the potential for AIF to impact student outcomes across tasks.[3]

Fifth and last, for all the three programming tasks, we augmented each subgoal with a short, human-authored label designed to be *interpretable* for both students and instructors. We presented these labels in a subgoal list for each task as explained in Section III-A1. We then use this list to present feedback on each subgoal using the hybrid algorithm that continuously runs in the back end of the Snap*!* programming environment.

[3] Subgoal labels of PolygonMaker and Daisy tasks are available at https://go.ncsu.edu/subgoals2021.

TABLE II
SUMMARY OF THE DIFFERENCES BETWEEN VERSIONS OF THE AIF SYSTEMS

	AIF 1.0	AIF 2.0	AIF 3.0
Subgoal Labels	Short labels	Short labels	Short labels with expandable descriptions
Feedback	Correctness (complete/absent)	Correctness (complete/absent)	Progress (continuous)
Interactivity	None	None	Student select subgoals, self-assess completion
Subgoal Detection	Expert-authored autograders[14]	Data-driven feature detection algorithm[18]	Hybrid subgoal detector algorithm[19]

1) Initial Results: In Summer 2019, we developed and tested AIF version 1.0 that uses expert-authored autograders, called objective detectors, to detect the absence or completion of each objective in student code [14]. We embedded these autograders in the Snap*!* environment to provide students with adaptive immediate feedback during programming. The initial interface differed from the current version in three ways: 1) used only short labels for subgoals; 2) provided correctness feedback, i.e., if a subgoal is complete or not; and 3) did not allow students to check off a subgoal if they think it is complete. In a controlled study, we evaluated AIF 1.0 with students performing short programming tasks in a 1-day summer workshop [14]. We found that AIF 1.0 improved students' engagement with the programming tasks, and increased students' intentions to persist in computer science.

In Spring 2020, we built AIF 2.0 using a data-driven feature detection (DDFD) approach, described in Section III-B, to learn subgoals from prior correct solutions [18], instead of expert-authored autograders. AIF 2.0 has a similar interface to AIF 1.0. We conducted a controlled study for one programming homework task and showed that AIF 2.0 increased students' engagement as measured by increased time on task. However, investigating the quality of data-driven subgoals detections, we found instances where false detections (i.e., false positives and false negatives) occurred, which might have affected students' performance or trust in the learning environment [10]. To mitigate these inaccuracies, we developed a hybrid data-driven algorithm, described and evaluated in more detail in [19], which adds human constraints to improve the quality of the generated data-driven feedback. For simplicity, Table II shows a summary of the differences between versions of the AIF system.

The remainder of this article presents our study of AIF 3.0, which uses the hybrid data-driven model for multiple assignments across multiple weeks in an authentic introductory nonmajors computer science classroom. For simplicity, we will refer to AIF 3.0 as AIF throughout the rest of the article.

IV. METHOD

We conducted a controlled classroom study in Spring 2021 in an introductory programming course (CS0) at a public U.S. university. This course introduces the principles of computer science (CS) to nonmajors, where students use the Snap*!* environment to learn programming. Snap*!* is a block-based programming environment built by UC Berkeley and designed to be used in the *Beauty and Joy of Computing* (BJC) curriculum used in CS principles courses for nonmajors in several universities [2]. Over the years, the Snap*!* environment has successfully leveraged the affordances of block-based programming languages for more complex problems (e.g., recursion) needed in college level introductory computing courses [2].

In this article, our research questions are the following. RQ: In an authentic classroom setting, what impact does a hybrid data-driven AIF have on (RQ1:) students' performance, (RQ2:) task completion rate, and (RQ3:) learning, and (RQ4:) how is it perceived by students? We hypothesized that AIF will help students know whether they are progressing or not, encouraging them to *persist* longer, which will lead to higher performance (measured by students' grades in these tasks) than students in the *Control* group (H1-performance). Furthermore, knowing that their code is incorrect or incomplete will motivate students to keep working to finish it *fully*, which will result in higher rates of *fully correct*, complete code being turned in (H2-completion). Additionally, by completing more of the assignment correctly, we hypothesize that students will *learn more*, which will be reflected in increased performance on a future transfer task, and on a future posttest (H3–learning).

A. Population

The CS0 course includes 65 undergraduate students, 50 of whom consented to our IRB-approved study. In this population, 63% of participants self-identified as male, 28.26% as female, and 8.6% unspecified; 62.5% are 20 years old or younger, 26.08% are 24 years old or younger, and 8.6% otherwise. Students self-identified their race/ethnicity with 63.03% White, 15.2% Asian, 10.8% Black or African American, 4.3% Native American, and 6.5% Other. While the course was designed for nonmajors without any programming experience; some might have had prior programming experience. This population is similar to most Snap*!* users, who are typically students in high school (grades 10–12) and first-year undergraduates taking an introductory computer science principles course [2].

