ABSTRACT

PRICE, THOMASON WILLIAM. iSnap: Data-driven Support for Novice Programming Informed by Evaluations of Hint Quality and Investigations of Student Help-seeking Behavior. (Under the direction of Tiffany Barnes.)

Growing national interest in Computer Science (CS) education has fueled efforts to recruit a larger and more diverse group of computing students. These efforts frame programming as a way to create meaningful, creative and impactful artifacts. This approach has resonated with learners, and the number of undergraduate CS majors has tripled in the past decade. However, CS can be a notoriously unforgiving subject, with the highest 4-year college attrition rate of any STEM field and CS1 pass rates at 60-70%. CS students are struggling and need additional support that can scale to modern computing classrooms, especially when instructors are unavailable.

I present research that addresses these challenges through the development and evaluation of iSnap, a novice programming environment that automatically supports students with next-step hints and feedback. iSnap's help is generated using SourceCheck, a data-driven algorithm that leverages the correct solutions of previous students to guide new students towards a correct solution. SourceCheck is also designed to support longer and more open-ended programming tasks, which are popular in computing curricula for novices. iSnap integrates these help features with a block-based programming interface, which allows students to construct programs with drag-and-drop blocks of code, minimizing the challenges of syntax.

In this work I investigate four research questions, concerning: 1) the design of the iSnap programming environment; 2) the generation of data-driven programming hints; 3) how to evaluate the quality of these data-driven hints; and 4) what factors influence students' willingness to seek and use this help when programming. I present three evaluation studies of iSnap, which address these questions. I find that iSnap's hints are relatively high-quality, often matching human-authored hints, but that their quality can also vary considerably. My results show that when students receive lower-quality hints, they are less likely to ask for additional help, even if they are struggling on an assignment. I present methods for benchmarking and comparing data-driven hint quality, so that improvements to hint generation algorithms can be more easily evaluated. Using these methods, I show that the quantity and source of data significantly impact the SourceCheck algorithm's hint quality. However, I also find that many students never ask for help in the first place and that a variety of factors besides hint quality impact students' willingness to seek help. These include students' desire for independence, their previous experiences with computer-based help, and the accessibility and salience of the help.

The contributions of this work include the iSnap programming environment and the SourceCheck data-driven hint generation algorithm, along with design recommendations for how to
effectively integrate intelligent, data-driven support into a novice programming environment. Additionally, it contributes methods for evaluating the quality of data-driven hints and evidence linking this hint quality to students’ help-seeking outcomes. Lastly, it identifies additional factors that influence students’ help-seeking behavior and translates these factors into design recommendations for more effective programming support features. Each of these contributions inform future directions for research on data-driven programming support, as it progresses beyond the next-step hints discussed in this work to support more complex and open-ended programming assignments.
iSnap: Data-driven Support for Novice Programming Informed by Evaluations of Hint Quality and Investigations of Student Help-seeking Behavior

by
Thomason William Price

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Computer Science
Raleigh, North Carolina
2018

APPROVED BY:

______________________________
James Lester II

______________________________
Sarah Heckman

______________________________
Roger Azevedo

______________________________
Tiffany Barnes
Chair of Advisory Committee
Thomason (Thomas) Price was born in Chapel Hill, N.C., to a family that encouraged his curiosity and questioning. He quickly found that computers offered him an unrivaled source of exploration and creativity, and he spent many of his formative years stumbling through the process of learning to program. Thomas attended Elon University to study Computer Science, where he grew to understand the level of privilege that he shared with many of his fellow students, which had allowed him the time and freedom to come to love programming on his own. Thomas chose to focus his undergraduate thesis on mobile game development to promote STEM education, with the hope that educational technology could give more students that same experience of finding their passion in an academic field. This experience led Thomas to pursue a Ph.D. at North Carolina State University, where he was quickly drawn to the research of Dr. Tiffany Barnes. Thomas’ graduate work focused on developing educational technologies that would make programming more accessible to a broader population of students. At NCSU, Thomas was a leader in the STARS Computing Corps, working to build community within the Computer Science department and broaden participation in computing. He served as co-president for 2 years and led the SPARCS@NCSU Computer Science middle school outreach group. While pursuing his degree, Thomas also worked with NCSU’s Khayrallah Center to develop an educational game to teach players about the experience of immigrating to the U.S. Thomas continues to look for ways to share his favorite hobby – programming – with others and to put it to good use in the world.
ACKNOWLEDGEMENTS

This work is the result of four invaluable years working with and learning from Dr. Tiffany Barnes, an experience for which I will always be grateful. Thank you for your unshakable belief in me and for modeling how to mix research with passion to make an impact on the world.

I would also like to thank the other members of my committee, Dr. James Lester, Dr. Sarah Heckman and Dr. Roger Azevedo, for the many meetings, suggestions and thoughtful critiques that made this document stronger and made me a better researcher.

Thanks also to:

• The many faculty who have shaped my research through advise, collaboration and mentorship: Dr. Min Chi, Dr. Stephen Cooper, Dr. Collin Lynch, Dr. Jennifer Albert and Dr. R. Michael Young

• My coauthors who contributed directly to the work presented here: Yihuan Dong, Dragan Lipovac, Rui Zhi, Zhongxiu Aurora Liu, Dr. Veronica Cateté and Nicholas Lytle

• My invaluable colleagues: Jennifer Tsan, who made essential contributions to the research presented in Chapter 7, and David Warren, who tested and improved iSnap and tirelessly explored many research directions with me

• ToniAnn Marini and Dana Lasher, who worked with me to deploy and evaluate my research in an authentic classroom context

• My collaborators, who have helped give this work meaning and context: Benjamin Paaßen, Dr. Kelly Rivers, Dr. Neil C. C. Brown and Dr. Akram Khater

• My friends, who kept me going with game nights, trivia outings, Ultimate Frisbee and coffee-hop counseling

My most heartfelt thanks goes to my family: Mom, Dad, James and Bridget, who supported me with love, presence and bravery.

This work was completed with support from the National Science Foundation under Grant #1623470: "EXP: Data-Driven Support for Novice Programmers" and Grant #1432156: "Educational Data Mining for Individualized Instruction in STEM."
# TABLE OF CONTENTS

## LIST OF TABLES

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Research</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Research Questions</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Contributions</td>
<td>4</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>Related Work</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Novice Programming Environments</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Examples and Evaluations of Novice Programming Environments</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Evaluations of Alternative Novice Programming Interfaces</td>
<td>9</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Intelligent Support in Novice Programming Environments</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Hints and Feedback in ITS</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Generation of Tutor Feedback</td>
<td>13</td>
</tr>
<tr>
<td>2.2.2</td>
<td>The Effectiveness of Tutor Feedback</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Automated Hint Generation</td>
<td>15</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Data-driven Hint Generation</td>
<td>16</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Automated Hint Generation without Data</td>
<td>20</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Evaluations of Automated Hints</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>Help-seeking Behavior</td>
<td>23</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Help-Seeking in the Classroom</td>
<td>23</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Help-seeking in Intelligent Tutoring Systems</td>
<td>24</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>iSnap: Intelligent Support for Novice Programming Environments</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>iSnap</td>
<td>28</td>
</tr>
<tr>
<td>3.2.1</td>
<td>On-demand Hints</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>Methods and Analysis</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>Hints in Action</td>
<td>31</td>
</tr>
<tr>
<td>3.5</td>
<td>Challenges</td>
<td>33</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Bottom-up Programming</td>
<td>33</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Reliance on Help</td>
<td>35</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Understanding Hints</td>
<td>35</td>
</tr>
<tr>
<td>3.6</td>
<td>Conclusions and Future Work</td>
<td>36</td>
</tr>
<tr>
<td>3.7</td>
<td>The Evolution of iSnap</td>
<td>37</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Designing to Address Challenges</td>
<td>39</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>SourceCheck: Data-driven Hint Generation for Open-Ended Programming</td>
<td>42</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Feedback Generation</td>
<td>45</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

| Table 3.1 | A comparison of the two versions of iSnap presented and evaluated in this document. | 41 |
| Table 5.1 | For each assignment with hints available, the number of attempts submitted (N), the number of attempts with one or more hint requests (H), the mean grade with SD, and the agreement ($\kappa$) for the graders. | 58 |
| Table 5.2 | Mean attribute ratings (with standard deviation) for first and second hints of attempts with each hint usage label. All hint ratings are summarized at the bottom. | 63 |
LIST OF FIGURES

Figure 2.1 A block-based Snap! program (left) and its corresponding graphical output (right). ................................................................. 8

Figure 2.2 A comparison of the textual (left) and block (right) interfaces used in [PB15a]. The interface for both consists of a work area (left), an output canvas where the game can be played (top-right) and a palette of usable blocks or code (bottom-right). ..................................................... 10

Figure 2.3 A simple example of an interaction network from a hypothetical algebra ITS. All students start in the same state (top) and attempt to achieve a goal state (bottom). Edge weight (thickness) corresponds to the frequency of students who took a given path. Multiple solution paths can lead to a given goal state, though some paths result in backtracking (right), indicative of a possible mistake. ............................................................ 17

Figure 3.1 The two primary types of hints offered by iSnap: script hints (top) and block hints (bottom). In both cases, a student can select a ‘hint bubble’ (left), which causes a hint dialog to display (right), suggesting the addition of an ‘ask’ block (top) or a ‘pick random’ block (bottom). ......................... 30

Figure 3.2 Anna’s code near the completion of her program. Letters mark where Anna followed hints. ................................................................. 32

Figure 3.3 An example of a hint for a “test script” that recommends modifying it like a main script. ................................................................. 33

Figure 3.4 An example of a hint to delete an ‘if’ block that is outside of a loop. ......... 34

Figure 3.5 A hint to properly nest an ‘if’ block that was not followed. ...................... 36

Figure 3.6 iSnap now highlights a student’s code to indicate possible errors. The magenta highlight (second line, ‘answer’) suggests removing a block, and the yellow highlight (6th line, ‘pick random’) suggests moving a block. ................. 37

Figure 3.7 iSnaps now displays next-step hints using “plus” buttons and blue-outlined input slots. The student can click on either to see a script hint or block hint dialog, respectively, as shown in Figure 3.1, right. ......................... 38

Figure 3.8 Informational dialogs explain the two types of feedback offered by isnap: error highlighting and next-step hints. ................................................. 40

Figure 4.1 The percentage of tutor Phase I edits predicted by SourceCheck (SC) and TC (the recall) for each edit type on the GuessingGame (GG) and Squiral (SQ) assignments. Bars are labeled with the total number of Phase I edits of each type (bottom) and the number of correctly labeled edits (top). ..................... 51

Figure 4.2 The recall for Phase II edits (top), as well the the percentage of SourceCheck (SC) and TC edits that agreed with a human tutor (the precision, bottom) on the GuessingGame (GG) and Squiral (SQ) assignments. Bars are labeled with the total number of edits of each type generated by SC and TC (bottom) and the number of correctly labeled edits (top). ..................... 52
Figure 5.1  A student's code is annotated with hint buttons (left). Upon clicking a button, a hint window suggests a change to the student's code (right). .......................... 57

Figure 5.2  Violin plots comparing grade distributions for F0 and F1 attempts at both the Squirrel (left) and Guessing Game 2 (right) homeworks. Inside each plot is a boxplot, and a red dot indicating average grade. .......................... 61

Figure 6.1  Cold start curves for the iSnap dataset (black) with uniform hint weights (solid) and voting-based weights (dotted). ALLEXPERT (green) and ONEEXPERT (blue) baselines are shown for comparison. .......................... 72

Figure 6.2  Cold start curves for the ITAP dataset (black) with uniform hint weights (solid) and voting-based weights (dotted). ALLEXPERT (green) and ONEEXPERT (blue) baselines are shown for comparison. .......................... 74
1.1 Motivation

Over the past decade demand for computing education has increased dramatically. The number of students enrolling in undergraduate Computer Science (CS) degree programs at U.S. doctoral-granting universities has tripled from 2006 to 2015 [Com17]. At the same time, there is a national effort in the U.S. to increase the availability of computing education in K-12 schools. The National Science Foundation has funded efforts to train 10,000 K-12 teachers in 10,000 high schools [Cun11], leading to large-scale professional development efforts [Cod15; Goo14; Gra15; Mor15a; Pri16c]. Partially as a result of these efforts, over 47,000 students enrolled in the new AP Computer Science Principles course and took the AP exam in 2017, more than doubling AP CS participation by females and underrepresented minority students [Cod17]. These K-12 trends suggest that undergraduate enrollment will continue to increase.

Once students enroll in CS degree programs, however, many struggle to succeed. A 2013 study by the National Center for Education and Statistics [CS13] showed that 59% of students entering Computer/Information Science Bachelor's Degree (B.S.) programs left the program before graduating, with 31% leaving postsecondary education entirely. This is the highest attrition rate of any STEM B.S. program. Most students who left their programs did so in their first year, which may be due to the high failure rates of undergraduate CS1 courses. Two separate studies [BC07; WL14] have estimated that only 67% of students pass introductory CS1 courses, with the other 33% failing or withdrawing.
There is also little evidence to suggest that this trend is improving [WL14]. While these estimates are based on a limited sample of institutions, it seems clear that many novices are struggling to learn in CS classrooms.

The growing popularity of computing education and the challenges faced by computing students both point to a clear need for better support for novice learners that can scale well to modern computing classrooms. Instructors play an important role in providing this support, but this can be a challenge for the growing number of newly-trained K12 CS teachers [Cod15; SC17], for instructors of MOOCS with thousands of students [Wan17], or in informal learning settings [Mal08] with no trained instructor. In these situations, students need a source of support that is always available. To address this, many technologies have been developed to ease the challenges of learning to program and to improve student learning. Two technologies in particular have shown promise in empirical studies: intelligent tutoring systems and novice programming environments.

Intelligent tutoring systems (ITSs) are computer systems designed to approximate the role of a human tutor, using AI and data-mining to support students as they work independently on assignments. Many ITSs have been developed in the domain of computer programming (e.g. [Ger16; Gro14b; Hol09; Mit98; RK17; WB01]), and a variety of empirical studies have shown an overall advantage of ITSs over large group instruction and other forms of computer-based instruction in the domain of computing education [Nes14]. A key feature of any ITS is the ability to give students context-sensitive feedback during problem solving, often in the form of hints [Van06]. In the domain of programming, this feedback has been shown to improve students' performance, both inside the tutor and on subsequent assessments [CA01].

