iSnap: Towards Intelligent Tutoring in Novice Programming Environments

Thomas W. Price
North Carolina State Univ.
Raleigh, NC, USA
twprice@ncsu.edu

Yihuan Dong
North Carolina State Univ.
Raleigh, NC, USA
ydong2@ncsu.edu

Dragan Lipovac
North Carolina State Univ.
Raleigh, NC, USA
dlipova@ncsu.edu

ABSTRACT

Programming environments intentionally designed to support novices have become increasingly popular, and growing research supports their efficacy. While these environments offer features to engage students and reduce the burden of syntax errors, they currently offer little support to students who get stuck and need expert assistance. Intelligent Tutoring Systems (ITSs) are computer systems designed to play this role, helping and guiding students to achieve better learning outcomes. We present iSnap, an extension to the Snap programming environment which adds some key features of ITSs, including detailed logging and automatically generated hints. We share results from a pilot study of iSnap, indicating that students are generally willing to use hints and that hints can create positive outcomes. We also highlight some key challenges encountered in the pilot study and discuss their implications for future work.

Keywords

Novice programming, Intelligent Tutoring Systems, Hints, Data-driven, Logging, Snap

1. INTRODUCTION

Programming environments intentionally designed to support novices as they learn computing have become increasingly popular in recent years, in both formal and informal learning contexts. These novice programming environments (NPEs), such as Alice [10], Greenfoot [7], Scratch [15] and Snap*!* [4], have thousands of users and often constitute a learner's first interaction with computing. Empirical studies have shown that these environments and their associated curricula offer benefits over traditional instruction, such as increased retention [10], improved test scores [3] and more effective programming editors [13].

While NPEs can lessen the initial difficulties of programming and allow for more interesting assignments, they currently offer little support to students who get stuck and do not know how to proceed with an assignment. This work still largely falls to instructors, who are not always available to help. Intelligent Tutoring Systems (ITSs) are computer systems designed to approximate the role of a human tutor, using artificial intelligence techniques to support students as they work on challenging assignments. ITSs have shown much promise in the domain of computer programming, with students using an ITS performing as much as two standard deviations higher than those who received conventional instruction [1]. A key feature of any ITS is the ability to give students individualized feedback during problem solving, often in the form of hints. In the domain of programming, this feedback has been shown to improve students' performance, both inside the tutor and on subsequent assessments [2].

Unfortunately, little effort has been made to bring the intelligent features of ITSs to NPEs. This is due in part to the large investment of time required by domain experts to create ITSs, which can be as much as 300 hours to create one hour of intelligent content [11]. Further, the use of open-ended programming assignments, which makes these environments so appealing to students and teachers [4], also makes providing intelligent, adaptive feedback difficult.

In this paper, we present an extension to the Snap*!* NPE called iSnap, which offers students on-demand, next-step hints. These hints are automatically generated using a data-driven algorithm designed to overcome the challenges presented by NPEs. It uses previous students' submissions for an assignment to advise new students how to proceed. We evaluated iSnap with a pilot study of students in an introductory university computing course for non-majors. We present a brief overview of how students used hints in iSnap, as well as a case study of one student. We then discuss some challenges brought to light by the study, which inform recommendations on future work integrating features of ITSs into NPEs.

SIGCSE '17, March 8–11, 2017, Seattle, WA, USA.
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4698-6/17/03…$15.00
DOI: http://dx.doi.org/10.1145/3017680.3017762

2. RELATED WORK

2.1 Novice Programming Environments

Novice programming environments (NPEs) broadly encompass those environments with features specifically designed to aid novices in learning to program. We focus here primarily on popular NPEs which have emerged in the past decade. While NPEs are diverse, they generally share the goals of creating a more engaging context for programming and reducing the initial challenges of programming syntax.

NPEs attempt to foster engagement by connecting programming to students' interests, such as games, stories and simulations [18]. For example, Maloney et al. [8] observed that students in an urban after-school center willingly used Scratch to create hundreds of games, music videos and animations, and found that students formed peer mentoring communities around the design of their programs.