B. Procedure

We conducted a controlled study that lasted for three weeks. Due to the COVID-19 pandemic, this class was held online through Zoom. On the first day of the CS0 class, one researcher introduced the study to all students and offered them the opportunity to consent to participate in the research. Students were then randomly assigned to the *AIF* condition or the *Control* condition, allowing for between-subjects comparison. In the *AIF* condition, students received feedback through the AIF system, and in the *Control* condition, students did not have access to the AIF system. We note that students in both groups had detailed assignment instructions with ordered steps. Additionally, both had equal access to request next-step hints from Snap*!*, which suggests a single edit to bring student

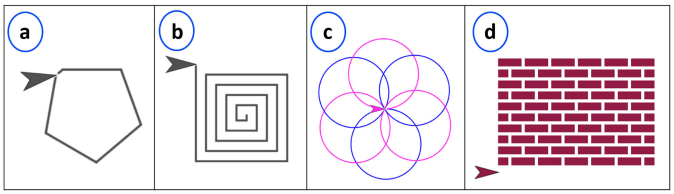

Fig. 4. Expected output of Polygon Maker (a), Squiral (b), Daisy (c), and Brick Wall (d).

code closer to the correct solution. Hints were *already* part of the class to help students when they got stuck, so we do not analyze its effect in this study. Out of the 50 students, 23 were assigned to the *AIF* condition, and 27 were assigned to the *Control* condition.

1) Preassessment (HW0): In the first week, students were assigned HW0, which is a self-paced, independent programming assignment that teaches students the usage of loops [37]. It includes nine subtasks, each requiring 3–10 blocks of code, with increasing difficulty. We used HW0 as a preassessment[4] of students abilities, and Snap*!* did *not* provide AIF support to either condition. We did not perform a traditional pretest, since this was many students' first experience with programming, and we did not want to create a harmful expectation that students should know material that is not yet covered in class.

2) Practice Tasks: We describe three programming tasks, together called practice tasks, that students completed using AIF or not, based on their condition assignment, with the graphical output of each shown in Fig. 4. In week 1, students had their first *in-class* programming task called *PolygonMaker*, which asks students to create a procedure with 3 parameters: *"n," "len," "thick,"* to draw a polygon with *"n"* sides, each with with length *"len,"* and thickness *"thick."* In week 2, students had two homework tasks: *Squiral* and *Daisy*[5]. *Squiral* asks students to create a procedure that takes user input *"r"* and draws a spiral square with *"r"* rotations. *Daisy* asks students to create a procedure that draws a daisy with a user-specified number *"n"* of overlapping circular petals with alternate colors. For all these tasks, solutions generally conformed to a single high-level strategy, which was reflected in the AIF system subgoals. This is because the AIF system is specifically designed to support more straightforward, introductory-level tasks, with instructions that specify a set of subgoals. While within a given subgoal there may be different possible implementations (e.g., using multiple procedures or just a single large one); the algorithm generating feedback in the AIF system uses disjunction features (described in Section III-B) to detect diverse subgoal implementations.

3) Post Measures: In week 3, students completed a posttest programming assignment without the AIF system in either condition, to compare the impact of AIF on learning for a similar transfer homework task without adaptive immediate feedback. This homework 3, *Brickwall*, asked students to draw a wall of bricks, with alternate rows of bricks using nested procedures, variables, conditions, and loops. In week 4, the instructor gave students a postsurvey that collected students' perceptions about whether the AIF was helpful and why; followed by a posttest. The posttest includes seven multiple choice questions on variables, loops, and conditionals, adapted from the block-based commutative assessment [46].

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

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

C. Measures

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

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

2) Performance: We measured the impact of the AIF system on students' performance by comparing students' scores (i.e., grades) in all practice tasks across the two treatment conditions. We graded each task, blind to condition, adopting a 4-item binary rubric for each, adapted from the assignment instructions. Each rubric item roughly corresponds to an AIF subgoal, and completing all four rubric items is equivalent to completing the programming task. One researcher graded students' submissions of all programming tasks. To get credit for a rubric item, a student code has to: 1) be functioning such that its output matches the output for that particular rubric item; and 2) use the key programming code blocks required (i.e., a loop, or an if condition). Students receive 0 in a rubric item if it was incomplete/missing in their code, and 1 otherwise, where the grades ranged from 0% (i.e., student failed to complete all rubric items) to a maximum of 100% (i.e., student successfully completed all rubric items). To avoid bias, the researcher compared their grades with the TAs' grades, and then discussed/resolved a small number of conflicts. Across both conditions and all practice tasks, the mean programming performance was 89% (min = 0%; SD = 22.7%; max = 100%).

Analytical approach: We measured programming performance for all practice tasks together to measure the overall effect of the AIF system on students' programming performance through several weeks. To do so, we used linear mixed effects models because the data have a nested structure, where a student's score in each exercise is treated as one unit of analysis, accounting for the lack of independence between observations [47].

3) Completion Rates: We defined the completion rates of each group (*Control* or *AIF*) as *the proportion of students who turned in fully correct, complete code*, where a fully complete

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

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

Analytical approach: We evaluated students' correctness rate on all three practice tasks together to measure the overall impact of the AIF system on students' completeness rate for all three tasks through several weeks of programming. To do so, we used a linear probability model (LPM) with mixed effects to predict the likelihood that a student completed a task ("1" = completed, "0" = not completed). Because data is binary (i.e., whether a student submitted a complete correct solution (score = 100%), or not [score $<$ 100%)], we used linear probability models (LPMs) which are simply linear mixed effects models that predict *binary* outcomes [48].