Despite positive empirical evaluations, these specialized ITSs are rarely used in introductory programming classes outside of research studies. In particular, new introductory Computer Science (CS) curricula, such as CS Principles1 and Exploring CS2 are increasingly incorporating novice programming environments (NPEs) into their lessons. NPEs are systems with features specifically designed to aid novices in learning to program, such as Scratch [Res09], BlueJ [Köl03] and Snap! [Gar15]. NPEs engage students in creating open-ended projects, such as games, stories and simulations [Utt10]. These environments often include drag-and-drop, block-based programming interfaces that improve student performance and learning by minimizing the challenges of syntax [PB15b; WW17]. NPEs offer improved outcomes over traditional instruction, such as increased retention [Coo03; Mos04] and improved test scores [Dan12].

Aside from some preliminary research [Coo12; Dia17; Gus17], little effort has been made to bring the intelligent features of ITSs to these NPEs. This is due in part to the large investment of time required by domain experts to create ITSs, which can range from tens to hundreds of hours to create one hour of intelligent content [Mur99]. Further, the use of open-ended programming assignments,

1 www.csprinciples.org
2 www.exploringcs.org
which makes these environments so appealing to students and teachers [Gar15], also serves as a major barrier to providing intelligent, adaptive feedback. These assignments often have multiple, interdependent objectives that emphasize the ill-defined nature of programming and make it difficult to construct ITSs. The research presented in this document takes steps towards overcoming these challenges, combining the empirically-driven support features of ITSs with the popularity, accessibility and effectiveness of NPEs. The motivating goal of this work is to create intelligent support features that work in modern classrooms by integrating with existing programming tools, curricula and teacher training, and by scaling to new assignments using data-driven methods, rather than relying on experts.

1.2 Research

The research in this document presents the design, evaluation and implications of iSnap, a novice programming environment that offers intelligent support to students, specifically data-driven hints and feedback. I begin in Chapter 2 with a review of the relevant literature on ITSs, NPEs and data-driven programming hints. In Chapter 3, I present the design of iSnap and discuss challenges identified in a pilot study of the system, as well as how the system has evolved to address these challenges. In Chapter 4, I present SourceCheck, the data-driven hint generation algorithm that iSnap employs, and I evaluate the quality of these data-driven hints using expert tutors.

In Chapter 5, I discuss results from an evaluation of iSnap, which investigates the relationship between the quality of the data-driven hints that students received and their future help-seeking behavior. In Chapter 6, I present a methodology for comparing the quality of different data-driven hint generation approaches. I use this methodology to investigate the impact of the data quantity and data source (collected from students or expert-authored) on the quality of data-driven hints. In Chapter 7, I investigate other factors, besides hint quality, that impact students’ willingness to seek and use help from both humans and computers. Chapter 8 closes with a discussion of the implications of these results on my research questions, the contributions of this research, and important directions for future work.

1.2.1 Research Questions

This work investigates 4 high-level research questions:

**RQ1** How can we effectively integrate the support features of intelligent tutoring systems into a novice programming environment? This RQ is addressed in Chapters 3 and 4, which discuss the design and evolution of iSnap and the challenges of providing intelligent support for the more open-ended curricula associated with NPEs.
RQ2  How can we leverage student data to automatically generate hints for open-ended programming problems? This RQ is addressed in Chapters 4 and 6, which discuss the SourceCheck algorithm used to generate iSnap’s data-driven hints and how to address the challenges of hint generation using student data.

RQ3  How can we evaluate and compare the quality of techniques for generating data-driven hints? This RQ is addressed in Chapters 4, 5 and 6, which present novel methods for evaluating and comparing the quality of data-driven hints. These chapters also discuss how these methods can be used to inform our design of hint generation algorithms.

RQ4  What factors impact students’ willingness to seek and use programming help? This RQ is addressed by Chapters 5 and 7, which use classroom and laboratory studies to identify barriers that discourage students from seeking help on programming tasks when they need it, as well as factors that encourage help-seeking. Both chapters suggest how these barriers can inform our design of help systems.

1.3 Contributions

This work makes the following primary contributions:

1. The iSnap programming environment, which is the first to combine the support features of block-based programming and automatically generated hints.

2. The SourceCheck data-driven hint generation algorithm for programming, which is the first to support multiple programming languages and the more open-ended problems used in NPEs.

3. Evidence that highlights the importance of data-driven hint quality and its impact on students’ willingness to use a help system.

4. Novel methods for evaluating the quality of data-driven hints using experts, which allow researchers to benchmark and compare hint generation algorithms.

5. Results identifying factors that influence students’ willingness to seek and use help when programming and design recommendations for help systems informed by these results.
This section will review the diverse body of related work that serves as the foundation for the research presented in this document. The goal of that research is to leverage the scaffolding and popularity of existing, novice programming environments (NPEs) and combine this with the effective support features of intelligent tutoring systems (ITSs), specifically data-driven hints and feedback. This section therefore begins with a review of NPEs, ITSs for programming, and the empirical support for each. It then reviews existing techniques for automatically generating programming hints and how these techniques have been evaluated. This section closes with a review of research on how students seek and use help in the classroom and in ITSs.

2.1 Novice Programming Environments

Novice programming environments (NPEs) broadly encompass systems with features specifically designed to aid novices in learning to program. 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 through novel programming interfaces. A full review of novice environments is beyond the scope of this document, though Guzdial [Guz04] and Kelleher et al. [KP05] both offer a good review of less recent systems. Instead, this section highlights three examples of popular NPEs (Scratch, Alice and Snap!) and discusses how these systems have been evaluated empirically. A key feature of many NPEs is their use of block-based programming interfaces, and this section also
presents a review of evaluations of these interfaces. It closes with a discussion of research towards bridging NPEs with ITSs.

2.1.1 Examples and Evaluations of Novice Programming Environments

Scratch [Res09], developed by researchers at MIT, is one of the best-known NPEs. It was designed to be “more tinkerable, more meaningful, and more social” than previous environments, such as LOGO [Pap99]. To that end, Scratch features a drag-and-drop, block-based programming interface, which allows users to construct programs by composing atomic blocks of code together to form program structure (an example of block-based programming is shown later in Figure 2.1). These blocks may represent high-level structures, such as methods or loops, or low-level operators such as multiplication or equality comparison. Scratch’s output is primarily graphical, allowing users to create and manipulate 2D sprites, while adding music, animation and interactivity. The Scratch website allows users to upload, share and remix each other’s programs, adding social and collaborative elements to the environment [Fie13]. Scratch is also notable for its use of executable program fragments. Scratch programs can be built in small chunks, and any piece of Scratch code can be individually executed, with its effects immediately visible. Scratch has become a widely used programming language. As of January 2018, it ranks 17th on the TIOBE index¹, which measures programming languages’ popularity based on search engine results.

Classroom evaluations of Scratch have shown evidence that it can be an effective tool for K-12 learning when integrated into a programming curriculum but that it also presents specific challenges for students. Meerbaum-Salant et al. [MS10] designed a semester-long 9th grade Scratch curriculum and observed its implementation in two ninth grade classrooms. An analysis of student scores on a pre- and post-test of CS concepts showed significant improvement after using Scratch, though students did struggle with more abstract concepts such as initialization, variables and concurrency. However, the authors also found evidence that learning with Scratch could lead to less desirable programming habits, such as bottom-up programming with little consideration for larger program structure and extremely fine-grained programming [MS11]. Similarly, Grover et al. [Gro16] developed a 7-week FACT curriculum based on Scratch for middle school students, which significantly improved students’ computational thinking (CT) [Win06] skills. They also found that while Scratch made programming easier than its textual counterparts, students still came away with specific misconceptions about loops, variables and boolean logic, some of which may have been exacerbated by the programming environment [Gro17b]. For example, they found that many students held the misconception that loops produced the exact same output at each iteration, a notion which may have been reinforced by using Scratch’s “repeat” loop, which does not have a changing counter variable.

¹www.tiobe.com/tiobe-index/
Scratch has also been evaluated outside of the classroom in informal and independent learning settings. Maloney et al. [Mal08] describe their experience using Scratch in an urban after-school center and their analysis of the programs created. They found that Scratch was popular, and students used it voluntarily and more frequently than any other available design software, forming communities with informal, peer-to-peer mentoring [PK07]. While around 20% of the projects included only media manipulation without code, about half of the remaining programs employed loops and user interaction, and their programs increased in complexity over time. Aivaloglou and Hermans [AH16] investigated 250,000 projects in the Scratch online community, many of which are created in informal and independent learning settings. Similar to Maloney et al., they found that Scratch users commonly employed loops, and to a lesser extent, conditionals and variables (in 77%, 40% and 32% of projects, respectively). However, users employed more advanced concepts, such as procedures, less frequently. The informal learning settings investigated in these studies highlight how NPEs are often used when instructors may not be available, and the automated, intelligent support of an ITS would be useful.

Alice 2 [Coo03], and its successor Alice 3 [Dan12], are NPEs which allow users to program within a 3D environment. Alice was one of the first novice programming environments to adopt a drag-and-drop interface. It employs an event-based, object-oriented paradigm, allowing users to add objects to their scene, manipulate the objects’ attributes and call their methods. Alice offers a library of 3D objects, animations and sounds, making it a media-rich experience. Alice uses drag-and-drop controls for manipulating lines of code and some expressions, but relies more heavily on menus to give the user access to the many manipulable properties of each object. Alice’s interface shares the goal of many NPEs of simplifying programming by removing the capacity for syntax errors.

Evaluations of Alice have focused on its use in the classroom, where it has been shown to improve students’ grades [Coo03; Dan12] and retention [Coo03; Mos04] over traditional curricula. Moskal et al. [Mos04] developed an introductory college curriculum using Alice. They compared students who underwent this course prior to or concurrently with their first CS1 course to students who took only the CS1 course. They found that the Alice course significantly improved students’ grades in the CS1 course, as well as retention in CS over a two-year period. They found these trends more apparent with “high-risk” students, who had less math experience and no programming experience prior to college. A similar study by Cooper et al. [Coo03] supports these findings, showing that an Alice-based curriculum helped achieve improved grades and retention. Kelleher et al. [Kel07] found that Alice could be further adapted to young female students by adding features to facilitate storytelling programs, such as easier animations and more character-driven methods. Dann et al. [Dan12] 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. Students were given a test at the end of the course, which used Java code in its questions. The authors compared students’ scores with those of the previous, all-Java version of the course. They found that the Alice classes performed
Figure 2.1 A block-based Snap! program (left) and its corresponding graphical output (right).

on average at least one grade level higher than the previous pure Java classes on each section of the test.

Snap! [Gar15] is a web-based NPE based on Scratch, which adds features such as higher order functions, a web API, javascript interoperability and accessibility features. Figure 2.1 shows a Snap! program and corresponding graphical output. Snap! is most notable for its tight integration with the Beauty and Joy of Computing (BJC) [Gar15], an AP Computer Science Principles course taken by over 3500 students in 2017, which focuses on creating interesting computational artifacts through creative and collaborative assignments. The curriculum is backed by a successful BJC teacher professional development course that has trained hundreds of teachers and sparked over 80 BJC courses [Pri16b]. A primary goal of the course is to broaden participation in computing, and survey data supports the notion that the course (which relied heavily on Snap!) is equally appealing to and effective for diverse students [Pri15]. Snap! is a popular programming environment, actively used in high school and college classrooms, as well as in informal learning contexts – settings where students would benefit from additional help. The iSnap environment, presented in Chapter 3, attempts to address this need by extending Snap! with data-driven support features.

Other popular NPEs include MIT App Inventor [PV13], which allows users to design and program Android apps in a block-based web application. It has been evaluated in K-12 classrooms and summer camps, suggesting it is a powerful, motivational and accessible tool [Mor11], which can serve as a bridge to textual coding in Java [Wag13]. Greenfoot [Köl10] and BlueJ [Köl03] are Java-based environments designed for novice and intermediate programmers, allowing them to design games and simulations. BlockPy [Bar17a] features a split block/text programming interface and
supports novices in performing data analysis on real-work datasets from the CORGIS project [Bar17b]. While most NPEs are developed in academic settings, others, such as Code.org’s Code Studio\(^2\) and Processing\(^3\), have been developed by industry, nonprofit organizations and individuals. For example, LEGO Mindstorms NXT are customizable LEGO robotics, which can be programmed using a simple block-based programming environment. They have been used to teach programming to middle school students [Cat14] and in introductory undergraduate CS courses [Cli06].

2.1.2 Evaluations of Alternative Novice Programming Interfaces

The above studies focus on the general evaluation of NPEs and the curricula that use them, but other studies have looked at the effects of specific design choices of NPEs. A growing community of researchers have evaluated the alternative programming interfaces found in many NPEs, especially block-based programming interfaces. Taken together, these studies show clear evidence that block-based programming offers benefits over textual programming for novice programmers, including increased performance, learning and reduced difficulty with syntax errors.

Many studies have compared the outcomes of learners working in block-based and textual programming environments. Booth and Stumpf [BS13] worked with adult end-users learning to program Arduino boards. They compared the outcomes of two groups, one using Java in the Arduino IDE, and the other using the block-based ModkitAlpha programming environment. They found that the block group perceived the interface to be more friendly and experienced lower perceived workload. Lewis [Lew10] compared the block-based Scratch environment to the textual Logo environment in a 5th grade summer camp and found that students perceived exercises to be equally difficult with both environments and that each environment was better suited to teach different programming topics. Weintrop and Wilensky [WW15a] compared students’ perceptions of the block-based Snap! and textual Javascript languages. They found that students perceived Snap! to be easier, in part due to it being easier to read, with better natural layout. However, they also perceived Snap! to be less powerful, authentic and efficient to work with. The authors also compared student learning under these block and text conditions and found that while the block interface led to increased learning on most CS topics, the exact reasons for improved performance are complex and merit further study [WW15b]. Kolling and McKay [KM16] applied three sets of heuristics, designed to identify usability problems in programming environments, to Scratch and two textual environments, Greenfoot (Java) [Köl10] and Visual Basic, which are also used in educational settings. In total, they identified 58 problems in Scratch, compared to 57 for Visual Basic and 65 for Greenfoot, though for any given heuristic, Scratch had the fewest (or tied for fewest) identified problems. One of these heuristics was designed specifically for NPEs and included factors such as engagement, consistency, appropriate abstractions and feedback.