NPEs address the challenge of programming syntax in a variety of ways, such the detailed structural highlighting that is offered in recent versions of Greenfoot [7]. Many NPEs, such as Scratch [15], Alice [3] and Snap*!* [4], avoid syntax errors altogether by using drag-and-drop, block-based programming. These NPEs use blocks and menus to represent code elements, and any configuration of these blocks represents valid syntax. Price and Barnes [13] compared novices using a block-based and textual programming interface and found that the block group completed significantly more objectives in less time and spent significantly less time idle compared to the text group.

Empirical evaluations show that introductory CS courses based on NPEs are an effective way to teach novices. Meerbaum-Salant et al. [9] found ninth-grade students significantly improved on most CS concepts after taking a weekly Scratch course for one semester, though students did struggle with more abstract concepts such as initialization, variables and concurrency. Dann et al. [3] used Alice 3 in an introductory undergraduate CS course, and transitioned from Alice's original block interface to a Java implementation of the Alice API. The authors compared students' final exam scores (on a Java-based assessment) with those of the previous, all-Java version of the course and found that the Alice classes performed over one letter grade higher on average than the previous class. Work by Price et al. [12] suggests that the BJC curriculum designed around Snap*!* is not only well-liked, but also appeals equally to students from diverse backgrounds.

2.2 Intelligent Tutoring Systems

Intelligent Tutoring Systems (ITSs) are computer systems that attempt to provide the benefits of individualized human tutoring [1], often by guiding students through a series of problem solving tasks, dynamically adapting instruction to individual students [11]. For example, the ACT Programming Tutor (APT) [2] offers next-step hints to students working on short programming exercises in Lisp. It generates these hints using a technique called model tracing, which defines correct programming behavior using a set of expert-defined, if-then "production rules." Ask-Elle [5] is a Haskell tutor that augments traditional model-tracing with property-based testing, a sort of automated unit testing framework, to help the student identify errors. More recently, data-driven tutors, such as ITAP [16], have shown that programming hints can be effectively constructed automatically from previous students' data. Using data from an introductory Python course, ITAP was able to construct a set of hints leading to a solution for 98% of incorrect solution attempts.

There is empirical evidence to suggest that tutor hints do provide a benefit to students. Stamper et al. [17] compared two versions of the Deep Thought logic tutor, one with on-demand hints and one without, across two semesters and found that the Hint group completed significantly more of the tutor, had significantly less dropout and had a significantly higher final course grade than the control group. In the domain of programming, Corbett et al. [2] compared a variety of help mechanisms in the APT, including immediate feedback, on-demand hints and debugging feedback, and found that students who received feedback during tutoring completed the tutor faster and completed a subsequent programming assessment in significantly less time, with significantly fewer errors. Holland et al. [6] evaluated their J-LATTE ITS in a small study comparing a version with and without error feedback. The feedback group completed significantly more problems than the control group, while the control group progressed through the tutor quicker, viewing significantly more problems but spending significantly less total time in the tutor.

3. iSNAP

iSnap is an extension of the Snap*!* [4] programming environment. Snap*!* is based on Scratch and adds new features, such as higher order functions, a web API and accessibility features. Snap*!* is most notable for its tight integration with the Beauty and Joy of Computing curriculum [4], an AP Computer Science Principles course that focuses on creating interesting computational artifacts through creative and collaborative assignments. As a first step towards developing a full programming ITS, iSnap extends Snap*!* with detailed logging and on-demand hints.

iSnap logs all student actions to a remote database, including any interactions with the user interface and coding area. It also logs complete snapshots of students' code after each edit, allowing for complete replay of a student's actions within the environment. Each entry in iSnap's logs consists of an event type (e.g. "block grabbed", "project exported"), any associated data (e.g. an ID for the grabbed block) and an updated snapshot of the student's code (if changed). While this data could have many uses, we use it specifically to automatically generate contextual hints.