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

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

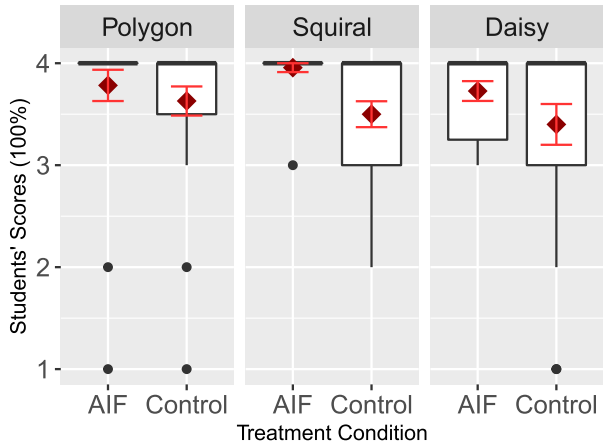

Fig. 5. Student practice task scores for AIF and control groups.

5) Students' Perceptions: We conducted thematic analysis on students' open-ended responses in the postsurvey, to identify themes on why the AIF system is more or less helpful. We followed six steps suggested in the practical guide of performing thematic analysis by Maguire *et al.* [49]. Two researchers, independently, 1) got familiar with the data, 2) generated a total of eight initial codes, 3) and then met to combine initial codes in dominating themes which resulted in five themes, 4) reviewed all themes together, and then 5) refined them to focus on only two main themes: *mechanisms* for which the AIF system was helpful, and *limitations* in its design. Step 6 was the write up of the results, which we present in Section V-E.

V. Results and Analysis

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

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

A. Performance Results(H1-Performance)

To measure performance, we compared scores of students who attempted the practice tasks. Fig. 5 visualizes boxplots and averages for each condition's performance on each task. As shown in Fig. 5, students using the AIF system have consistently *higher* practice task scores than that of the *Control* group.

TABLE III
ESTIMATED COEFFICIENTS (STANDARD ERROR) OF LINEAR PROBABILITY MODELS (LPM) AND LINEAR MIXED-EFFECTS (LME) MODELS PREDICTING STUDENTS' PERFORMANCE (I.E., GRADES) ON PROGRAMMING TASKS (MODEL A), AND THE LIKELIHOOD THAT A STUDENT COMPLETED PROGRAMMING TASKS (MODEL B), RESPECTIVELY

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

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

We then used a linear mixed-effects model (as in Section IV-C2), to compare combined performance on the practice tasks between the *AIF* group and the *Control* group. In Table III, we use condition (*AIF* = 1, *Control* = 0) and *TaskLevel* as independent variables in our model to predict students' performance as the dependent variable. The "TaskLevel" represents the categorical order of the programming tasks with two indicator variables to represent distinct categories [50], where "0,1" is PolygonMaker, "1,0" is Squiral, and setting both PolygonMaker and Squiral to 0 (i.e., "0,0") is Daisy (therefore, we do not need a third variable for Daisy). The total number of observations is 146 instead of 150 (3 tasks * 50 students) since we excluded four students because of missing data[6].

Model A (shown in Table III) does not show a statistically significant impact of *TaskLevel* on students' performance. However, the model shows that the AIF system significantly improves students' performance. As shown in Table III, the *AIF* variable has a significant impact on performance (p = 0.025), with the coefficient suggesting that students using the AIF performed, on average, 7.76% points higher, compared to the *Control* students across all three practice tasks. Since the average grade was 89%, this represents a distinct improvement. These results provide support for H1-performance, as the AIF system improves students' performance across the practice tasks.

B. Completion Rates (H2-Completion)

The improvement in scores with AIF confirms H1, but does not help us understand *how* students benefited. We hypothesized that the AIF system would benefit students by encouraging them to persist and helping them to catch mistakes, both of which would translate into higher task completion rates. Therefore, we measured students' willingness and ability to persist to task completion using the *completion rate, i.e., a binary score of 1 for 100% score and 0 otherwise*. Grouping all the three tasks together, we use a linear probability model (LPM) with mixed effects to predict the likelihood that a student completed a task ("1" = completed, "0" = not completed,

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

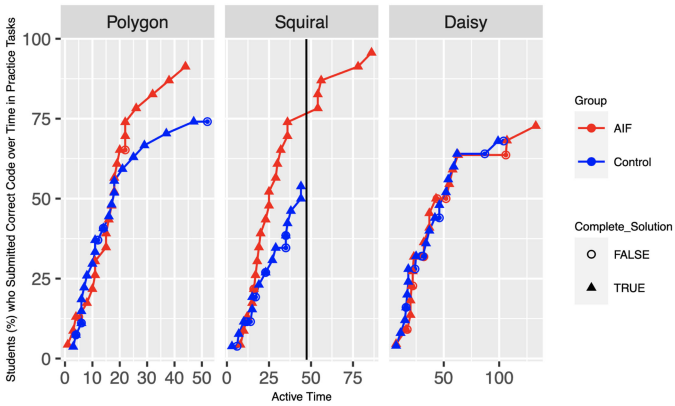

Fig. 6. Percent of students who submitted correct code over time in practice tasks. The triangular points indicate complete submissions, and the circular points indicate incomplete submissions.

dependent variable), using the student's treatment condition and the *TaskLevel* as predictors (independent variables) as we described above in Section V-A. As shown in Model B in Table III, we find no effect of the *TaskLevel* on students' likelihood of completing a task (p = 0.12, 0.93). However, we found that the AIF system had a significant effect on completion rate (p = 0.01), and students using the AIF system were overall 20% more likely to complete a programming task than students in the *Control* group. Looking at each task separately, we found that a higher number of students in the *AIF* group completed each task (91.3%, 95.7%, and 72.7%) than that of the *Control* group (74.1%, 57.7%, and 68%) in *Polygon Maker*, *Squiral*, and *Daisy* tasks, respectively. We see that the group using the AIF system was consistently more likely to complete each assignment. These results provide support for H2-Completion, but also suggest the need for further investigation, as discussed in Section VI.