\(^2\)studio.code.org
\(^3\)www.processing.org
More recently, researchers have begun to compare block-based and textual programming interfaces directly. Price and Barnes [PB15a] compared the performance of novices using a block-based and textual programming interface for a web-based Hour of Code activity, designed to introduce the fundamentals of programming. Unlike previous work, this study controlled all aspects of the programming environment and activity between the block and text conditions, except for the programming interfaces themselves, shown Figure 2.2. They found that the block group completed significantly more objectives in less time and spent significantly less time idle compared to the text group. They also found no significant difference in students’ perceived difficulty during the programming task and hypothesized that while the text group was challenged in part by syntax, the block group was challenged by the inherent difficulty of programming. This hypothesis would support the need for intelligent support in NPEs, despite their block interfaces.

Price et al. [Pri16a] performed a similar experiment, comparing the frame-based programming interface, Stride [Köl15], to a textual programming interface in the Greenfoot [Köl10] programming environment. Like block-based programming, frame-based programming attempts to ease the challenges of syntax and syntax errors by removing many syntactic elements (e.g. semicolons, brackets), but it still uses the keyboard as the primary input method. Price et al. found that middle school novice programmers using Stride progressed through an activity significantly more quickly, spent significantly less time making purely syntactic edits, and spent less time with un compilable code.

Weintrop and Wilenski [WW17] evaluated isomorphic block-based and textual versions of the Pencil.cc programming environment. They performed a quasi-experiment, comparing two sections of a high school introductory programming course over 5 weeks, where each section used either the block or textual programming interface. They controlled for teacher effects, time on task and setting. They found that while both groups improved significantly from pre- to post-test, the effect
size was almost twice as large for the block group, with significantly higher post-test scores. Unlike Price and Barnes [PB15a], they found that students in the block group did rate programming tasks as significantly easier than the text group.

Other work has investigated the transition from blocks to textual programming. Wagner et al. [Wag13] introduced K-12 students at a summer camp to programming through MIT App Inventor, but they transitioned after two days to the Java Bridge, a Java implementation of the App Inventor API. They found that by repeating exercises first using a block interface and then a textual interface, students were able to mentally map familiar block procedures to the new textual procedures. As noted earlier, Dann et al. [Dan12] designed a curriculum to mediate transfer from Alice to textual Java programming using “bridging” and “hugging” techniques that helped students abstract concepts and apply them in multiple programming environments. Increasingly, block-based programming environments support dual-modality programming, in which students can freely switch between blocks and text (e.g. [Bar17a; HN14]), and a number of studies [Bar17a; Mat15; WH17] have investigated how students use these dual-modality environments. Results suggest that students migrate from blocks to text at varying rates [Mat15] and also transition back to blocks to accomplish a variety of tasks more easily [WH17].

2.1.3 Intelligent Support in Novice Programming Environments

The NPEs reviewed in this section are popular and accessible systems. Many have been empirically evaluated with positive results, both in general and with respect to their interfaces and associated curricula. However, despite the ingenuity of their design, none of these systems offer the intelligent support features of ITSs (discussed in more detail in Section 2.2). In addition to the work presented in this document, some preliminary research has investigated how NPEs might provide this support.

Cooper et al. [Coo12] added a series of guided tutorials to Alice, which relied on “buggy” worked examples to preemptively correct or prevent common mistakes and misconceptions. However, while the authors call the system an ITS, their work focused more on creating tutorial content “to support the ITS,” rather than its intelligent, adaptive features. BlockPy [Bar17a] is a block-based NPE that offers students sophisticated feedback, but this feedback also relies heavily on instructors to manually author the feedback and the conditions under which it should be presented [Gus17]. Diana et al. created a real-time dashboard to support Alice teachers in detecting struggling and idling students [Dia17]. The dashboard used a predictive model to estimate students’ learning outcomes using historical data, relating Alice programming log data to a subsequent assessment. Using the same dataset, Grover et al. [Gro17a] developed a framework for identifying evidence of computational thinking from log data, based on an Evidence Centered Design methodology.

This initial work is promising, but it is still preliminary, and these support features still largely fall short of the automated, adaptive support provided by ITSs. The next section reviews how ITSs
use hints and feedback to support students and how these systems have been evaluated.

2.2 Hints and Feedback in Intelligent Tutoring Systems

Intelligent tutoring systems (ITSs) are computer systems designed to approximate the role of a human tutor, using AI and data-mining to support students as they work independently on assignments. In the domain of programming, these systems have shown much promise, with a 2014 meta-analysis by Nesbit et al. indicating that ITSs for programming produced an aggregate 0.46 standard deviation improvement over other, traditional forms of instruction [Nes14]. Other research provides overviews of ITSs for programming [CLr18] and AI-supported programming instruction [Le13], so this section focuses specifically on how these systems support users with hints and feedback.

The ability to offer contextualized feedback to students during problem solving is a key feature of ITSs in many domains, comprising part of the “inner-loop” and an ITS [Van06]. While this feedback can take a variety of forms, two of the most common are next-step hints, which guide the student through completing the next step of a multi-step problem, and error-specific feedback, which identifies errors in the student’s work and explains how to fix them. Both types of feedback often come in multiple “levels,” where each level gives increasingly specific advice, until a “bottom-out” level, in which the tutor tells the student exactly what to do. For example, the Andes tutoring system [Van05] operates in the domain of physics and guides students through selecting and applying appropriate domain principles. It offers both next-step and “what’s wrong” (error-specific) help in three levels: first a pointing hint, which directs a student’s attention to the location of their error; then a teaching hint, which informs the students about a relevant domain principle; and finally a bottom-out hint, which tells the student exactly what to do next.

In the domain of programming, Le et al. [Le16] have further classified adaptive feedback in educational systems into 5 categories: 1) yes/no feedback, which simply tells the student if their program is correct, e.g. through test cases; 2) syntactic feedback, which augments the compiler’s syntactic error checking, e.g. with better explanations; 3) semantic feedback, which helps the student understand and complete the requirements of the programming task; 4) layout feedback, which helps with style and coding conventions; and 5) quality feedback, which identifies ways the student’s code could be made more efficient. This document focuses largely on semantic feedback, as do most of the systems classified by Le et al. [Le16]. The remainder of this section focuses on how programming feedback has historically been generated in ITSs and evaluations of the impact of this feedback on students.
2.2.1 Generation of Tutor Feedback

Historically, ITSs have generated feedback using an expert model, which encodes domain knowledge in a structured way that a computer can apply to the student's work. These models have historically been constructed through extensive knowledge engineering, for example by using cognitive task analysis [MG88] to learn the structure of domain knowledge from subject matter experts. This process can be very time consuming, but it can be made more efficient with authoring tools such as the Cognitive Tutor Authoring Tools (CTAT) [Ale09].

Model-tracing tutors use an expert model to “trace” a student's solution path, providing feedback to ensure that the student remains aligned to an expert solution. The expert model is encoded as a set of production rules [And93], which define appropriate problem-solving strategies using if-then rules. The ACT Programming Tutor (APT; formerly the LISP Tutor) [And89] is a model-tracing cognitive tutor for the LISP programming language and one of the first ITS for programming. The APT uses a structured code editor, which prevents students from leaving this expert path, and it provides feedback if their actions would do so. Students can also request similar feedback on demand through a next-step hint. The Ask-Elle tutor [Ger16] for the Haskell programming language is a more modern example of a model-tracing tutor, which allows an expert (e.g. a teacher) to define correct programming behaviour using a “strategy language,” rather than production rules. Ask-Elle augments traditional model-tracing error feedback with property-based testing, a sort of automated unit testing framework that finds examples of input which produce incorrect output for a student's program.

Constraint Based Modeling (CMB) [Mit07] employs an expert model consisting of a set of constraints, which describe correct behavior for a given problem or domain. Like production rules, these constraints are encoded as if-then rules, but they describe properties of a correct solution, rather than strategies for deriving one. SQL-Tutor [MO99] was the first ITS to employ CBM, though unlike the next-step hints of the previously discussed tutors, it waits until a student has submitted a programming assignment before offering feedback. SQL-Tutor identifies constraints which apply to a student's submitted solution but are not satisfied, and it then offers feedback on how to satisfy them using an expert-authored help message for each constraint. This work has been extended through other constraint-based programming tutors, which offer similar feedback upon program submission, including J-LATTE [Hol09] for the Java programming language and INCOM [LM07], a web-based tutor for the Prolog programming language.

While next-step hints and error feedback are the most common forms of intelligent feedback provided by programming systems, other approaches do exist. ELM-PE [Web96] and its web-based successor ELM-ART [WB01] are intelligent learning environments for the LISP programming language. These systems offer example-based help, in which students struggling on a programming task are given relevant, correct examples from a database of code snippets to use as a reference. The
systems use both domain knowledge of the LISP programming language and episodic knowledge of the student's behavior on previous programming exercises to recommend the most appropriate example. This example-based support is feedback, in that it is responsive to the student's past actions, but it is much less contextual than the feedback discussed above. Gross et al. [Gro14b] created an ITS which addresses this by combining example-based feedback with next-step hints. The system uses a database of expert- and user-authored solution traces, broken down into steps. When a student asks for help, the system searches this database for the closest matching solution step and presents part of this solution to the student as an example. Paasßen et al. [Paa15] extended this work by creating an adaptive distance metric that can be trained using a dataset of programs labeled with the strategy being used. The distance metric is then able to distinguish between various programming strategies for a given assignment.

2.2.2 The Effectiveness of Tutor Feedback

There is relatively good evidence to suggest that ITS feedback does provide a benefit to students, both generally and specifically in the domain of programming. Stamper et al. [Sta13] compared two version of the Deep Thought logic tutor, one with on-demand hints and one without, across two semesters of data 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. Beck et al. [Bec08] used data mining to evaluate the effect of help on student performance in the Project LISTEN reading tutor. They trained a modified Bayesian Knowledge Tracing model, which distinguished between instances of practice where the tutor did and did not elect to offer feedback. They found evidence in their model that when students received help, the probability of learning a skill was marginally higher, and the probability of demonstrating that skill was much higher, indicating that the help somewhat improved learning, and definitely scaffolded performance.

In the domain of computer programming, there is similar evidence in support of hints. Corbett et al. [CA01] compared a variety of help mechanisms in the ACT Programming Tutor, 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 and with significantly fewer errors. However, it is important to note that unlike in the feedback condition, students in the no-feedback condition were permitted to give up on problems without submitting a correct solution, which they did 30% of the time, meaning these students had less total exposure to the learning content (despite spending more time in the tutor). Holland et al. [Hol09] evaluated their J-LATTE system 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. Fossati
et al. [Fos15] developed 5 versions of the iList ITS to teach linked lists, which offered increasingly advanced forms of hints and feedback. They compared students working each version of iList to students who worked with a human tutor and students who received no instruction. They found that students using the most advanced version of the ITS learned significantly more than students who received no instruction, and they also learned the same amount as students who had human tutoring. Section 2.3.3 further discusses the effectiveness of hints for programming, with a focus on evaluating automatically generated hints.

Despite the large number of ITSs which offer hints and feedback to students during problem solving, as well as the empirical evidence above in support of these features, researchers are still trying to determine when and how this help actually helps. Aleven et al. [Ale16] reviewed the literature on principle-based hints in ITSs and reflected on their own experience building the Help Tutor, designed to teach students to make better use of this help (discussed further in Section 2.4.2). They concluded that “hints and help-seeking can help students learn, but that they are not the strong influence on student learning that we once thought they were” [Ale16]. A study by Roll et al. [Rol14] on data from the Geometry Cognitive Tutor provides some evidence for this idea, showing that hints may only be effective for some students. They found that hints were most effective for students with an intermediate skill level on a given problem but not effective for students with a low skill level for that problem. Regardless of their impact on learning, hints may be necessary in many ITSs, especially “bottom-out” hints that allow a student to progress when stuck in order to reach subsequent learning content. A study by Shih et al. [Shi08] suggests that these bottom-out hints may be useful for learning when students take time to self-explain the meaning of the hint, but such instances of spontaneous self-explanation are rare, suggesting a need for the ITS to scaffold this self-explanation process. These studies temper the earlier results of the benefits of tutor help, suggesting that its role is nuanced and not fully understood, and emphasizing the need for more work across domains.

### 2.3 Automated Hint Generation in Programming

The model tracing and constraint-based ITSs described earlier are effective when expert content authors are available, and when programming assignments are easily defined by a set of constraints or production rules. However, this work focuses on NPEs, where a diversity of curricula and contexts makes it infeasible to support each assignment with expert-authored content. Further, the assignments themselves are often open-ended⁴ and creative, with multiple, interdependent goals, making it difficult to author ideal programming behavior without restricting students’ solutions.

---

⁴While the term “open-ended” is used in a variety of ways in the literature, in this document it is used simply to mean assignments with goals that can be accomplished in a variety of ways, involving discretion on the part of the student. These assignments may still have clear objectives and structure.
This section describes automated approaches to generating feedback for programming, specifically programming hints. These approaches focus less on knowledge-engineered domain models, relying instead on examples, test cases and data to generate hints. These systems are specific to the domain of programming and come from the ITS, Artificial Intelligence, Human-Computer Interaction and Software Engineering research communities. This is a quickly growing field of research, with few papers published before 2014, and over 25 published between 2014 and 2017. The remainder of this section reviews data-driven and automated methods for hint generation in programming with a focus on how hints are generated and evaluated.