3.1 On-demand Hints

Hints in iSnap are generated using the Contextual Tree Decomposition (CTD) algorithm [14]. CTD provides hints by matching a student's current code to prior student solutions to recommend edits based on prior student actions. CTD takes as input a set of student attempts at the assignment for which hints will be generated, with sequential snapshots of each student's code from beginning to submission. CTD converts each snapshot into an Abstract Syntax Tree (AST) and independently models how students edit the children of each node in the AST. This tree decomposition allows easier matching between student solutions – a key feature for hint generation with open-ended assignments.

When a student requests help, CTD identifies nodes in their current AST that match those from historical data and, for each one, computes the edit to its children that is most likely to result in a correct solution. This results in a set of hints, one for each AST node that requires an edit. The hints update dynamically as a student edits their code and can support multiple solutions to an assignment. CTD hints always suggest an edit to the children of a single node in an AST and therefore cannot suggest changing how code elements are nested. An initial technical evaluation of the CTD algorithm on historical data showed that it was able to generate hints for all observed student snapshots and that, compared to other hints policies, applying these hints would help complete more assignment objectives and move students closer to their final solutions [14].

The main addition that iSnap makes to the interface of

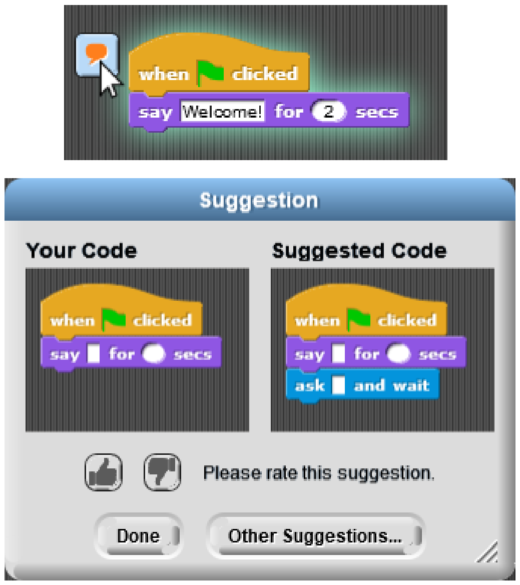


Figure 1: A student selects a 'hint bubble' (above), which causes a script hint to display (below), suggesting the addition of an 'ask' block.

Snap*!* is a Help button. When a student presses this button, iSnap annotates their code with "hint bubbles," indicating where hints are available. When a student hovers over a hint bubble, part of the student's code is highlighted to show where that hint applies. By presenting all available hints, we allow the student to choose where to focus the help request, though we recognize that this may be challenging for students who are unsure where they *need* help.

If the student clicks a hint bubble, all bubbles disappear and the selected hint is displayed, showing the student's current code, next to the code suggested by the hint, as shown in Figure 1. Script hints suggest a modification to a script (a vertical chain of blocks). Block hints suggest modifying a block's parameters. Both scripts and blocks can be nested inside of other blocks, and in this case the hint dialog shows only the nested code to the student. Hints generally suggest a single edit to a student's code, such as the insertion, deletion or movement of a block of code. This hint display strips out all information not directly relevant to the hint. For example, the script hint shown in Figure 1 encourages the student to add an 'ask' block to their main script, but the parameters of this block are left blank. This gives the student some responsibility for implementing the hint by filling in relevant parameters.

4. METHODS AND ANALYSIS

We conducted a pilot study of iSnap with students in an introductory CS course for non-majors at a research university during the Spring 2016 semester. The first half of the course focused on learning the Snap*!* programming language through a curriculum based on the Beauty and Joy of Computing (BJC) [4].

Hints were available on one graded in-lab assignment, the "Guessing Game." In this assignment, students were asked to create a program that stores a random number and then repeatedly asks the player to guess it until they are correct, informing them if they have guessed too high or too low. The assignment required the use of loops, conditionals, variables and conditional operators. The hints used in this study were generated using 32 fully correct student submissions of the Guessing Game, collected during the Fall 2015 semester of the same course. Before students started the assignment, a researcher briefly introduced the Help button and hint interface, explaining that the students were encouraged to use hints without any penalty to their grades but that the hints may not be perfect. Students worked on the Guessing Game assignment during class for approximately one hour.