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

Fig. 6 plots the percentage of all students in each group who had submitted a *complete, correct* submission (y-axis) as time *progressed* (x-axis). Each point represents one student submission, but only complete submissions (triangles) correspond to an increase in the y-value. First, in *Polygon Maker* and *Squiral* tasks, we see a gap suggesting that both groups were submitting code, but AIF group submissions were less likely to have

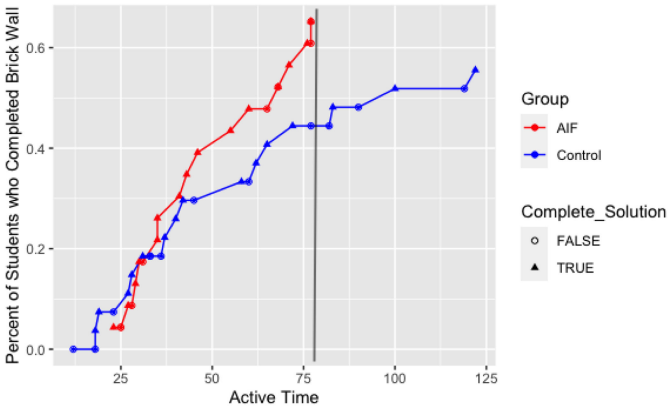

Fig. 7. Percent of students who submitted code over time in Brick Wall homework. The triangular points indicate complete submission, and the circular points indicate incomplete submission.

errors. Additionally, in *Squiral*, we see that 22% of students (5/23) in the *AIF* group *continued* to work and submit correct code, even after all *Control* group students had submitted (solid black line), suggesting increased persistence to submit complete code. This is consistent with the fact that students in the *AIF* condition spent more time on each task, spending 2, 9, and 4 min more than that spent by the *Control* group, in *Polygon Maker*, *Squiral*, and *Daisy*, respectively. These exploratory results suggest that students in the *AIF* condition were not necessarily *more efficient* at solving each problem, but instead, they may have invested the necessary time, which paid off in the form of increased task completion rates.

C. Learning Results (H3–Learning)

We hypothesized that students who received adaptive immediate feedback on their progress and completed more correct tasks would therefore learn more and perform better on subsequent programming and assessment tasks. We first measured learning by investigating student transfer performance on HW3 *BrickWall*, where neither group used the AIF system nor received hints during programming this task. We found that students in the *AIF* group completed 5.13% (M = 91.6%; Med = 100; SD = 14.43) *more* subgoals than the *Control* group (M = 86.54%; Med = 100; SD = 17.65) in *BrickWall* task. While a Mann-Whitney U Test does not show a significant difference ($p = 0.31$), perhaps due to a ceiling effect (most students performed perfectly in both groups), there was a medium effect size (Cohen's $d = 0.34$)[7]. We also compared the proportion of students submitting correct attempts in *BrickWall*. We found that 71.43% of the *AIF* group submitted correct attempts, which is higher than that submitted by the *Control* group (57.7%), but a Fisher's exact test shows that this difference was not significant ($p = 0.37$). It is also worth noting that the six *slowest* students to turn in *BrickWall* were all in the *Control* group (22% of the group), taking longer than 77 min (Fig. 7, solid black line) to submit the assignment, and only 50% of these (i.e., 3) were fully correct. While we see from the results that students using the AIF system performed better in the transfer task, and in less time than that of the *Control* group, these results provide inconclusive evidence to support H3-learning, i.e., the AIF system did not significantly improve students' performance in a transfer task.

Our second measure of learning is comparing students' scores in the posttest (described in Section IV-C4). We found that 48 students took the posttest; 21 in the *AIF* group, and 27 in the *Control* group. On average, we found that the *AIF* group solved more questions correctly (M = 89.11%; Med = 85.71%; SD = 10%) than the *Control* group (M = 83.07%; Med = 85.71%; SD = 15.87%). A Mann–Whitney U test showed that this difference was not significant ($p = 0.23$); but had a medium effect size (Cohen's $d = 0.44$). Taken together, these results suggest that the AIF system does not harm learning (i.e., by making the tasks too easy), and *may* provide a benefit to learning if a larger study were performed.

D. Case Studies

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

1) Case Study Sam: This is a case study of student *Sam* who was assigned to the *AIF* group and submitted a complete attempt of *Squiral* task after 19 min of work. This case study presents how students can use the AIF system, and how it might have helped them complete a task. Sam started by creating a custom block named "draw squiral" nested with a move and turn blocks, where the AIF system showed them 33%, 16%, and 18% increase in objective 1, 2, and 4, respectively. *Sam* kept working for three more minutes, running their code 29 times, without notable progress and the system popped up an encouraging pop up message: "keep going," *Sam* then created a "length" variable and used it in the move block, and immediately the system noted they completed subgoal 3. Sam asked for a hint that suggested using a "change _ by _" block; however, *Sam* used a different block that has a similar color to the "change" block, and no increase in its corresponding subgoal took place. *Sam* then clicked on objective 1, where the system marked it in yellow and presented more descriptive text of objective 1 (as described in Section III). Sam then double clicked on objective 3 and marked it as complete, and then clicked again on objective 1. Afterward, *Sam* asked for a couple of hints, and followed them immediately, leading to completion