2.3.1 Data-driven Hint Generation

Data-driven hint generation uses a database of correct solutions to a given programming problem to automatically generate hints for students. This section primarily focuses on next-step, edit-based hints, which suggest edits (e.g. insertions or deletions) that will bring a student's code closer to a correct solution, but it also briefly discusses other forms of data-driven hints. Some hint generation algorithms take as input a database of final, complete solutions to a problem, while others utilize a database of solution traces, which describe each step in constructing the solution program. Typically, these solutions or traces come from other students, for example from a previous semester, who correctly solved the problem. However, as explored further in Chapter 6, the data could also come from experts. Algorithms often represent solutions and partial solutions using an Abstract Syntax Tree (AST), a directed, rooted tree where each node represents a program element, such as a function call, control structure or variable, and the hierarchy of the tree represents how these elements are nested together.

2.3.1.1 The Hint Factory and Graph-based Approaches

Many data-driven hint generation methods are based on the Hint Factory [BS10], an algorithm for generating next-step hints for students working on multi-step problems in a variety of domains. It operates on a data-structure called an interaction network [Eag12], which is built from log data of the interactions between students and a learning environment for a given problem. The interaction network is a directed graph, where each vertex represents a state of the problem. In programming a state usually corresponds to a snapshot of the student's current code. States are connected by edges, which represent student actions, such as adding, editing or deleting code, which transform one state into another. Each student attempt is traced from a start state to its final state, and is added to the interaction network. If this final state is a correct solution, it is labeled as a goal state. By combining all students’ attempts into a single network and weighting edges with the number of attempts that passed through them, the interaction network forms a compact representation of student problem-solving strategies for a given problem. Figure 2.3 gives a simple example of an
Figure 2.3 A simple example of an interaction network from a hypothetical algebra ITS. All students start in the same state (top) and attempt to achieve a goal state (bottom). Edge weight (thickness) corresponds to the frequency of students who took a given path. Multiple solution paths can lead to a given goal state, though some paths result in backtracking (right), indicative of a possible mistake.

The Hint Factory uses the interaction network for a given problem to generate hints for new students working on that problem. When a student requests a hint, the algorithm matches that student to an existing state in the network and then calculates the best path from that state to a goal state. Once a solution path is calculated, it is typically used to provide a next-step hint, which points the student towards the next state on the solution path. The Hint Factory uses a Markov Decision Process (MDP) to calculate this solution path [BS10], but other techniques can also be used, which are more effective in some contexts. For example, Stamper and Barnes [SB09] found that using the frequency of solution steps to assign values to states in the interaction network produced hints that an expert rater preferred to MDP-based hints. Additionally, Piech et al. [Pie15] developed two “desirable path” hint policies for suggesting programming hints using a graph structure similar to an interaction network. The “Poisson Path” policy directed students down the path with the lowest expected time to reach a correct solution, and the “Independent Probable Path” policy directed students down the path that the average successful student was most likely to have taken. The authors found that both policies outperformed an MPD policy, as discussed in Section 2.3.3.

The domain of computer programming presents a serious challenge for the Hint Factory. Even for simple programming problems, the space of possible solutions is quite large, often infinite, and there may be little overlap among student solutions [RK12; PB15a]. This overlap is important for the Hint Factory, allowing it to connect the hint-requesting student to a variety of correct solution states.
in the interaction network. To address this lack of overlap, Rivers and Koedinger [RK17] extended the Hint Factory in their ITAP tutor, using a strategy called path construction to generate a path from a student’s current state to a previously observed goal state, rather than relying on observed student paths. Their approach computes a change vector of all edits needed to transform the student’s current state into the goal state, and tests to see if any closer solutions are discovered along the way. Hicks and Peddycord III [Hic14; Ped14] applied the Hint Factory to a programming game called BOTS, but rather than representing a student’s state using an AST (a codestate), they used the state of the game world after running the student’s program (a worldstate). The authors found considerably more overlap among worldstates than codestates, allowing more hints to be generated; however, these hints may be more challenging for students to use.

Others have used graph-based hint generation approaches very similar to the Hint Factory. Fossati et al. [Fos15] employed a graph-based approach in their iList ITS, which offers reactive and proactive feedback on linked-list problems. Min et al. [Min14] proposed an adaptive scaffolding system for a programming game, which would use Hint-Factory-style hints during problem solving that are adaptive to the learning policy being employed by the “outer loop” of the system [Van06]. Chow et al. [Cho17] also developed a path-based hint generation technique for programming, but their approach does not match a hint-requesting student to a single state in the graph. Instead, they use test cases to match the hint-requester to other code states that failed the same set of test cases, which have structurally similar code. They identify which of these code states come from students with similar attributes to the hint-requester, such as edit style, progression speed and consistency. They then recommend edits based on what the majority of those similar students did to fix the failing test.

Most methods for hint generation, especially graph-based methods, benefit from increased overlap among student programs, making it easier to connect various solution paths. This overlap can be increased through “canonicalization,” which standardizes the syntax of programs, while maintaining their semantic meaning. For example, the expression \( a > b \) can be rewritten \( b < a \) without changing its meaning. Rivers and Koedinger [RK12] developed a comprehensive technique for canonicalization, which standardizes programs in a variety of ways, such as normalizing arithmetic and boolean operators, removing unreachable and unused code and inlining helper functions. Jin et al. [Jin12] used a different approach, representing a student’s program as a Linkage Graph, where each vertex is a code statement, and each directed edge represents an ordering dependency. This removes some semantically unimportant ordering information from the program, allowing for more overlap.
2.3.1.2 Solution-based Approaches

While graph-based approaches like the Hint Factory focus on problem-solving paths, other approaches focus instead on complete solutions and the syntactic structure of students’ code. The most straightforward of these approaches is to identify a correct solution that is similar to a hint-requesting student's current code and suggest edits to transform that code into the target solution (e.g. [ZR15; Wan17]). Others cluster student solutions and use these clusters to generate hints (e.g. [Gro14a; Hea17]). An alternative approach is to focus on patterns in the student's code to identify possible bugs or missing elements (e.g. [Laz17]). Many of these hint generation techniques, including graph-based approaches (e.g. [Cho17; Hea17; LB14; Per14; RK17]), require that a problem can be checked with a set of test cases, which allows the algorithm to determine how a student’s code is failing and to test whether a suggested edit will correct that problem.

A common solution-based hint generation approach is to use a single target solution in the database for hint generation. Zimmerman and Rupakheti [ZR15] use a pq-Gram tree edit distance algorithm to match a student’s program to its closest counterpart in a database of expert-curated solutions, though one could imagine just as easily applying this method to a database of student solutions. They use the PQ-gram output to identify the set of node insertions, deletions and relabelings necessary to transform the student's AST into this solution. Similarly, the iGrader system [Wan17] searches a database of correct student solutions for the best match and infers edits between the student’s code and the selected solution to generate hints. To facilitate the search process, iGrader only searches for solutions that share the same control flow structure of the hint-requesting student.

Others use clustering to identify a set of solutions in the database that closely match a hint-requesting student’s code and use this cluster to generate hints. Gross et al. [Gro14a] cluster their database of student solutions, with the intention that each cluster represents a different “solution strategy.” They identify the cluster to which the hint-requesting student belongs and generate a “prototype” solution for that cluster to use for hint generation. They identify two methods for generating hints from this exemplar: F1, which highlights lines where the hint-requesting student’s code differs from the exemplar and encourages reflection; and F2, which shows the student the part of the exemplar code which contrasts with the student’s own code. These feedback strategies differ from the other edit-based hints in this section, and may be more effective for some students; however, this method could also easily be adapted for edit-based hint generation. The MistakeBrowser [Hea17] identifies edits in historical data that caused a formerly failing test case to pass and then clusters students based on whether this same edit similarly fixes the same test case. This edit can then be provided as a hint to any new student who is assigned to the same cluster. MistakeBrowser also contains an interface by which an instructor can browse these clusters and author textual feedback to be presented along with the hint.

Other approaches attempt to break a program down into smaller pieces and generate hints for
the pieces individually. Lazar and Bratko [LB14] generate hints based on single lines of code. To generate a hint for a given program, they look up the individual lines in their database of solutions and suggest the most common transformation for that line. Lazar et al. [Laz17] later took a different, rule-based approach to hint generation. They extracted common AST patterns from a dataset of both correct and incorrect student attempts, and they attempted to identify conjunctions of patterns ("rules") that predicted correct (positive rules) or incorrect programs (negative rules). They generated hints to address the presence of a negative rule (a "buggy hint") or the absence of a positive rule (an "intent hint").

2.3.1.3 Other Data-driven Hints

This section has focused primarily on next-step, edit-based, semantic programming hints, which suggest how a student can edit their code to bring it closer to a correct solution, but some data-driven systems provide other types of hints. The AutoStyle system [Mog15] suggests how students can improve their already correct solutions with better style. AutoStyle shows students examples of other student solutions that are similar but have a better style rating, computed automatically using static analysis. Choudhury et al. [Cho16] extend this approach by clustering solutions into different “approaches” to solving the problem. They then asked instructors to label each cluster with a quality rating, an exemplar solution, and a natural language hint, which are used to present style hints. HelpMeOut [Har10] and BlueFix [Wat12] are social recommender systems that focus on helping students resolve compiler error messages by showing students with errors examples of how other students resolved the errors. The DeepFix system [Gup17] also helps students resolve compiler errors, but it used deep learning with a recurrent neural network to learn these fixes from student data. TraceDiff [Suz17b] visualizes execution traces for a student's program and can highlight where the student's trace diverges from a corrected version of their program. In addition to their edit-based hints, the system designed by Chow et al. [Cho17] also offers input-based hints that suggest test inputs for a student's code that the student has not yet tried and which similar students did try.

2.3.2 Automated Hint Generation without Data

Others have employed methods from software engineering literature to generate hints. Rather than relying on a database of solutions, these methods use a model of knowledge about the programming language, paired with a set of test cases, to transform a student's solution until errors have been corrected. Singh et al. [Sin13] use program synthesis to generate a new solution from the student's current program. They use an expert-provided Error Model, which defines a set of potential transformations to a student's code for a given problem, to generate a set of edits which lead to a correct solution. This Error Model has a number of similarities to the expert models of many ITSs but focuses more on the mechanics of program transformation, rather than the domain content.
of the programming assignment, and as such can discover a wider variety of possible solutions. Phothilimthana et al. [PS17] extend this work by creating a simpler language for encoding error models. They also add a “case analyzer” which offers hints informing students if they are missing a conditional statement that is present in the instructor’s solution.

Perelman et al. [Per14] also employ program synthesis to search for a set of transformations from the student’s current program to a solution program. They use a domain-specific language (DSL), which allows authors to define possible program transformations; however, they show that this DSL can also be automatically generated from previous student solutions. Yi et al. [Yi17] adapted automated program repair (APR) to the task of providing programming hints on introductory assignments. They found that without modification, APR is insufficient to fix the many bugs found in student programs but that when modified to produce partial program repairs, APR is almost twice as effective.

2.3.3 Evaluations of Automated Hints

Authors have developed a number of approaches for evaluating data-driven hints. Most of these are technical evaluations, which estimate the value of hints using historical data. Unless otherwise specified, the authors used all historical data in their evaluations (or a specific subset, such as incorrect attempts), weighting all attempts equally. The results may therefore not represent the reality that some students are more likely to requests hints than others. These evaluations can be classified into the following categories:

Availability: This is the simplest evaluation method, which estimates how frequently hints would be available to requesting students. Perelman et al. [Per14] found that their test-driven synthesis algorithm could generate hints for 65% of player attempts in a dataset from the CodeHunt programming game, while the iGrader system [Wan17] could generate hints for 54% of incorrect attempts in a C# online course. A common approach for measuring hint availability is to perform a “cold start” analysis [BS10] to investigate how much student data is necessary to achieve high hint coverage for students. Many authors of data-driven hint systems have performed this analysis [Cho17; Ped14; RK17; BS10], and this approach is discussed further in Chapter 6.

Correction: For programming environments that use test cases to measure correctness, a similar evaluation method measures how frequently a system can fully correct an incorrect attempt. For the attempts where iGrader [Wan17] was able to generate hints, it could fully correct 72% of these attempt with at most three “fixes.” Lazar and Bratko [LB14] were able to correct 35-70% of incorrect attempts at a set of Prolog problems with their hint generation system. The DeepFix system [Gup17] could correct 27% of incorrect C programs fully and an additional 19% partially. Singh et al. [Sin13] were able to correct 65% of incorrect student attempts from an introductory programming MOOC with their program synthesis approach.
**Student Choice:** Using historical data, it is possible to generate a hint for a given student at a point in time and then determine whether that student later enacted the edit suggested by the hint. Lazar et al. [Laz17] found that 97% of their hints aimed at correcting errors would hypothetically have been followed by students, while 56% of their hints aimed at furthering the program would have been followed. Price et al. [Pri16b] found that 32-35% of the hints generated by their CTD algorithm would have brought students closer to their own final solution to the problem, compared to 61% of the student's own actions.

**Student Evaluations:** Some data-driven hint generation algorithms have been deployed in classroom or laboratory studies with students. Some of these evaluations involved ITSs and were discussed in Section 2.2.2, showing that ITSs employing data driven hints can foster learning gains equivalent to human tutors [Fos15] and improve students’ grades [Sta13]. Outside of ITSs, Phothilimthana et al. [PS17] compared students’ performance with and without programming hints available across semesters of a Scheme course and found that students with hints made significantly fewer submission attempts before completing problems. Yi et al. [Yi17] found that their automated program repair hints did not reduce the time it took students to solve programming problems but that these hints may help TAs grade more quickly (they did not test if this difference was significant). Choudhury et al. [Cho16] found that students with access to their code style hints produced significantly better quality correct solutions than students who did not. Student evaluations sometimes also include surveys on student satisfaction with the hint system [Fos15; Har10; PS17]. While many studies have evaluated ITSs and learning environments for programming using students, most evaluated whole systems, rather than the direct impact of programming hints, as in the studies listed here.