We collected 63 attempts at the Guessing Game with hints available, consisting of 10753 total code snapshots[1]. Only 5% of snapshots were found in more than one student's submission. Two researchers manually graded the solutions based on 9 objectives taken from the instructions, achieving 99.8% agreement (Cohen's $\kappa = 0.99$). Submissions achieved on average 91.5% of objectives (SD = 16.1%), with 40 submissions (63.5%) achieving all objectives. Grades are likely inflated due to the presence of TAs, who were able to help struggling students to finish the assignment.

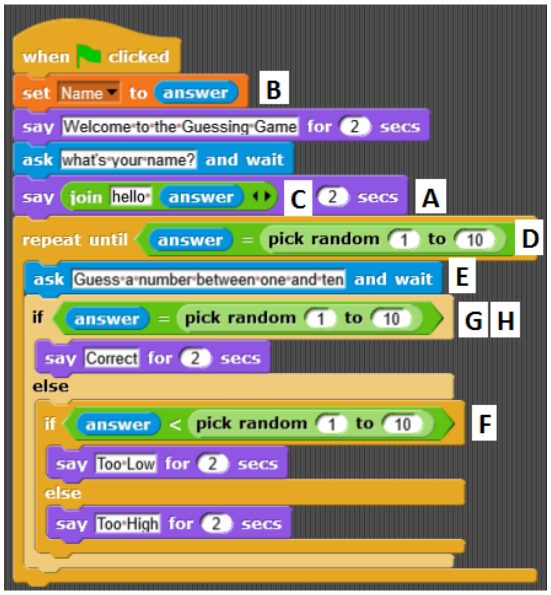
We first performed a number of high-level aggregate analyses of the data to better understand how students used hints. Afterwards we manually inspected the log data of each student who requested hints (n=34) to find common trends. The high-level analysis shows that the 63 students requested an average of 4.95 hints each, with the number of hints ranging from 0 to 62 hints per student. We consider the student attempt with 62 hints an outlier and omit it from analysis. Of the remaining 62 students, 33 (53.2%) made at least 1 hint request, 23 (37.1%) made at least 3 hint requests, and 2 students requested a maximal 30 hints. Students followed approximately half (49.4%) of the 312 total hints requested within the next 5 edits, indicating that they were generally well-received. Interestingly, students were 39.2% more likely to follow hints that would immediately complete an assignment objective – following 44 (68.8%) of the 64 objective-achieving hints received. This suggests that students may need more support in understanding the long-term goals behind hints, as we discuss in more detail in Section 6.

Of the 62 students, 21 followed at least 1 hint of the hints they requested, and 13 followed at least two hints. The five (8.1%) students who failed the assignment (achieving less than half of the objectives), followed either 0 or 1 hints. The 13 students who followed more than 1 hint achieved at least 7 of the 9 assignment objectives, with 12 students achieving 8-9 objectives. This suggests that hints, when followed, may serve their intended role of scaffolding students, but there is insufficient variance in grades to verify this claim statistically. The median student achieved all assessment objectives.

5. HINTS IN ACTION

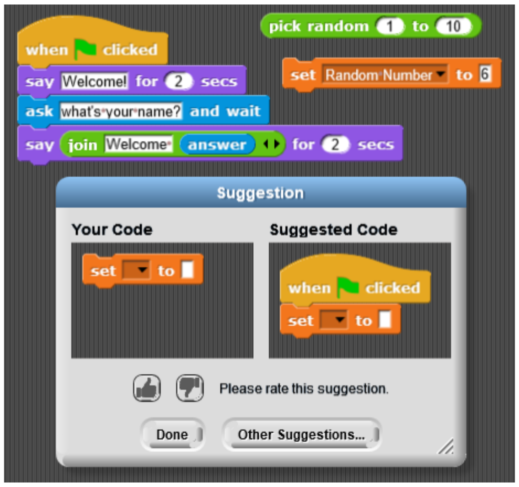
We now present a case study of how one student (whom we will call Anna) worked with iSnap hints. Anna requested a total of 12 hints throughout her work and followed 9 of them. A snapshot of Anna's work near completion is shown in Figure 2, with letters labelling where she receives hints. Anna starts her project by welcoming the player and asking for the player's name as specified in the lab objectives, but