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

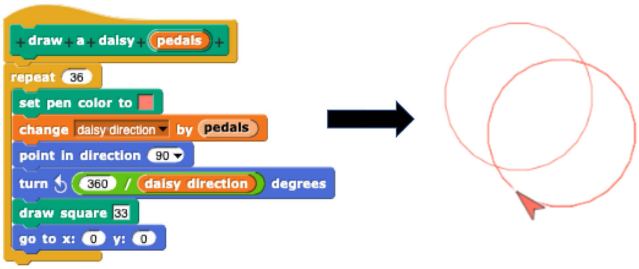

Fig. 8. Submitted code by student Em and its output on the right.

of objectives 2 and 4, which the student then marked as complete. The system by default then turned objective 1 to yellow, as it is the remaining incomplete objective. While *Sam*'s code was producing the correct output for *Squiral*, the code did not use an input parameter for the number of rotations in the "draw squiral" custom block. Sam then spent three more minutes to successfully add the parameter, and the system popped up a message: "You finished the 4 objectives!." Finally, *Sam* ran their code with several inputs and then submitted their code.

Sam's log data showed that their code was often able to generate the correct output, but had missing components needed for full credit. However, *Sam* persisted until all the objectives were complete. This may have been due to the AIF system's tracking of *Sam*'s progress, and presenting how far they had progressed in each objective. In contrast, in the *Control* group, we found that 91% of students who had incomplete code had submitted code that seemed to generate the correct output but was missing components of a correct solution. This might be due to their lack of understanding of some programming concepts, like how to use function parameters, or students relying on assessing the output generated by their code. From an instructors' perspective, these seem like natural mistakes for novices to make. It is also worth mentioning that we found no new implementation strategies in any of the *Squiral* task's subgoals used in *Sam*'s case study. Overall, *Sam*'s case study shows how helpful the AIF is in prompting students to self-assess their code and helps them determine whether it meets the assignment's requirements; which led them to have higher performance and completion rates than that of the *Control* group.

2) Case Study Em: This is a case study of student *Em* who was assigned to the *Control* group, and therefore they did not use the AIF system. *Em* spent 31 min actively programming the *Daisy* homework, and ended up submitting incomplete code. At the beginning of solving *Daisy*, *Em* started by creating a procedure to "draw a circle," and at this moment, if they *had had access* to the AIF system, it would have shown that they completed the objective to "draw a circle." *Em* then kept working ∼20 min and requested nine hints while trying unsuccessfully to complete the "draw daisy" procedure. *Em* then deleted *all* their correct code, which is a common behavior among novices when they are *uncertain* about the correctness of their code [9]. At this moment, if *Em* had access to the AIF system, it would have shown that their progress in one of the objectives went down to 0%, which might have encouraged them to restore their deleted code. *Em* then stopped working for 2 h, and when they returned, they made a couple of edits and finally turned in incomplete code as shown in Fig. 8 after a total of 34 min. If *Em* had been using the AIF system, they would have seen that their last edits improved their progress ∼ 15% and that they were overall 65% close to the correct solution. In contrast, students using AIF for *Daisy* spent an average of 44 min, and every student using AIF achieved at least 75% progress on this task (as shown in Fig. 5).

These results suggest that providing only hints to students is not enough; and that hints and AIF feedback have complementary goals. Our proposed AIF system feedback provides information about student errors and successes, while hints only address student uncertainty about what to do next. AIF feedback is offered continuously in the background, while hints are provided on demand. AIF feedback is an important addition to hints because the AIF feedback helps students resolve uncertainty about whether they are on the right track, affirming their progress as they work, and giving them positive feedback. On the other hand, hints focus on what should be done next. Overall, these results reinforce the finding by Mitrovic *et al.* [12] that "a tutoring system that teaches to student errors can be improved by adding a capability to teach to their successes as well."

E. Student Perceptions

To gain insights into the students' *perspectives* on the AIF system, we examined students' responses to an open-ended question. Only 29 students took the optional survey, 20 in the *AIF* group and nine in the *Control* group. We performed thematic analysis (described in Section IV-C5) to analyze students' responses in the open-ended question: "how or when were the objectives' list with progress bars helpful or less helpful in Snap?" Our analysis resulted into two main themes: *mechanisms* by which the AIF system was helpful, and *limitations* in its design, which we report below. Note that when reporting quotes from students' responses, we put an anonymous student ID preceding their quote (e.g., [S1] means student 1).

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

2) Limitations in the AIF Design: Students' comments also revealed tradeoffs in the design of adaptive feedback, and ways the AIF system could have been improved. Five (out of 20)

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

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

VI. Discussion

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

A. H1-Performance—Supported

We found that AIF led to increased overall performance on tasks, reflected by more assignment objectives completed. These results show that the AIF system has the potential to increase students' grades *without* giving away any of the solution code (e.g., like hints may do). This performance increase is important because in introductory CS courses, students frequently make frequent, negative self-assessments of their programming ability that may lead students to leave the field [8]. AIF may help, not only by giving the student a sense of progress and accomplishment during programming, but also by securing more positive feedback from instructors when turning in more correct code. While this work did not measure students' affective outcomes, our own prior work suggests that adaptive immediate feedback can lead to increased intentions to persist in computing [14].