**Expert Evaluations:** While classroom and laboratory evaluations produce the strongest evidence, they are also expensive to conduct. An alternative is to directly evaluate the quality of data-driven hints using experts. Hartmann et al. [Har10] manually labeled hints from their HelpMeOut system as “Helpful” or “Not Helpful” and found that 47% of hint requests returned helpful suggestions. Watson et al. [Wat12] used a similar method to compare their BlueFix system to HelpMeOut on a set of author-generated buggy hint requests and found that their system performed significantly better. Stamper and Barnes [SB09] compared two hint generation methods by having an expert select which method generated the better hint. Gross et al. [Gro14a] used experts to validate their cluster-based hint generation and found that experts often selected an exemplar solution from a different cluster than their algorithm to generate feedback. Piech et al. [Pie15] compared the effectiveness of a variety of hint generation algorithms by comparing their output to “gold standard” hints generated by a group of human experts. The evaluations of data-driven hint quality presented in Chapter 4 and Chapter 6 draw heavily on this “gold standard” rating techniques.
2.4 Help-seeking Behavior

Many forms of feedback in ITSs, especially next-step hints, require the student to initiate a request for help. Making this request is a form of help-seeking, a self-regulatory skill that plays an important role in learning. Help-seeking can be defined as the “the ability to solicit help when needed from a teacher, peer, textbook” or other information source [Ale06]. How students seek and use help strongly impacts how effective hints can be, as explored in Chapters 5 and 7 of this document. Help-seeking has been studied extensively in the classroom, as well as in ITSs, and the rest of this section reviews literature on both of these settings.

2.4.1 Help-Seeking in the Classroom

Some of the first efforts to understand help-seeking in the classroom focused on formulating a model of students’ help-seeking behavior. Most of these models are rooted in original work by Nelson-Le Gall [Nel81], who proposed a cognitive and behavioral model of help-seeking in school-aged children, consisting of five components: 1) Awareness of the need for help; 2) Decision to seek help; 3) Identification of potential helpers; 4) Employment of strategies to elicit help; and 5) Reactions to help-seeking attempt(s). The model was intended as a heuristic aid, rather than a formal theory, and focuses on instrumental help-seeking, in which the child is seeking only enough help to allow them to complete the problem on their own. Newman [New94] expanded this model, in part by presenting step (2) as a comparison between two factors. The first is a learner's self-efficacy level (SEL), a dynamic self-assessment of confidence and ability for a given problem or step. This is compared against the learner's confidence tolerance level (CTL), the threshold of challenge and risk the student is willing to accept, inherent in the student’s beliefs, to determine whether help should be sought. Both authors stress that dependence (evidenced by requests for help) and independence are not mutually exclusive ends of a spectrum, but rather that students can use instrumental, adaptive help-seeking as a self-regulatory skill that maintains independent work [Nel81; New94].

Researchers have also identified key factors that influence students’ decision to seek or avoid help in the classroom. Students may avoid help for practical reasons: help may be inaccessible or inconvenient, and the use of help may be prohibited or go against social norms of the classroom [Rya01]. Students may avoid help because of concerns about the help-giver’s competence (e.g. if it is a peer), a desire for independence [Van88], or because of a perceived threat to competence [But98]. Lower-performing students and students with lower self-esteem are more likely to feel a threat to competence by seeking help, and this threat can manifest in help avoidance or seeking expedient help (with the goal of finishing the problem), rather than instrumental help to foster learning [Kar04]. Students’ help-seeking behavior is also influenced by their achievement-goal orientation, a motivational variable that describes their disposition towards learning and achievement. Students
with a *performance* achievement-goal orientation are motivated by others' perception of their relative competence, and those with a *mastery* orientation are motivated by a desire for learning and self-improvement (though these are not mutually exclusive orientations). Ryan et al. [RP97; Rya01] found that students' performance goal orientation correlated negatively with adaptive help-seeking, while mastery orientation correlated positively. Help-seeking in the classroom is also a social experience, and students with a social status-goal orientation, focused on social visibility and prestige, may feel increased social costs of help-seeking and therefore avoid help [RP97; Rya01].

### 2.4.2 Help-seeking in Intelligent Tutoring Systems

While most research has focused on how students seek help in the classroom, a more recent effort has been made to understand how this work translates to the context of ITSs and other interactive learning environments. Aleven and Koedinger [AK00] noted that help-usage in their PACT Geometry Tutor ITS was far from ideal. Students in the tutor requested help on only 22-29% of steps, and when they did request help, students skipped through all the levels of help to get to a “bottom-out” hint that explained the answer on 82-89% of steps. Students have also been shown to “game the system,” misusing help features to quickly progress through the assignment, rather than as learning tools, a behaviour which leads to decreased learning [Bak04]. Aleven et al. [Ale06] addressed this poor usage of ITS help by formulating their own model of ideal help-seeking behavior in a Cognitive Tutor [Ale06]. This model drew on the previous models of Nelson-Le Gall and Newman, but situated itself in the context of step-by-step problem solving with on-demand computer help. Additionally, theirs was a model of *ideal* help-seeking, which noted where students might deviate from this model as “help-seeking errors.”

Aleven et al. [Ale06] used this model as the basis for the Help Tutor, a cognitive tutor which teaches the metacognitive skill of help-seeking as students solve problems in the Geometry Tutor. In an empirical evaluation, the authors found that help-seeking errors negatively correlated with domain learning, that the Help Tutor reduced the incidence of some of these errors, but that it had no impact on domain learning [Rol06]. Tai et al. had similar success improving students' help-seeking behavior by portraying the tutor-student relationship as teamwork, but they also found that this did not result in improved domain learning [Tai13]. Mercier and Frederiksen [MF08] attempted to generalize the model put forward by Aleven et al. to create a cognitive model of help-seeking in interactive learning environments, which added steps for diagnosing the problem and comprehending help. Additional work by Roll et al. [Rol14] suggests that appropriate help-seeking behavior depends on a student's current skill level. While students who followed the authors’ model of appropriate help-seeking showed improved learning overall, this was not true of students who had low skill coming into a problem. Instead, these students benefited from quick (often wrong) attempts at the problem, rather than asking for help immediately when encountering unknown material.
Others have explored student factors that impact help-seeking in ITSs. Wood and Wood [WW99] found that students with lower prior knowledge were more likely to seek on-demand help and benefit from it in the QUADRATIC math learning environment. This contrasts with classroom research on help-seeking, which shows that lower-achievers (or those who perceive themselves to be) seek less help [Rya01]. In a later study, Wood used a pretest to select challenging problems for each student [Woo01]. In this case, prior knowledge did not predict help-seeking, suggesting that the original effect was due to subjective problem difficulty, rather than prior knowledge alone. Bartholomé et al. [Bar04] studied pairs of students working in a “plant identification” learning environment with on-demand help and found that pairs with mixed (high and low) prior knowledge were more likely to seek help and less likely to produce errors, but they found no effect of pairs’ “motivational orientation” or interest. Vaessen et al. [Vae14] studied help-seeking in the context of a computing learning environment for Haskell with on-demand help. They clustered students’ attempts at short programming problems based on log data to identify help-seeking strategies. They identified 5 strategies and noted that students’ achievement-goal orientations were predictive of 3 of the strategies. Aleven et al. also offer useful reviews of help-seeking research in interactive learning environments [Ale03; Ale06; Ale16].

One of the goals of the research presented in this document is to better understand how students seek and use help, specifically in the domain of computing. Chapters 5 and 7 explore this theme further by investigating how students use data-driven hints in the iSnap programming environment. The next chapter introduces this environment and the data-driven help it offers.
CHAPTER
3

ISNAP: INTELLIGENT SUPPORT FOR NOVICE PROGRAMMING ENVIRONMENTS

This chapter was adapted from:

The original text has been modified as follows: The Related Work section has been incorporated into Sections 2.1 and 2.2 of Chapter 2, and Section 3.7 has been added, which describes changes that have been made to the iSnap programming environment since the original publication.

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.

3.1 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 [Mos04], Greenfoot [Köl10], Scratch [Res09] and Snap! [Gar15], 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 [Mos04], improved test scores [Dan12] and more effective programming editors [PB15b].

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 [Cor01]. 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 [CA01].

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 [Mur99]. Further, the use of open-ended programming assignments, which makes these environments so appealing to students and teachers [Gar15], also makes providing intelligent, adaptive feedback difficult.

In this chapter, we investigate RQ1: How can we effectively integrate the support features of intelligent tutoring systems into a novice programming environment? To address the question, 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.

3.2 iSnap

iSnap is an extension of the Snap! [Gar15] programming environment. Snap! is based on Scratch [Res09] 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 [Gar15], 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.

3.2.1 On-demand Hints

In the original version of iSnap, discussed in this chapter, hints are generated using the Contextual Tree Decomposition (CTD) algorithm [Pri16b] (Section 3.7 discusses how iSnap has evolved since this original version). 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 students edit their code, and hints 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 [Pri16b].

The main addition that iSnap makes to the interface of 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 3.1. 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 3.1 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.

3.3 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) [Gar15].

Hints were available on one graded in-lab assignment, GuessingGame. 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. A full description of the GuessingGame assignment, along with all the iSnap assignments referenced in this document, can be found in Appendix A. The hints used in this study were generated using 32 fully correct student submissions of GuessingGame, 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 GuessingGame
Figure 3.1 The two primary types of hints offered by iSnap: script hints (top) and block hints (bottom). In both cases, a student can select a ‘hint bubble’ (left), which causes a hint dialog to display (right), suggesting the addition of an ‘ask’ block (top) or a ‘pick random’ block (bottom).

We collected 63 attempts at GuessingGame with hints available, consisting of 10,753 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

---

\(^1\)Datasets and a demo of iSnap are available at http://go.ncsu.edu/isnap
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, and students followed 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 3.5.

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 hints or 1 hint. 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.

3.4 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 3.2, with letters labelling where she received hints. Anna started her project by welcoming the player and asking for the player’s name as specified in the lab objectives, but she got stuck trying to greet the player by name. She first added, and later deleted, a ‘say’ block. She then asked for a script hint, which directed her to add a ‘say’ block (A). This confirmed her original idea, and she followed the hint. She asked for another script hint, which directed her to store a variable at the beginning of her script (B). Anna named this variable ‘Name,’ presumably because she thought it was addressing her current goal of greeting the player; however, in reality the hint was addressing the later goal of storing a random number. Having not yet finished greeting the player by name, Anna requested another hint, this time getting a block hint for the ‘say’ block she had added previously. This hint directed her to add a ‘join’ block (C), which she did. Once she found the ‘join’ block, Anna independently filled in its parameters to greet the player by name.

Anna then proceeded 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 requested a block hint for the ‘repeat-until’ loop condition, which directed her to add an equals comparison (D). Before acting, Anna also asked for a script hint for the body of the same loop, which directed her to replace her ‘say’ block with an ‘ask’ block (E). She proceeded to follow both hints, but she filled in their parameters on her own. For the equals loop condition, she mistakenly compared the player’s answer to a newly generated random number each time, rather than comparing it to a single, stored variable. Anna
then added an ‘if-else’ block to the loop body, and requested a script hint for this block, which
directed her to nest a second ‘if-else’ block in the else body (F). She requested a block hint for the
first ‘if-else’ block as well (G), which directed her to use an equals comparison as the condition. She
did so, and then figured out the condition for the second ‘if-else’ block on her own. For each of these
comparisons, she made the same mistake that she had made in her loop condition, comparing
the player’s answer to a new random number. She requested a hint for the first if-else comparison (H),
which directed her to replace this random number with a variable. She independently created and
properly instantiated this variable, and followed the hint. She then made the same correction for
the other three comparisons on her own. Now, with a complete program, she asked for hints one
last time. Only two hint bubbles were displayed, indicating that she was nearly done. She chose not
to click a hint bubble on her ‘if-else’ block, but she did select a hint telling her she may have too
many variables. She followed this hint, deleting the ‘Name’ variable she had created earlier.

Anna shows how hints can be used to overcome challenges in the GuessingGame assignment
that otherwise would require an instructor. Anna requested help when she was stuck but used the
help to complete more of the assignment than the hint covered, showing her ability to generalize
ideas. While some might argue that she used help too much, her work demonstrates several positive
3.5 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.

3.5.1 Bottom-up Programming

Scratch (and by extension Snap!) was designed to facilitate bottom-up programming and tinkering [Res09]. 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. 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 3.3 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 3.4 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.
3.5.2 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 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 Bill, relied on hints to complete the beginning of the assignment, including his first use of an ‘if’ block with an equals comparison. However, Bill then correctly constructed the next two ‘if’ blocks without hints, perhaps referring to the original hint-based code as an example. Next, Bill 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, Bill was still able continue the assignment independently without them.

3.5.3 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 them closer to a solution or correcting errors, but the students chose not to follow the hints. 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 3.5, 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.

3.6 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. Section 3.7 discusses how iSnap has evolved to address many of the challenges identified in this pilot study, such as providing hints for students who use bottom-up programming. Additionally, 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. We also noticed that some students did not follow the hints they received, and there are several possible explanations for this: it could
Figure 3.6 iSnap now highlights a student’s code to indicate possible errors. The magenta highlight (second line, ‘answer’) suggests removing a block, and the yellow highlight (6th line, ‘pick random’) suggests moving a block.

be a lack of understanding, a lack of hint quality, or simply that the student had another intention or belief about what the hint would show. Chapters 5 and 7 discuss how we can improve student adoption of hint suggestions.

Even with these proposed improvements, we recognize that next-step hints will not satisfy all students’ help requests. For example, 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. Section 3.7 discusses how iSnap’s feedback has changed to provide these error-checking help features as well.

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 [Cor01], for example, could be used to tailor programming environments to individual students based on models of their cognitive and affective states.

3.7 The Evolution of iSnap

Based on the results of the pilot test described in this chapter, we redesigned many aspects of iSnap to produce a new version, iSnap v2. This section describes the evolution of the system since this work was originally published. The largest change to iSnap is that it now divides its help into two
Figure 3.7 iSnaps now displays next-step hints using “plus” buttons and blue-outlined input slots. The student can click on either to see a script hint or block hint dialog, respectively, as shown in Figure 3.1, right.

categories: error-checking feedback and next-step hints. In Section 3.6, we noted that many students pressed iSnap’s Help button when their program was nearly complete, indicating that they were interested in identifying possible errors before submitting. iSnap now offers that feature explicitly. The Help button has been replaced by a “Check My Work” button. When the button is clicked, iSnap starts by highlighting a student’s code to indicate possible errors, as shown in Figure 3.6. iSnap highlights blocks in magenta if they likely do not belong in a correct solution, and it highlights blocks in yellow if they likely do belong but appear to be in the wrong location. These highlights replace many of the hint buttons from iSnap’s original interface, which could easily become cluttered when too many hints were available.