[1]Datasets and demo available at http://go.ncsu.edu/isnap



```
when clicked
set Name to answer
say Welcome to the Guessing Game for 2 secs
ask what's your name? and wait
say join hello answer 2 secs
repeat until answer = pick random 1 to 10
    ask Guess a number between one and ten and wait
    if answer = pick random 1 to 10
        say Correct for 2 secs
    else
        if answer < pick random 1 to 10
            say Too Low for 2 secs
        else
            say Too High for 2 secs
```

Figure 2: Anna's code near the completion of her program. Letters mark where Anna followed hints.



```
pick random 1 to 10

when clicked
say Welcome! for 2 secs
ask what's your name? and wait
say join Welcome answer for 2 secs

set Random Number to 6
```

Suggestion

Your Code

```
set to
```

Suggested Code

```
when clicked
set to
```

Please rate this suggestion.

Figure 3: An example of a hint for a "test script" that recommends modifying it like a main script.

gets stuck trying to greet the player by name. She first adds, and later deletes, a 'say' block. She then asks for a script hint, which directs her to add a 'say' block (A). This confirms her original idea, and she follows the hint. She asks for another script hint, which directs her to store a variable at the beginning of her script (B). Anna names this variable 'Name,' presumably because she thinks it is addressing her current goal of greeting the player; however, in reality the hint is addressing the later goal of storing a random number. Having not yet finished greeting the player by name, Anna requests another hint, this time getting a block hint for the 'say' block she added previously. This hint directs her to add a 'join' block (C), which she does. Once she has found the 'join' block, Anna independently fills in its parameters to greet the player by name.

Anna proceeds on her own, adding a needed 'repeat-until' loop with a 'say' block inside, to address the goal of repeatedly asking the player for a guess. Anna then requests a block hint for the 'repeat-until' loop condition, which directs her to add an equals comparison (D). Before acting, Anna also asks for a script hint for the body of the same loop, which directs her to replace her 'say' block with an 'ask' block (E). She proceeds to follow both hints, but fills in their parameters on her own. For the equals loop condition, she mistakenly compares the player's answer to a newly generated random number each time, rather than comparing it to a single, stored variable. Anna then adds an 'if-else' block to the loop body, and requests a script hint for this block, which directs her to nest a second 'if-else' block in the else body (F). She requests a block hint for the first 'if-else' block as well (G), which directs her to use an equals comparison as the condition. She does so, and then figures out the condition for the second 'if-else' block on her own. For each of these comparisons, she makes the same mistake as in her loop condition, comparing the player's answer to a new random number. She requests a hint for the first if-else comparison (H), which directs her to replace this random number with a variable. She independently creates and properly instantiates this variable, and follows the hint. She then makes the same correction for the other three comparisons on her own. Now with a complete program, she asks for hints one last time. Only two hint bubbles are displayed, indicating that she is nearly done. She chooses not to click a hint bubble on her 'if-else' block, but does select a hint telling her she may have too many variables. She follows this hint, deleting the 'Name' variable she created earlier.

Anna shows how hints can be used to overcome challenges in the Guessing Game assignment that otherwise would require an instructor. Anna requests help when she is stuck but uses the help to complete more of the assignment than the hint covers, and shows the ability to generalize ideas. While some might argue she used help too much, her work demonstrates several positive examples of hint usage.

6. CHALLENGES

In the previous section, we showed how iSnap's hints can play their intended role of helping a student progress through the assignment. Here, we present some of the more serious challenges encountered in this pilot study, including students' bottom-up programming behavior, over-reliance on help, and difficulty understanding the ideas behind hints. It is important to remember that the interactions described here represent students' first exposures to the hint system. Many of these challenges might be remedied if students are given more instruction on how the hints work and how to use them effectively.