B. H2-Correctness—Supported

Our results suggest that, across tasks, students with AIF were more likely to turn in fully complete and correct code. Our case studies, surveys, and log data analyses suggest two likely mechanisms for this improvement. First, students who struggle on challenging independent homework tasks sometimes give up, like student *Em* did, without completing the task. Those with AIF may have been encouraged by the increasing progress bars and positive pop-up messages that show students they are "headed in the right direction [S2]". As shown in Fig. 6, students in the *AIF* group continued to work on the *Squiral* and *Daisy* homeworks, turning in correct code long after the *Control* students. Since the code was ultimately correct, it suggests that this extra time was well spent. Second, some students may have been unaware that their code failed to meet assignment objectives, as we discussed in *Em*'s case study, leading them to turn in erroneous code. The AIF system's automated assessment can highlight this incorrect code, giving students the opportunity to fix it. Fig. 6 shows that students *do* fix these errors, as evidenced by the gap between the percentage of students turning in fully correct code on the *Polygon Maker* and *Squiral* tasks over time. In both cases, our results show not only that the AIF system can encourage students to address these errors, but that students are often *capable of fixing them*, as long as they are made aware of them.

While our data cannot show whether students gave up intentionally or missed errors in their code, or how precisely the AIF changed their behavior, our hypothesis aligns with prior research and theory on the efficacy of formative feedback. Using the AIF system, students received immediate specific feedback, then corrected errors and turned in more complete code, which are behaviors that align both with instructor desires, and learning theories to improve learners' outcomes [21]. We also note that even in the rare occasions when the system's feedback is inaccurate, as noted in Section V-E, prompting students to engage in self-regulated learning skills, such as progress monitoring and self-assessment, can improve students' learning [43].

There are two explanations that may have led to improved completion (more scores of 100%) and correctness (i.e., higher scores) with AIF. First, the AIF system simplifies the learning process by dividing the task into a set of subgoals, and provides corrective immediate feedback on each. This task breakdown might have reduced students' cognitive load, and based on cognitive load theory, this could lead to an improvement in their scores [13]. Another interpretation is that, because students can see a change in their progress when they add or delete correct blocks, they might be more cautious not to delete correct code, which is an influence on students' "behavior," leading to higher scores. This was also suggested by students' responses in the postsurvey when one student said: "It lets me know what is needed to be done so I am not lost making mistakes and unnecessary edits [S4]."

C. H3-Learning—Inconclusive

We found suggestive evidence that students with AIF performed better on a subsequent transfer task *without* hints or feedback, as well as on a posttest. However, neither of these effects were significant. It is possible that our sample size of 50 consenting students was too small to detect this difference. It is also possible that the primary benefit of AIF is on the task where it is given, not on learning. However, we argue that if assignments are well-designed, encouraging students to complete more of the task objectives should reasonably have

some benefit to learning, as students who persist are exposed to more learning content.

D. Limitations

This article has three limitations: 1) survey data; 2) missing data; and 3) possible confounds. First, only 20 out of 23 students in the *AIF* group took the postsurvey and therefore our qualitative results may not be representative of all students in the *AIF* group. Second, while we suggest that the one mechanism by which the AIF system improves student performance and learning is by creating opportunities for positive self-assessment, we did not use surveys to measure self-assessment. This was due to some technical issues with the deployment of the surveys. However, students' responses in the postsurvey indicate that the AIF system may have promoted student self-assessment during programming, and our results in prior work show that such formative feedback improved students' intentions to persist in CS [14]. We plan to measure self-assessment and self-efficacy in future work.

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

VII. Conclusion

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

References

[1] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Trans. Comput. Educ.*, vol. 10, no. 4, 2010, Art. no. 16.

[2] D. Garcia, B. Harvey, and T. Barnes, "The beauty and joy of computing," *ACM Inroads*, vol. 6, no. 4, pp. 71–79, 2015.

[3] S. Cooper, W. Dann, and R. Pausch, "Alice: A 3-D tool for introductory programming concepts," *J. Comput. Sci. Colleges Consortium Comput. Sci. Colleges*, vol. 15, no. 5, pp. 107–116, 2000.

[4] L. Mannila, M. Peltomäki, and T. Salakoski, "What about a simple language? Analyzing the difficulties in learning to program," *Comput. Sci. Educ.*, vol. 16, no. 3, pp. 211–227, 2006.

[5] W. Dann, D. Cosgrove, D. Slater, D. Culyba, and S. Cooper, "Mediated transfer: Alice 3 to java," in *Proc. 43rd ACM Tech. Symp. Comput. Sci. Educ.*, 2012, vol. 12, pp. 141–146.

[6] W. Wang, C. Zhang, A. Stahlbauer, G. Fraser, and T. Price, "Snapcheck: Automated testing for snap programs," in *Proc. 26th ACM Conf. Innov. Technol. Comput. Sci. Educ.*, 2021, pp. 227–233.

[7] J. Denner and L. Werner, "Computer programming in middle school: How pairs respond to challenges," *J. Educ. Comput. Res.*, vol. 37, no. 2, pp. 131–150, 2007.

[8] J. Gorson and E. O'Rourke, "Why do CS1 students think they're bad at programming? Investigating self-efficacy and self-assessments at three universities," in *Proc. ACM Conf. Int. Comput. Educ. Res.*, 2020, pp. 170–181.