These highlights represent more than a change in the hint interface. As discussed in Chapter 4, iSnap now uses a new hint generation algorithm, SourceCheck. In addition to suggesting inserting and deleting blocks, these hints can also suggest that students move a block of code to a new location. This is demonstrated in Figure 3.6, which depicts a scenario that Anna encountered, as described in Section 3.4. Anna was struggling with two related issues: she used a variable to store the player’s name, rather than storing a random number, and she generated a new random number for each comparison in the ‘repeat until’ loop, rather than comparing to a variable. If Anna had used iSnap’s new error-checking feedback, she would not only have seen that the magenta-highlighted ‘answer’ block should be removed, she would also have seen that the yellow-highlighted ‘pick random’ block should be moved. If Anna had hovered her cursor over the ‘pick random’ block, iSnap would have also indicated the correct location for this block, inside of the ‘set variable’ block.

Students can also ask iSnap to offer next-step hints. As shown in Figure 3.7, iSnap supplements its error highlighting with buttons that will show the student a next-step hint dialog, just as in the
original interface (see Figure 3.1, right). Any “block hint” that suggests changing the inputs of a block is displayed as a blue highlight around that input slot, further reducing the clutter of additional hint buttons. Even if a student does not click on these hints buttons or blue highlights, their presence also serves as a passive indicator that additional code should be added in that location. These blue indicators disappear when a student adds the correct code, serving as a passive confirmation. While highlighting is active, students can toggle next-step hints on or off using a dialog window, shown in Figure 3.8, which also provides information about iSnap’s feedback.

3.7.1 Designing to Address Challenges

Section 3.5 describes three challenges that were identified in the pilot study, which informed the redesign of iSnap and its new SourceCheck hint generation algorithm. To address the challenge of bottom-up programming, and specifically the “test scripts” depicted in Figure 3.3, iSnap will only suggest adding code to a script if it matches a script in the correct solution. Instead of suggesting that the student add more blocks to a test script, iSnap now highlights the existing blocks in the script to indicate whether they belong somewhere in a correct solution (yellow), or if they are likely unneeded (magenta).

To address the challenge of overreliance on hints, iSnap now gives a warning when students have used more than a given number of hints in a row, asking if the student is feeling stuck, and suggesting that they stop using hints and consult a teaching assistant. During the Fall 2017 semester, the same class described in this study used iSnap, with hints available on 4 programming assignments, and no more than 4 out of 52 students in the class received this warning on any given assignment. The warning was set to display if students used more than 5 hints in a row. However, those students who did receive the warning almost never took the suggestion to stop using hints, so help abuse remains a concern.

Many of the following chapters concern the challenge of helping students to understand hints. They present evaluations of both the old CTD [Pri16b] hint generation algorithm and the new SourceCheck algorithm (Chapter 4) that focus on hint quality, a notion which includes how relevant and interpretable the hints are to students (see Chapter 5). These evaluations were used to improve SourceCheck to make hints more understandable.

In addition to the three primary challenges identified in Section 3.5, iSnap’s redesign addressed other issues that students encountered. Section 3.4 describes Anna’s confusion around a hint to create a variable. Presumably, she thought it was related to her current objective of greeting the player by name, when in reality it was intended to store the random number for GuessingGame. iSnap no longer offers hints suggesting that the student add or remove variables, since they often came too early, as in Anna’s case. Instead, it uses variable assignment blocks as an opportunity to suggest that the player needs a variable. Additionally, iSnap now specifies which variable a hint is
referring to, rather than referring to all variable as ‘var,’ reducing possible confusion. Another major challenge that we observed is that many students are hesitant to use hints, even when they need them. This is a major theme of this document, discussed in more depth in Chapters 5 and 7. To address this, iSnap now reminds students that they can ask for help if they need it at the start of each assignment. The first time that students run their programs, iSnap also asks them if they would like it to check their work (with error highlighting) when they run their program.

Table 3.1 summarizes the changes between the original iSnap and the updated iSnap v2 described in this section. Different studies presented in this document have used data collected with different versions of iSnap, and this information is also summarized in the last row of Table 3.1. Chapters 5 and 7 present evaluations of the original version of iSnap. Chapter 4 presents the SourceCheck algorithm and evaluates it using data collected with iSnap v2. Chapter 6 uses data collected from both the original version of iSnap and iSnap v2. Where iSnap v2 was used in a study, this will be explicitly noted; otherwise, the original iSnap was used.
Table 3.1 A comparison of the two versions of iSnap presented and evaluated in this document.

<table>
<thead>
<tr>
<th></th>
<th>iSnap</th>
<th>iSnap v2 (Highlights)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Chapter 3</td>
<td>Section 3.7</td>
</tr>
<tr>
<td>Hint Algorithm</td>
<td>CTD [Pri16b]</td>
<td>SourceCheck (Chapter 4)</td>
</tr>
<tr>
<td>Insert a Block</td>
<td>All hints indicated with hint buttons: ![hint button image] Presented in a hint dialog:</td>
<td>Insertions indicated with blue outlines or hint buttons (see Figure 3.7): ![insertion button image] Presented in a hint dialog (as in iSnap).</td>
</tr>
<tr>
<td>Delete a Block</td>
<td>![deletion highlight image]</td>
<td>Deletions indicated with magenta highlights (see Figure 3.6): ![deletion highlight image]</td>
</tr>
<tr>
<td>Add/Remove a Variable or Procedure</td>
<td>![add/remove variable image] Not shown</td>
<td></td>
</tr>
<tr>
<td>Reorder Blocks</td>
<td>Not supported</td>
<td>Reorders and Moves indicated by yellow highlights (See Figure 3.6): ![reorder highlight image]</td>
</tr>
<tr>
<td>Help-seeking suggestions</td>
<td>Reminders that help is available when starting.</td>
<td>Reminders that help is available when starting; prompts to check work when the program is run; warnings if too many hints are requested.</td>
</tr>
<tr>
<td>Semesters Used</td>
<td>Spring 2016 – Fall 2016</td>
<td>Spring 2017 – Present</td>
</tr>
<tr>
<td>Datasets Used in</td>
<td>Chapters 5, 6 and 7</td>
<td>Chapters 4 and 6</td>
</tr>
</tbody>
</table>
This chapter was adapted from:

The original text has been modified as follows: A preface has been added to introduce iSnap’s original hint generation algorithm, the Contextual Tree Decomposition (CTD) algorithm [Pri16b], and to discuss the challenges of data-driven hint generation specifically for more open-ended programming problems. Some references and the research questions have been updated to refer to relevant parts of this document.

**Preface**

Chapter 3 introduced iSnap, a novice programming environment that offers data-driven hints and feedback. This chapter details how that feedback is generated, addressing RQ2 (*How can we leverage student data to automatically generate hints for open-ended programming problems?*). It
also investigates how we can evaluate the quality of that feedback, partially addressing RQ3 (How can we evaluate and compare the quality of techniques for generating data-driven hints?). RQ2 focuses specifically on hint generation for open-ended programming problems, found in curricula like the Beauty and Joy of Computing [Gar15], which iSnap is designed to support. These open-ended programming problems pose serious challenges for many automated hint generation techniques (reviewed in Section 2.3 of Chapter 2). Many of these techniques rely on test-cases (e.g. [Per14; RK17; Sin13]), but problems in iSnap often have multiple, loosely ordered objectives, which cannot be assessed automatically. Other approaches are based on the Hint Factory, but as discussed in Chapter 3, there was only 5% overlap among student students’ code states when using iSnap, making the Hint Factory difficult to apply directly.

My first attempt to address this challenge was the Contextual Tree Decomposition (CTD) algorithm [Pri16b], which adapted the Hint Factory to the domain of open-ended programming problems. Rather than matching code states directly, it decomposed students’ code into “subtrees” and matched these smaller pieces of code to other students in the database. These subtrees had much greater overlap between students, allowing CTD to apply the Hint Factory to the subtrees individually. The algorithm took additional measures to ensure that these subtree hints were appropriately contextualized by the code around them and to ensure that hints presented a smooth path towards a correct solution, without backtracking or adding extraneous code. An evaluation of CTD showed that it was capable of generating hints for almost any student that led to a correct solution, and that these hints generally led students towards accomplishing assignment objectives.

While the CTD algorithm represented a major step towards data-driven hint generation for open-ended programming, it also had important limitations, such as its inability to recommend that students move code from one part of the program to another. The SourceCheck algorithm presented in this chapter works to address these limitations. It also provides error-checking feedback in addition to next-step hints. Unlike CTD, which took as input a set of solution traces, including the full history of students’ code, SourceCheck operates on a set of completed solutions. The CTD algorithm was used in iSnap for the studies described in Chapters 5 and 7. However, the current version of iSnap uses the SourceCheck algorithm, as does the evaluation in Chapter 6. Because SourceCheck effectively replaced CTD, this document and the remainder of this chapter focus primarily on the SourceCheck algorithm.

Abstract

In this chapter we present a novel, data-driven algorithm for generating feedback for students on open-ended programming problems. The feedback goes beyond next-step hints, annotating a student’s whole program with suggested edits, including code that should be moved or reordered. We also build on existing work to design a methodology for evaluating this feedback in comparison
to human tutor feedback, using a dataset of real student help requests. Our results suggest that our algorithm is capable of reproducing ideal human tutor edits almost as frequently as another human tutor. However, our algorithm also suggests many edits that are not supported by human tutors, indicating the need for better feedback selection.

4.1 Introduction

A hallmark of Intelligent Tutoring Systems (ITSs) is their ability to support learners with adaptive feedback as they work on problem-solving tasks. In the domain of open-ended computer programming, much research has addressed how this feedback can be generated automatically using reference solutions [ZR15] or data-driven methods [Pie15; Pri16b; RK17]. However, existing techniques (including our own work [Pri16b]) have two notable limitations: the type of feedback they can provide and the methods with which they are evaluated.

Existing work, including the CTD algorithm, has focused almost exclusively on generating next-step hints, suggesting how a student can proceed if they get stuck. Next-step hints make sense in the context of a structured problem-solving task, with well-defined, discrete steps, but they may not always be appropriate in an open-ended programming context. Students may request help for other reasons, such as to verify that code they have written is correct, or to help find a bug in code that does not produce correct output. A more comprehensive feedback generation algorithm is needed to address these concerns. In this work, we present SourceCheck, a novel feedback generation algorithm that builds on existing work to check over a student’s whole program, suggesting useful edits throughout.

While extensive effort has been put into the generation of feedback for programming, efforts to evaluate the quality of this feedback are still underdeveloped. Most existing evaluations are either technical evaluations that focus on how often hints can be generated and theoretical hint quality (e.g. [Pri16b; RK17; ZR15]) or small classroom studies that use case studies (see Chapter 2, Section 2.3.3 for a detailed review of evaluations of data-driven hints). Ideally, we would employ controlled studies to evaluate the impact of feedback on students’ course outcomes, as was done by Stamper et al. in their evaluation of data-driven hints in the Deep Thought logic tutor [Sta13]. However, as discussed in Chapter 5, programming hints can vary widely in quality and low-quality hints may deter students from later asking for help when they need it. A meaningful first step would therefore be to better understand and evaluate the quality of the feedback we generate. Piech et al. [Pie15] suggest evaluating automatically generated hints for programming by comparing them to “gold standard,” expert-authored hints. We build on this method to evaluate our feedback algorithm, comparing it to human-authored feedback.

Our initial results show that SourceCheck’s feedback has good overlap with that from human tutors. However, SourceCheck also produces much more feedback than human tutors, and much of
this feedback is not represented in human tutor feedback. This suggests that SourceCheck has good potential but that more work is needed to select targeted feedback from potential suggestions.

4.2 Feedback Generation

This section investigates RQ2: \textit{How can we leverage student data to automatically generate hints for open-ended programming problems?} To address this question, we present the SourceCheck data-driven hint generation algorithm. At a high level, SourceCheck works on a simple premise. To generate feedback for a student on a given assignment, we use a two-step process. First, in the \textit{Solution Matching} step, we look at previously submitted, correct student solutions for that assignment and select the one that best matches that student's code. Then, in the \textit{Edit Inference} step, we extract the edits that separate the student's code from the correct solution and present these as feedback. This idea dates back to the original Hint Factory [BS08] and was successfully implemented by Rivers and Koedinger for programming hints [RK17]. Rather than changing the fundamentals of this idea, we present techniques for improving both steps of the process. These improvements center on the understanding that students' solutions are diverse and often include much correct code that does not directly match a known solution because of small changes in structure. SourceCheck attempts to make use of this code, and can suggest \textit{moving} code in addition to inserting and deleting it.

SourceCheck takes as input a set of complete, correct prior student solutions for an assignment and a snapshot of code from a new student requesting a hint. As in previous work, we represent both as an abstract syntax tree (AST), a directed, rooted tree where each node is labeled to represent a program element, such as a function call, control structure or variable, and the hierarchy of the tree represents how these elements are nested together. To each AST we apply simple canonicalization to reduce syntactic complexity while preserving semantic meaning, as described in [Pri16b]. SourceCheck outputs a set of edits, (insertions, deletions, moves and reorders) that can be used to annotate the student's code with feedback. While this feedback can include next steps hints in the form of insertions, it also highlights potential errors and provides reassurance that unannotated code is likely correct.

4.2.1 Solution Matching

Most hint generation algorithms for programming select a goal solution by finding the “closest” solution to the student's current code, determined by some distance metric. Researchers have used string edit distance [RK17] and approximations of tree edit distance [ZR15], though more complex metrics have been proposed [Paa15]. The problem with edit distances, however, is that they heavily penalize differences in the position of code fragments [Mok13; Paa15]. For example, swapping the
order of two independent subroutines in a program does not affect its semantic meaning, but this movement is treated as a large set of deletions and insertions by edit distance algorithms.