6.1 Bottom-up Programming

Scratch (and by extension Snap*!*) was designed to facilitate bottom-up programming and tinkering [15]. ITSs, by contrast, often impose a top-down approach to problem solving. The challenge of reconciling these two approaches played out in our pilot study of iSnap. Students frequently constructed parts of their programs in "test scripts," which may consist of a single block with parameters, or a part of the main script that has been split apart for refinement. Students did not request hints too frequently for these test scripts, but when they did they never followed the hints.

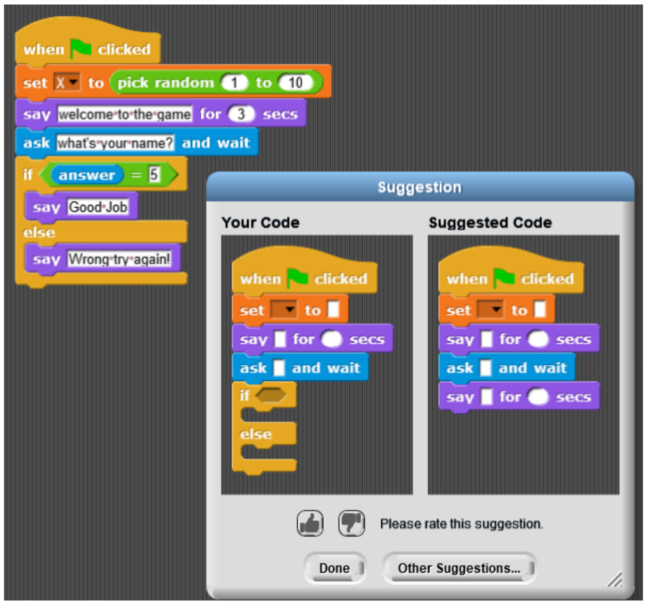

```
when [flag] clicked
set X to pick random 1 to 10
say welcome to the game for 3 secs
ask what's your name? and wait
if answer = 5
    say Good Job
else
    say Wrong try again!
```

Figure 4: An example of a hint to delete an 'if' block that is outside of a loop.

This was not surprising, since the CTD algorithm suggests hints that grow these test scripts into a correct main script, but students were more likely seeking help on how to expand the code fragment or integrate it into their main script. Figure 3 shows an example, where a student added a variable assignment block and asked for a script hint. The hint makes a recommendation to add a 'when-flag-clicked' block before the variable assignment block, since this is present in most correct main scripts.

Even when students were working with a single script, they still seemed to develop their programs incrementally, building partial functionality first and improving it over time. For example, students commonly added an 'if' block to their main script, which informed the player if their guess was correct or not, before creating the loop to repeat the guessing procedure. Figure 4 shows one student who did this and then asked for a script hint. The hint recommended removing the 'if-else' block from this script, since a correct solution would not put the if-else outside of a loop. While this hint is arguably "correct," it considers only the final program structure, rather than the particular process a student might use to get there.

6.2 Reliance on Help

An important concern when adding a help feature to a learning environment is whether or not that help will be used appropriately by students. In this pilot study, we observed a relatively low incidence of help abuse in iSnap. It is difficult to say what an appropriate number of hints is, since students may look through multiple hints to find one that they would like to use. However, we did find that 4 of 63 students (6.3%) followed more than 10 hints, with one student requesting 62 total hints. These students each exhibited clear over-reliance on hints, repeatedly requesting hints and then following them exactly. We also observed a strong correlation between the number of hints a student requested and the percentage of these hints which the student