[9] Y. Dong, S. Marwan, V. Catete, T. Price, and T. Barnes, "Defining tinkering behavior in open-ended block-based programming assignments," in *Proc. 50th ACM Tech. Symp. Comput. Sci. Educ.*, 2019, pp. 1204–1210.

[10] P. Shabrina, S. Marwan, M. Chi, T. W. Price, and T. Barnes, "The impact of data-driven positive programming feedback: When it helps, what happens when it goes wrong, and how students respond," in *Proc. Educ. Data Mining Comput. Sci. Educ. Workshop*, 2020.

[11] V. J. Shute, "Focus on formative feedback," *Rev. Educ. Res.*, vol. 78, no. 1, pp. 153–189, 2008.

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

[13] F. Paas, J. E. Tuovinen, H. Tabbers, and P. W. Van Gerven, "Cognitive load measurement as a means to advance cognitive load theory," *Educ. Psychol.*, vol. 38, no. 1, pp. 63–71, 2003.

[14] S. Marwan, G. Gao, S. Fisk, T. Price, and T. Barnes, "Adaptive immediate feedback can improve novice programming engagement and intention to persist in computer science," in *Proc. Int. Comput. Educ. Res. Conf.*, 2020, pp. 194–203.

[15] M. Ball, "Lambda: An autograder for *snap*," Elect. Eng. and Comput. Sci. Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2018-2, 2018.

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

[17] L. Gusukuma, A. C. Bart, D. Kafura, and J. Ernst, "Misconception-driven feedback: Results from an experimental study," in *Proc. ACM Conf. Int. Comput. Educ. Res.*, 2018, pp. 160–168.

[18] S. Marwan, T. W. Price, M. Chi, and T. Barnes, "Immediate data-driven positive feedback increases engagement on programming homework for novices," in *Proc. Educ. Data Mining Comput. Sci. Educ. Workshop*, 2020.

[19] S. Marwan, Y. Shi, M. Chi, T. Barnes, and T. W. Price, "Just a few expert constraints can help: Humanizing data-driven subgoal detection for novice programming," in *Proc. Int. Conf. Educ. Data Mining*, 2021, pp. 68–80.

[20] S. Marwan, J. Jay Williams, and T. W. Price, "An evaluation of the impact of automated programming hints on performance and learning," in *Proc. ACM Conf. Int. Comput. Educ. Res.*, 2019, pp. 61–70.

[21] M. Thurlings, M. Vermeulen, T. Bastiaens, and S. Stijnen, "Understanding feedback: A learning theory perspective," *Educ. Res. Rev.*, vol. 9, pp. 1–15, 2013.

[22] M. C. Scheeler, K. L. Ruhl, and J. K. McAfee, "Providing performance feedback to teachers: A review," *Teacher Educ. Special Educ.*, vol. 27, no. 4, pp. 396–407, 2004.