Mokbel et al. suggest addressing this by fragmenting each AST into subgraphs, pairing similar subgraphs from the two ASTs, and computing their distance independently [Mok13]. We build on this idea, along with our previous work decomposing ASTs using root paths [Pri16b], to produce a distance metric designed specifically for code. The root path of a node $n$ in an AST is the sequence of node labels on the path from the root of the AST to $n$. Multiple nodes in an AST will have the same root path if they and each of their respective ancestors have matching labels, such as two calls to the same function in the same block of code.

Given ASTs $A$ and $B$, consisting of nodes $\{a_1, \ldots, a_{|A|}\}$ and $\{b_1, \ldots, b_{|B|}\}$ respectively, SourceCheck produces a matching, $M = [(a_i, b_j), \ldots]$, pairing nodes from $A$ to nodes from $B$, and a cost $C$ for the mapping. Nodes can only appear in one pair, and some nodes may be left unmatched. First, we iterate over each root path in $A$, from shortest to longest path. For a given root path $r$, let $A_r$ and $B_r$ be the set of nodes in $A$ and $B$ respectively with root path $r$. Let us define $c(n)$ as the child-sequence of $n$, or the sequence of node labels of the immediate children of $n$. For each pair of nodes $a_{ri} \in A_r$ and $b_{rj} \in B_r$, we compute the pairwise distance between their child-sequences, $d(c(a_{ri}), c(b_{rj}))$. This is used to match nodes with the same root path and similar children.

For the distance function $d$, we could use a string edit distance, such as Levenshtein distance, since AST child-sequences are just sequences of node labels. However, SourceCheck is designed to match incomplete student code ($A$) to complete solutions ($B$), so for $d$ we use a “progress” function that measures how much of $c(a_{ri})$ represents progress towards $c(b_{rj})$. Our progress function is similar to an edit distance, but it is intentionally asymmetrical and penalizes deletions (student’s code not found in the solution) much more than insertions (solution code not yet found in the student’s code). Additionally, our progress function identifies insertion/deletion pairs with the same label and treats these as a “reorder,” which has a much lower cost, distinguishing between code that should be deleted and code that is out of order.

SourceCheck calculates the pairwise distances for all nodes in $A_r$ and $B_r$ and then uses the Hungarian algorithm to select the set of pairs of minimum total cost, which it adds to the mapping, $M$. This cost is added to $C$. This procedure is performed for each root path in $A$ to determine the total mapping and cost. To select a target solution $T$ for a student’s current code $S$, SourceCheck simply finds the solution with the minimum mapping cost. The result is a target solution that maximizes the number of nodes in the student’s code which can be reasonably mapped to nodes in the target solution.
4.2.2 Edit Inference

Once a target solution $T$ has been identified for a student’s code $S$, SourceCheck identifies a set of edits that can bring the student closer to this solution. In previous work, this is accomplished by selecting the top-level applicable edit [RK17] or following edits from previous students [Pri16b]. Instead, we use the mapping $M$ between the student’s AST and the target solution to calculate a more precise set of edits between $S$ and $T$. These edits take the form of Moves and Reorders, along with traditional Insertions and Deletions, determined as follows:

**Deletions**: First, all nodes $s \in S$ without a pair in $M$ are marked for deletion; however, these nodes may be reused in the final step of the algorithm.

**Moves**: Next, we consider all pairs $[s_i, t_i] \in M$. Let $P(n)$ denote the parent of the node $n$ in its AST. If $[P(s_i), P(t_i)] \notin M$, this means $s_i$ is under the wrong parent node, so we mark $s_i$ to be moved under $p$, where $[p, P(t_i)] \in M$, at an index corresponding to that of $t_i$. If no such $p$ exists, this means that the appropriate parent has not yet been added to $S$. We still mark $s_i$ for movement, but we cannot specify a destination.

**Reorders**: Next, we ensure that the children of $s_i$ are in the correct order. We do this by identifying the set of matching child pairs $[c_s, c_t] \in M$ such that $P(c_s) = s_i$ and $P(c_t) = t_i$. For each node $c_s$, if the node’s index among its siblings is different than that of its pair, $c_t$, we mark it for reordering.

**Insertions**: Any node $t \in T$ which has no pair in $S$ is marked for insertion. If $P(t)$ has a pair in $S$, this pair is used as a parent. If $P(t)$ has no pair in $S$, we do not yet have a place to insert $t$. We still mark $t$ for insertion, since it may be useful in the next step.

**Combining Insertions and Deletions**: If a node is deleted in one place and a node with the same label is inserted in another, this may actually represent a Move or Reorder. We identify pairs of Deletions and Insertions with the same label and replace these with an appropriate Move or Reorder. This is a key feature of SourceCheck that encourages a student to use existing code, rather than deleting and re-inserting it.

Using the mapping $M$, SourceCheck is able to infer more semantically meaningful edits, such as Moves and Reorders, which convey more information than their component insertions and deletions would alone. The Deletions that remain indicate likely errors to the students, and the Moves and Reorders suggest areas in need of editing. Any node not marked with an edit has been “checked” and likely represents correct code.

4.3 Methods

Our evaluation focuses on measuring the quality and appropriateness of SourceCheck’s feedback by comparing it to human tutor feedback. This is in contrast to previous technical evaluations [Pri16b; RK17] that used theoretical measures of hint quality and availability. Instead, we extend the
work of Piech et al., who assessed feedback quality by comparing hint policies with “gold standard,” human-authored, expert hints on small, constrained programming problems (4-6 lines of code for an ideal solution) [Pie15]. However, for the more complex problems we investigate here (about twice as many lines of code), we argue that it is not realistic to define a single best “gold standard” hint for a given code snapshot. There may be many useful ways a tutor can advise a student, so it is more reasonable to measure the similarity of human and algorithmic feedback, rather than whether they match exactly. We build on the “gold standard” method to compare the feedback of human tutors and SourceCheck in a more nuanced way. We use this method to investigate RQ3: How can we evaluate and compare the quality of techniques for generating data-driven hints? We focus on the following study-specific research questions:

**RQ3.1** How well does SourceCheck’s feedback agree with ideal human tutor feedback?

**RQ3.2** How does the agreement between SourceCheck and a human tutor compare to the agreement between human tutors?

We evaluated SourceCheck in the context of an introductory computing course for non-majors, consisting of 51 students, held at a research university during the Spring 2017 semester. During the first half of the course, undergraduate teaching assistants (TAs) facilitated Snap! programming labs derived from the Beauty and Joy of Computing (BJC) AP Computer Science Principles curriculum [Gar15] (available at bjc.edc.org). The course includes three in-lab programming assignments, completed with TA help available, interleaved with three homework assignments, completed independently. Students programmed using iSnap v2 (see Chapter 3), an extension of the block-based, novice programming environment Snap! [Gar15]. iSnap supports students working on open-ended assignments with data-driven, on-demand hints [Pri16b].

We selected one homework assignment (Squirrel) and one in-lab assignment (GuessingGame) for analysis. In Squirrel, students draw a square-shaped spiral using loops, variables and a custom block (function), and a typical solution is around 10 lines of code. In GuessingGame, students create a simple game in which the player must guess a random number using loops, variables, conditionals and user input, and a typical solution is around 13 lines of code. We built a dataset of student hint requests on GuessingGame and Squirrel to serve as authentic scenarios for evaluating SourceCheck. We sampled up to two hint requests from each student. Where possible we sampled one request from the first half of their working time and one from the second half to avoid overly similar samples. We also ensured that at least 30 seconds and one code edit occurred between sampled hint requests. We sampled hints from 14 and 15 students on Squirrel and GuessingGame for a total of 22 and 29 hints respectively, 51 altogether.
4.3.1 Human Feedback Generation

For each hint request, we extracted a snapshot of the student’s code at the time of the request. Importantly, these snapshots represent code for which real students requested help, making them an ideal sample on which to evaluate SourceCheck. We did so using a post hoc Wizard-of-Oz-style experiment. The first two authors, who were familiar with the assignments and context, acted as human tutors and manually generated feedback for each selected snapshot. The two tutors were graduate students in Computer Science who were domain experts but not teaching experts, making them similar to most course TAs for advanced computing courses.

When generating feedback, the human tutors attempted to offer pedagogically useful feedback, but they were limited to communicating their feedback using the edits defined in Section 4.2.2. These edits also had to be independent, meaning no edit could be dependent on the student following another suggested edit (e.g. suggesting inserting both a for-loop and the body of the loop). Tutors crafted these edits with the understanding that the edits would be (theoretically) presented to students without further explanation or any guarantee of further feedback requests. These limitations forced the human tutors to generate feedback that could be provided through the same user interface that SourceCheck would use, as in a Wizard-of-Oz experiment, allowing us to directly compare human and algorithmic feedback. Tutors generated their feedback based on the student's current code at the time of the hint request, using previous snapshots of the student's code for context. However, tutors did not have access to a student’s code after the hint request or the student’s final solution. While the two tutors generated feedback independently, they first practiced on a dataset with the same assignments from another semester and compared results to ensure a consistent understanding of the feedback guidelines. The tutors generated feedback in a two-phase process:

**Phase I**: Tutors identified the edit(s) they would recommend to best support the student’s current goal and promote learning. The edit(s) should convey a single idea.

**Phase II**: Tutors envisioned a correct solution that most closely matched the student’s current code and identified all edits that would bring the student closer to this solution.

Phase I allows us to measure how well the algorithm reproduces ideal, targeted tutor feedback, addressing RQ3.1. In Phase II, tutors generate a large set of all applicable edits, just as SourceCheck does, allowing us to directly compare algorithmic and human feedback, addressing RQ3.2.

4.4 Analysis and Results

To quantify the overlap between two sets of feedback for a given snapshot, we define feedback generation as the process of labeling each node of an AST with an edit (Delete, Reorder, Move
or nothing) and generating a set of Insertions (each consisting of a type of node to insert and an index in the AST at which to insert it). Under this definition, we can treat SourceCheck as a classifier and evaluate its ability to predict the feedback provided by tutors. We measure classification success for each type of edit separately, treating it as a binary classification task. For Deletions and Moves, we consider each node of each snapshot in our dataset to be a classification instance, where both the tutors and SourceCheck have labeled the node. Successful classification occurs when SourceCheck produces the same label as the tutor. Each Insertion provided by either the tutors or SourceCheck for a given snapshot is also considered a classification instance, where both the tutors and SourceCheck have either included or not included that Insertion. Since Reorders were rarely suggested by SourceCheck and were never suggested by tutors, we exclude them from analysis.

Treating feedback generation as a classification task allows us to address RQ3.1 and evaluate the extent to which SourceCheck agrees with (predicts) the feedback of human tutors. The results of this evaluation would be difficult to interpret without a baseline for comparison. Therefore, we also define a “Tutor Classifier” (TC), which predicts feedback from Tutor 1 using the feedback collected in Phase II from Tutor 2, and vice versa. Since tutors generate a full set of applicable edits in Phase II, just like SourceCheck, we can directly compare the SourceCheck and TC classifiers. This allows us to address RQ3.2, comparing the agreement of human and algorithmic feedback with that of two humans. We would not generally expect an algorithm to predict human tutor feedback better than it would be predicted by another tutor, so TC provides a high performance target.

4.4.1 Results

We first look at predicting the targeted feedback that tutors provided in Phase I. Figure 4.1 shows the percentage of the tutor edits that were also generated by SourceCheck and TC, or the recall of both predictors. We did not observe large differences between prediction success for edits generated by the two human tutors, so we report their results in aggregate. While Deletions were fairly rare, SourceCheck performs quite poorly at predicting them on both assignments. However, SourceCheck predicts 46% and 47% of tutor Moves and Insertions respectively on GuessingGame, and 69% and 80% of Moves and Insertions for Squirrel, where it even outperforms TC. Totalling all edits, SourceCheck had a recall of 0.45 and 0.57 on GuessingGame and Squirrel respectively, while TC achieved 0.59 and 0.65.

An important limitation of recall is that it only considers how many of the tutor edits were successfully predicted, and not how many “guesses” (suggested edits), it took to do so. To understand how much of SourceCheck’s feedback agrees with tutor feedback, we must compare it against all tutor edits collected in Phase II. Figure 4.2 shows the recall (top) for SourceCheck and TC over all Phase II edits, as well as the precision (bottom), or the percentage of SourceCheck and TC edits that agreed with human tutor edits. We see very similar trends for recall across Phases I and II, implying
that both SourceCheck and TC predict “ideal” (Phase I) and “possible” (Phase II) edits at similar rates. Totalling all Phase II edits, SourceCheck had a recall of 0.41 and 0.41 on GuessingGame and Squiral respectively, while TC achieved 0.57 and 0.54.

However, SourceCheck’s precision is much lower, particularly for GuessingGame, where SourceCheck suggests more of every type of edit, for a total of over 50% more suggested edits. SourceCheck generated on average 10.7 and 6.4 edits per snapshot on GuessingGame and Squiral respectively, compared to 6.3 and 5.2 edits per snapshot for the tutors’ Phase II edits. Despite this low precision, SourceCheck is not simply suggesting edits everywhere in the code and getting a few correct by chance. It correctly suggests no edit for 1092/1238 (88%) of GuessingGame AST nodes where the tutors also did not suggest an edit in Phase II and for 662/703 (94%) of Squiral nodes. Totalling all edits, SourceCheck had a precision of 0.27 and 0.38 on GuessingGame and Squiral respectively, while TC achieved 0.57 and 0.54.

4.4.2 A Closer Look

We manually investigated edits on which the human tutors and SourceCheck disagreed, and in this section we present some common causes of disagreement:

Variables: We noticed that many disagreements were over variable assignments and references. For example, most of the Phase I deletions that SourceCheck failed to predict were instances of a tutor deleting a variable, such as when a student used the wrong variable in an expression. This is largely due to SourceCheck’s canonicalization process [Pri16b], which currently gives all variables the same label, making them indistinguishable. This simplification makes code matching easier, but clearly a more robust solution is needed.

1Note that because with TC, Tutor 1 predicts Tutor 2 and vice versa, the precision and recall of TC in Phase II will be the same, and this value indicates percent agreement.
Supporting Unusual Code: Many times, a tutor suggested an edit, such as deleting an unneeded control structure, that would lead the student away from a potentially confusing program state. In many of these cases, SourceCheck found some solution which used this unusual code correctly and instead suggested how the student could do the same. We view this behavior as a design choice, rather than a flaw per se, but it is worth investigating when this behavior would lead to its intended effect of supporting unconventional solutions, and when it would lead to confusion.