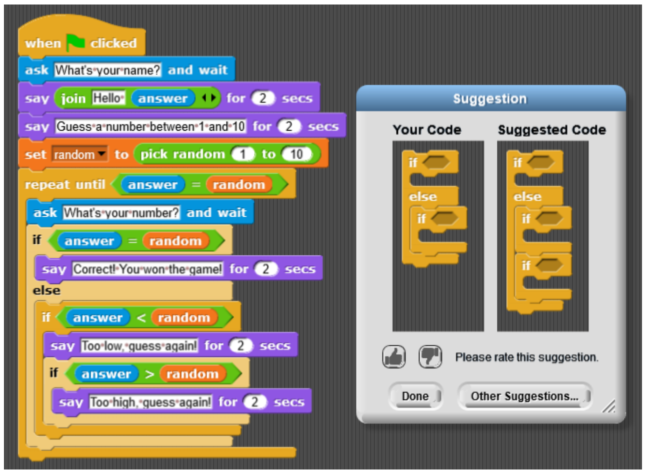

```
when [flag] clicked
ask What's your name? and wait
say join Hello answer for 2 secs
say Guess a number between 1 and 10 for 2 secs
set random to pick random 1 to 10
repeat until answer = random
    ask What's your number? and wait
    if answer = random
        say Correct! You won the game! for 2 secs
    else
        if answer < random
            say Too low, guess again! for 2 secs
            if answer > random
                say Too high, guess again! for 2 secs
```

Figure 5: A hint to properly nest an 'if' block that was not followed.

followed ($r = 0.785$; $p < 0.001$), meaning those students who asked for more help were more likely to follow it.

Even among students who used the most help, however, there were still examples of students taking initiative once they understood how to proceed. For example, one student, whom we will call Bob, relied on hints to complete the beginning of the assignment, including his first use of an 'if' block with an equals comparison. However, Bob then correctly constructed the next two 'if' blocks without hints, perhaps referring to the original hint-based code as an example. Next, Bob made the common mistake of incorrectly nesting 'if' blocks with mutually exclusive conditions, but then corrected the code without using hints. This indicates that despite his initial reliance on hints, Bob was still able continue the assignment independently without them.

6.3 Understanding Hints

Data-driven hints do not rely on experts to construct an explicit model of domain content and can therefore be generated easily for a variety of assignments. However, without an expert, these hints are limited to telling a student *what* to do, not *why* the advice should be followed. We saw this limitation in our data when students received hints which were correct, leading the student closer to a solution or correcting errors, but were not followed. For example, multiple students made the error of placing statements in unnecessary 'if' blocks with trivially true conditions. One student asked for a script hint that instructed them to replace this 'if' block with the 'say' block it contained. However, the student chose not to follow this hint, keeping the unneeded 'if' block until the end, which suggests the student did not understand what the hint was communicating.

Another fairly common mistake was nesting two 'if' blocks with mutually exclusive conditions. One student who made this mistake requested a hint, shown in Figure 5, which directed the student to add an 'if' block in the correct location. However, the student ignored this advice and ended up submitting the flawed program. Unlike in the previous example, this problem cost the student points on their final grade. Admittedly, this hint requires the student to connect the addition of a second 'if' block, as suggested by the hint, to their improperly nested 'if' block, which might be difficult for an already-struggling student to do. These examples suggest that even reasonable and correct hints may be ineffective without adequate explanation.

7. CONCLUSIONS AND FUTURE WORK

We have presented iSnap, a first step towards integrating data-driven intelligent help into a NPE. We have shown that most students were willing to ask for hints, and that approximately half of all hints requested were also followed. We presented the work of one student, Anna, as an example of productive hint use but also identified a number of challenges that must be addressed for data-driven hints to be viable in NPEs.

While this pilot study represents a promising start for integrating data-driven ITS features into NPEs, the study revealed several remaining challenges for the CTD algorithm and the presentation of iSnap hints. We have already partially addressed the challenge of bottom-up programming by modifying the CTD algorithm to recognize when smaller pieces of code may belong inside of an existing main script, in which case it suggests combining the scripts together. We can prevent overreliance on hints by a (possibly adaptive) limit on the total number of hints and the number of consecutive hints, though we should be careful not to prevent possible frequent but productive requests, like those made by Anna. Future work will also investigate how to improve student adoption of hint suggestions. There are several explanations for not following hints – it could be a lack of understanding, a lack of trying, or simply that the student had another intention or belief about what the hint would show. Additionally, a larger-scale study of hint usage could yield useful data for improving the CTD algorithm itself.