[23] D. Fossati, B. Di Eugenio, S. Ohlsson, C. Brown, and L. Chen, "Data driven automatic feedback generation in the iList intelligent tutoring system," *Technol., Instruct., Cogn. Learn.*, vol. 10, no. 1, pp. 5–26, 2015.
[24] C. Watson, F. W. Li, and R. W. Lau, "Learning programming languages through corrective feedback and concept visualisation," in *Proc. Int. Conf. Web-Based Learn.*, 2011, pp. 11–20.
[25] T. Ashwell, "Patterns of teacher response to student writing in a multiple-draft composition classroom: Is content feedback followed by form feedback the best method?," *J. 2nd Lang. Writing*, vol. 9, no. 3, pp. 227–257, 2000.
[26] D. Wiliam, "Keeping learning on track: Formative assessment and the regulation of learning," *Rev. Process.*, vol. 20, pp. 1053–1098, 2005.
[27] F. M. Van der Kleij, R. C. Feskens, and T. J. Eggen, "Effects of feedback in a computer-based learning environment on students' learning outcomes: A meta-analysis," *Rev. Educ. Res.*, vol. 85, no. 4, pp. 475–511, 2015.
[28] A. Corbett and J. R. Anderson, "Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes," in *Proc. SIGCHI Conf. Hum. Comput. Interact.*, 2001, pp. 245–252. [Online]. Available: http://dl.acm.org/citation.cfm?id=365111
[29] B. Di Eugenio, D. Fossati, S. Ohlsson, and D. Cosejo, "Towards explaining effective tutorial dialogues," in *Proc. Annu. Meeting Cogn. Sci. Soc.*, 2009, pp. 1430–1435.
[30] M. R. Lepper, M. Woolverton, D. L. Mumme, and J. Gurtner, "Motivational techniques of expert human tutors: Lessons for the design of computer-based tutors," *Comput. Cogn. Tools*, vol. 1993, pp. 75–105, 1993.
[31] K. E. Boyer, R. Phillips, M. D. Wallis, M. A. Vouk, and J. C. Lester, "Learner characteristics and feedback in tutorial dialogue," in *Proc. 3rd Workshop Innov. Use NLP Building Educ. Appl.*, 2008, pp. 53–61.
[32] S. Marwan *et al.*, "Promoting students' progress-monitoring behavior during block-based programming," *Proc. 21st Koli Calling Int. Conf. Comput. Educ. Res.*, 2021, pp. 1–10.
[33] J. R. Anderson, A. T. Corbett, K. R. Koedinger, and R. Pelletier, "Cognitive tutors: Lessons learned," *J. Learn. Sci.*, vol. 4, no. 2, pp. 167–207, 1995.
[34] I.-H. Hsiao, S. Sosnovsky, and P. Brusilovsky, "Guiding students to the right questions: Adaptive navigation support in an E-learning system for java programming," *J. Comput. Assist. Learn.*, vol. 26, no. 4, pp. 270–283, 2010.
[35] D. E. Johnson, "Itch: Individual testing of computer homework for scratch assignments," in *Proc. 47th ACM Tech. Symp. Comput. Sci. Educ.*, 2016, pp. 223–227.
[36] T. W. Price, Y. Dong, and D. Lipovac, "iSnap: Towards intelligent tutoring in novice programming environments," in *Proc. ACM Tech. Symp. Comput. Sci. Educ.*, 2017, pp. 483–488.
[37] W. Wang, R. Zhi, A. Milliken, N. Lytle, and T. W. Price, "Crescendo: Engaging students to self-paced programming practices," in *Proc. 51st ACM Tech. Symp. Comput. Sci. Educ.*, 2020, pp. 859–865.
[38] L. E. Margulieux, R. Catrambone, and M. Guzdial, "Employing subgoals in computer programming education," *Comput. Sci. Educ.*, vol. 26, no. 1, pp. 44–67, 2016.
[39] J. Sweller, "Cognitive load during problem solving: Effects on learning," *Cogn. Sci.*, vol. 12, no. 2, pp. 257–285, 1988.
[40] R. Moreno and R. E. Mayer, "Cognitive principles of multimedia learning: The role of modality and contiguity," *J. Educ. Psychol.*, vol. 91, no. 2, 1999, Art. no. 358.
[41] T. W. Price, S. Marwan, and J. J. Williams, "Exploring design choices in data-driven hints for python programming homework," in *Proc. 8th ACM Conf. Learn., Scale*, 2021, pp. 283–286.
[42] S. Marwan, N. Lytle, J. J. Williams, and T. Price, "The impact of adding textual explanations to next-step hints in a novice programming environment," in *Proc. ACM Conf. Innov. Technol. Comput. Sci. Educ.*, 2019, pp. 520–526.
[43] D. Loksa and A. J. Ko, "The role of self-regulation in programming problem solving process and success," in *Proc. Int. Comput. Educ. Res. Conf.*, 2016, pp. 83–91.
[44] E. Cech, B. Rubineau, S. Silbey, and C. Seron, "Professional role confidence and gendered persistence in engineering," *Amer. Sociol. Rev.*, vol. 76, no. 5, pp. 641–666, 2011.
[45] R. Zhi, T. W. Price, N. Lytle, Y. Dong, and T. Barnes, "Reducing the state space of programming problems through data-driven feature detection," in *Proc. Educ. Data Mining Comput. Sci. Educ. Workshop*, 2018, pp. 1–5.
[46] D. Weintrop and U. Wilensky, "Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs," in *Proc. 11th Annu. Int. Conf. Int. Comput. Educ. Res.*, 2015, vol. 15, pp. 101–110.
[47] J. Bruin, "Introduction to linear mixed models," Accessed: Apr. 6, 2020. [Online]. Available: https://stats.idre.ucla.edu/stata/ado/analysis/
[48] P. V. Hippel, "Linear vs. logistic probability models: Which is better, and when," in *Proc. Stat. Horiz.*, 2015.
[49] M. Maguire and B. Delahunt, "Doing a thematic analysis: A practical, step-by-step guide for learning and teaching scholars," *Ireland J. Teach. Learn. Higher Educ.*, vol. 9, pp. 3351–3365, 2017.
[50] S. Skrivanek, "The use of dummy variables in regression analysis," More Steam, LLC, Powell, OH, USA, 2009.

Samiha Marwan received the Ph.D. degree in computer science from NC State University, Raleigh, NC, USA, in 2021.

She is currently a Postdoctoral Researcher and a CI-Fellow with the University of Virginia, Charlottesville, VA, USA. Her primary research interests include developing intelligent support features in block-based programming languages to improve novices' cognitive and affective outcomes.

Bita Akram received the Ph.D. degree in computer science from NC State University, in 2019.

She is currently a Teaching Assistant Professor with the Department of Computer Science, NC State University, Raleigh, NC, USA. Her research interests include the intersection of artificial intelligence and advanced learning technologies with its application on improving access and quality of CS education.

Tiffany Barnes received the B.S. and M.S. degrees in computer science and Mathematics, and the Ph.D. degree in computer science from NC State University, in 2003.

She is currently a Distinguished Professor of Computer Science with NC State University, Raleigh, NC, USA, and the Codirector for the STARS Computing Corps, Philadelphia, PA, USA. Her research interests include AI for education, educational data mining, CS education, and broadening participation in computing education and research.

Dr. Barnes was the recipient of an NSF CAREER Award for her novel work in using data and educational data mining to add intelligence to STEM learning environments.

Thomas W. Price received the M.S. and Ph.D. degrees in computer science from NC State University, in 2018.

He is currently an Assistant Professor of Computer Science with NC State University, Raleigh, NC, USA. He directs the Help through INTelligent Support (HINTS) Lab, which develops learning environments that automatically support students through AI and data-driven help features.

Dr. Price has been recognized by the STARS Computing Corps, Philadelphia, PA, USA, for his leadership in computing outreach.