Even with these proposed improvements, we recognize that next-step hints will not satisfy all students' help requests. For instance, we found 10 instances of hint requests on nearly-complete programs, suggesting that students were looking for help checking their work and identifying possible errors before submitting. In most cases, the hints did not serve this function and were not followed, indicating a need for "correctness" feedback and error-specific feedback, which are commonly found in ITSs. We intend to adapt the CTD algorithm to generate this feedback, not only to help students check their work when finished, but also to offer more high-level guidance to keep them on track as they work. Rather than offering a single suggested edit, our planned error feedback will highlight a student's whole program, using different colors to suggest code that should be kept, moved, deleted and inserted, without providing explicit instruction on how these might be accomplished.

It is our hope that future work will go beyond the CTD algorithm and iSnap. While this work has focused on how to offer help on a given problem, we believe that additional intelligent features from ITSs could be adapted for NPEs. Student modelling and mastery learning [1], for example, could be used to tailor programming environments to individual students based on models of their cognitive and affective states.

8. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under grants 1432156 and 1623470.

9. REFERENCES

[1] A. Corbett. Cognitive Computer Tutors: Solving the Two-Sigma Problem. In *Proc. 8th Int. Conf. on User Modeling*, pages 137–147, 2001.

[2] A. Corbett and J. Anderson. Locus of Feedback Control in Computer-Based Tutoring: Impact on Learning Rate, Achievement and Attitudes. In *SIGCHI'01*, pages 245–252, 2001.

[3] W. Dann, D. Cosgrove, and D. Slater. Mediated Transfer: Alice 3 to Java. In *Proc. of ACM SIGCSE'12*, pages 141–146, 2012.

[4] D. Garcia, B. Harvey, and T. Barnes. The Beauty and Joy of Computing. *ACM Inroads*, 6(4):71–79, 2015.

[5] A. Gerdes, B. Heeren, J. Jeuring, and L. T. van Binsbergen. Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback. *IJAIED*, pages 1–36, 2016.

[6] J. Holland, A. Mitrovic, and B. Martin. J-LATTE: a Constraint-based Tutor for Java. In *Proc. Int. Conf. on Computers in Education*, pages 142–146, 2009.

[7] M. Kölling. The Greenfoot Programming Environment. *ACM TOCE*, 10(4), nov 2010.

[8] J. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk. Programming by choice: urban youth learning programming with scratch. *ACM SIGCSE Bulletin*, 40(1):367–371, 2008.

[9] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari. Learning Computer Science Concepts with Scratch. In *Proc. of ICER'10*, pages 69–76, 2010.

[10] B. Moskal, D. Lurie, and S. Cooper. Evaluating the Effectiveness of a New Instructional Approach. *ACM SIGCSE Bulletin*, 36(1):75–79, 2004.

[11] T. Murray. Authoring Intelligent Tutoring Systems: An Analysis of the State of the Art. *IJAIED*, 10:98–129, 1999.

[12] T. W. Price, J. Albert, V. Catete, and T. Barnes. BJC in Action: Comparison of Student Perceptions of a Computer Science Principles Course. In *Proc. of RESPECT'15*, 2015.

[13] T. W. Price and T. Barnes. Comparing Textual and Block Interfaces in a Novice Programming Environment. In *Proc. of ICER'15*, 2015.

[14] T. W. Price, Y. Dong, and T. Barnes. Generating Data-driven Hints for Open-ended Programming. In *Proc. of EDM'16*, 2016.

[15] M. Resnick, J. Maloney, H. Andrés, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for All. *Communications of the ACM*, 52(11):60–67, 2009.

[16] K. Rivers and K. R. Koedinger. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *IJAIED*, 16(1), 2015.

[17] J. Stamper, M. Eagle, T. Barnes, and M. Croy. Experimental Evaluation of Automatic Hint Generation for a Logic Tutor. *IJAIED*, 22(1):3–17, 2013.

[18] I. Utting, S. Cooper, and M. Kölling. Alice, Greenfoot, and Scratch – A Discussion. *ACM TOCE*, 10(4), 2010.