Code Variability: The assignments we analyzed were complex enough to allow the student to make a number of small design choices, such as how to reset the sprite and canvas before drawing the “Squirrel” in the Squirrel assignment. Often, the tutor and the target solution chosen by SourceCheck made different, correct suggestions. This also occurred between human tutors, emphasizing that disagreement with the tutors does not always indicate poor feedback.

Human Traits: Sometimes the human tutors were able to infer information from natural language in a student’s code that influenced their feedback in a way that would not be possible for SourceCheck. For example, the name of a variable might imply how it is intended to be used (e.g. “randomNumber”). This sometimes led to very different edits from SourceCheck and the human tutors. On the other hand, humans are also capable of making careless errors, and our tutors sometimes simply forgot to suggest a small, useful edit in Phase II, which SourceCheck remembered.
4.5 Discussion

RQ3.1: How well does SourceCheck's feedback agree with ideal human tutor feedback? SourceCheck agrees with approximately half of ideal tutor feedback provided in Phase I, almost as much as another human tutor, with SourceCheck achieving a recall 76% and 88% as high as TC on GuessingGame and Squirrel respectively. This does not necessarily mean that SourceCheck's feedback is almost as good as a tutor's. It is possible that when SourceCheck's feedback diverges from a tutor's, it does so in a less useful way than when another tutor does so; however, this is difficult to investigate without some direct measure of hint quality (e.g. as presented in Chapter 5). For now, we can say that these results suggest good potential for data-driven feedback generation, in that ideal tutor feedback is frequently contained in the set of edits generated by SourceCheck.

RQ3.2: How does the agreement between SourceCheck and a human tutor compare to the agreement between human tutors? Our results for RQ3.2 are mixed. In Phase II, SourceCheck was 72-76% as likely to agree with a given tutor's edit as another tutor was on GuessingGame and Squirrel (as measured by recall). However, a given tutor was only 47-70% as likely to agree with SourceCheck's edit as with another tutor's edit (as measured by precision). This is largely because SourceCheck generated more total edits than the tutors did, especially on GuessingGame. This lack of precision seems to be the largest difference between SourceCheck and human tutors. Even if SourceCheck can produce quality feedback, the benefit to the student might be lost if it is hidden among less useful suggestions. Additionally, as discussed in Chapter 5, students seek less help after receiving poor quality hints. A critical direction for future research will be how to select feedback once a set of possible edits has been generated.

It is also worth noting that our two human tutors had relatively low agreement. Comparing all suggested Phase II edits, we see that they have a 54% and 57% agreement on the GuessingGame and Squirrel assignments respectively. In fact, tutors only agreed completely on 8 out of 22 Squirrel snapshots (36%) and 7 out of 29 GuessingGame snapshots (24%) in Phase I. This suggests the assignments we studied truly are open-ended, since tutors often disagreed on the best path forward, though we cannot make any strong claims using our human data because it was generated by the authors. This supports our choice to measure agreement using the similarity of edits, rather than using a single, best “gold standard” hint, as was done by Piech et al. on simpler assignments [Pie15].

4.6 Conclusion

In this work, we have presented SourceCheck, an algorithm for automatically generating data-driven feedback for students working on open-ended programming problems. SourceCheck builds on existing methods [Mok13; RK17] to improve the processes of selecting a target solution from a set of correct solutions and inferring edits to get the student to that solution. It does so with a code-specific
matching function and more semantically meaningful suggested edits: Moves and Reorders. We have also presented a method for evaluating automatically generated feedback by comparing it to feedback generated by human tutors playing the same role. We extend existing methods [Pie15] by using a dataset of real student help requests to ensure authenticity and by formulating the problem as a prediction task, allowing us to compare the similarity among an algorithm and multiple human tutors. This allows us to envision the high standard of an algorithm that is as similar to human tutors as human tutors are to each other. We show that SourceCheck approaches this target in some ways and falls well short in others.

The version of iSnap (v2) evaluated in this study was updated to include SourceCheck feedback, and we envision a number of practical application for the algorithm. In busy classrooms, large-scale MOOCs and informal learning settings, instructors are often absent or unavailable. The on-demand feedback provided by SourceCheck can keep students going when they get stuck and would otherwise give up. SourceCheck could also be used to identify potential struggling students in real-time, based on their distance to a known solution. Both SourceCheck and our evaluation methodology were designed to scale to the larger, more complex programming problems found in real classrooms. This will require SourceCheck to support a greater diversity of student code, which will require a larger dataset of correct solutions for matching.

This work also has clear limitations. We only used two tutors to generate human feedback, and the authors who served as tutors were not pedagogical experts and had limited teaching experience. While their experience is on par with many graduate computing TAs, results may be different with experienced teachers. Additionally, despite efforts at objectivity, the tutors’ familiarity with each other and with SourceCheck may have biased their feedback. Our work is also limited by the small sample of assignments and hint requests we investigated, especially given that our results were quite different for GuessingGame and Squirrel. Finally, the methods presented here do not lend themselves to traditional statistical testing, making it difficult to make claims about true differences in recall and precision. Our methods only speak to the relative similarity of algorithmic and human tutor feedback, but this does not directly assess feedback quality.

This work opens many avenues for future work. Our results suggest a number of ways SourceCheck could be improved, such as a method for selecting which of the generated edits are most useful to show the student. Future work could also explore how to expand data-driven ITS feedback for programming beyond edit-based hints, towards richer descriptions or explanations. Our results also raise questions about the consistency of human feedback on open-ended programming problems, and future work should determine how much agreement can be expected among human tutor feedback. Lastly, the methods presented here can be used to evaluate, compare and benchmark other feedback generation techniques, giving researchers a better understanding of their strengths and weaknesses.
CHAPTER 5

STUDY 1: THE EFFECT OF HINT QUALITY ON HELP-SEEKING BEHAVIOR

This chapter was adapted from:

The original text has been modified as follows: The related work subsections have been removed from the Introduction and integrated into Sections 2.2 and 2.4.2 of Chapter 2, and the section describing iSnap has been removed, since this is covered in Chapter 3. Some references and the research questions have been updated to refer to relevant parts of this document. The “Quality” metric used in this chapter has been renamed to QUALITYRATING to differentiate it from the QUALITYSCORE presented in Chapter 6. Note that this study evaluates the original iSnap, which used the CTD algorithm [Pri16b], not the SourceCheck algorithm presented in the previous chapter.

Abstract

Much research in Intelligent Tutoring Systems has explored how to provide on-demand hints, how they should be used, and what effect they have on student learning and performance. Most of this
work relies on hints created by experts and assumes that all help provided by the tutor is correct and of high quality. However, hints may not all be of equal value, especially in open-ended problem-solving domains, where context is important. This work argues that hint quality, especially when using data-driven hint generation techniques, is inherently uncertain. We investigate the impact of hint quality on students’ help-seeking behavior in an open-ended programming environment with on-demand hints. Our results suggest that the quality of the first few hints on an assignment is positively associated with future hint use on the same assignment. Initial hint quality also correlates with possible help abuse. These results have important implications for hint design and generation.

5.1 Introduction

A hallmark of Intelligent Tutoring Systems (ITSs) is their ability to provide next-step help to students while they work on problems. A variety of work has explored how such help can be generated [BS08; RK17], how it should be ideally used [Ale06], and its impact on student performance and learning [CA01; Sta13]. However, this body of work generally makes the assumption that all help provided by the tutor is equally useful and of high-quality. While some work distinguishes between levels of hints (e.g. pointing, teaching and bottom-out hints), each level represents “correct” advice, and is therefore assumed to be useful.

In this chapter, we argue that this assumption is not always valid. Increasingly, ITSs employ data-driven hint generation to reduce the need for expensive expert modelling and to target domains where such modelling is difficult [Pri16b; RK17]. While data-driven hints can be designed to meet quality criteria, (e.g. leading to a valid solution), this does not necessarily mean that every hint is useful to every student. In fact, a comparison of various hint generators found high variance in their agreement with expert hints [Pie15]. Even hints engineered by experts are still subject to the “expert blindspot,” where expert domain knowledge fails to translate into pedagogically useful feedback. Further, domains such as open-ended programming are so complex that even the best hints may not account for the diversity of possible solution paths a student might pursue [Pri16b]. Outside of ITSs, researchers have found that feedback has widely varying effects on task performance, often negative, depending on a multitude of factors [KD96], so we might expect similar variability in ITSs.

Because hint quality in open-ended problem-solving domains can be uncertain, it is important to understand its impact on students. In this work, we investigate the effect of hint quality on students’ help-seeking behavior in an open-ended programming environment with on-demand hints. Help-seeking plays an important role in student learning in ITSs, and students frequently fail to make good use of help facilities [Ale06]. We present evidence that hint quality does affect both positive and negative help-seeking behaviors.
Figure 5.1 A student's code is annotated with hint buttons (left). Upon clicking a button, a hint window suggests a change to the student's code (right).

5.2 Methods

The dataset analyzed in this chapter comes from students using the original version of iSnap [Pri17d], which is described in detail in Chapter 3. This study was conducted during an introductory computing course for non-majors, consisting of 68 students, held at a research university during the Fall 2016 semester. During the first half of the course, the lab sections taught the Snap! programming language through a curriculum based on the Beauty and Joy of Computing (BJC) [Gar15]. The course included three in-lab programming assignments, which were completed in class with help available from undergraduate teaching assistants, interleaved with three homework assignments, which were completed independently. Students completed all work in iSnap, and data-driven hints were available on the 2nd, 3rd, 4th and 5th assignments, shown in Table 5.1. Hints were generated using data from the Spring 2016 semester with the CTD algorithm [Pri16b].

Before the first hint-enabled assignment, a researcher briefly introduced the hint interface, explaining that the students were encouraged to use hints without any penalty to their grades and that the hints may not be perfect. Figure 5.1 shows how hints were requested and displayed in iSnap's interface. iSnap reminded students that help was available at the start of each assignment. There was no limit on the number of hints a student could request. iSnap recorded complete logs of hint requests and snapshots of students' code after each edit. Students did not login to use iSnap, so unfortunately we cannot analyze a given student's work over multiple assignments due the anonymous logs.

Using the log data, we identified 642 hint requests (when a next-step hint button was clicked) across the four hint-enabled assignments. For each hint, we calculated how long the hint window
was viewed and how long after viewing it the student made their next code edit. Because each hint suggested a specific edit to the student's code, we were also able to label each hint as “followed” if the student's code reflected the suggestion within their next 5 edits. Some students viewed a single hint multiple times in a short time span. For example, a student might browse through multiple hints and then return to one to implement it. We considered two hints duplicates for a given student if they suggested the exact same edit, and the student viewed the two hints within 60 seconds, or without changing their code in between. We merged duplicate hint requests to produce 542 final hint requests, which we used in our analysis. We considered these merged hints to be “followed” if any of their component hints were followed.

5.2.1 Submission Grading and Hint Rating

For each assignment, we identified 5-6 assignment objectives based on the instructions and the course instructor’s grading rubric. Two researchers independently graded each submission, marking each objective as complete or incomplete. The graders discussed disagreements to produce a final grade for each submission, calculated as the percent of objectives completed. Interrater reliability is given for each assignment by Cohen’s kappa ($\kappa$) in Table 5.1.

Table 5.1 For each assignment with hints available, the number of attempts submitted (N), the number of attempts with one or more hint requests (H), the mean grade with SD, and the agreement ($\kappa$) for the graders.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Type</th>
<th>N</th>
<th>H</th>
<th>Mean Grade (SD)</th>
<th>Grader $\kappa$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polygon Maker</td>
<td>In-lab</td>
<td>65</td>
<td>11 (17%)</td>
<td>97.5% (9.0%)</td>
<td>0.62</td>
</tr>
<tr>
<td>Squiral</td>
<td>HW</td>
<td>60</td>
<td>18 (30%)</td>
<td>78.6% (23.0%)</td>
<td>0.79</td>
</tr>
<tr>
<td>Guessing Game 1</td>
<td>In-lab</td>
<td>66</td>
<td>13 (20%)</td>
<td>95.2% (10.8%)</td>
<td>0.81</td>
</tr>
<tr>
<td>Guessing Game 2</td>
<td>HW</td>
<td>59</td>
<td>13 (22%)</td>
<td>89.8% (17.8%)</td>
<td>0.63</td>
</tr>
</tbody>
</table>

We also developed a detailed hint rating rubric to quantify the quality of iSnap hints, which is given in Appendix B. The hint rubric has 3 attributes, each rated 1, 2, or 3, with higher scores being better: Relevance, how likely the hint is to address one of the student’s current goals; Progress, how well the hint moves the student’s current code to a correct solution without removing useful code; and Interpretability, how easily a novice could understand the intention and value of the hint.

Across all four assignments, a total of 55 assignment attempts included at least one hint request. These may include attempts from the same student on multiple assignments. Of these 55 attempts, 39 (70.9%) included a second hint request, and 29 (52.7%) included a third request. We limited
our analysis to the first 2 hints in an attempt, since barely half of the relevant attempts included a third hint request. This resulted in a subset of 94 hints that were selected for rating. Two authors independently rated these hints on each attribute. The raters had access to a snapshot of the student’s code when the hint was shown. Disagreements were discussed and resolved to produce final ratings. The hint raters achieved squared-weighted Cohen’s kappas of 0.857, 0.756 and 0.685 for Relevance, Progress and Interpretability, respectively. This indicates substantial agreement, and while we make no claims that the ratings are objectively “correct,” the multiple raters helped to ensure consistent ratings across hints.

5.3 Analysis

We structured our analysis around RQ4: *What factors impact students’ willingness to seek and use programming help?* We focused on the following study-specific specific research questions, which are addressed in the remainder of this section:

**RQ4.1** How frequently did students use and follow hints in iSnap?

**RQ4.2** How did hint use relate to student performance on assignments?

**RQ4.3** How did a given hint’s quality affect whether it was followed?

**RQ4.4** How did the quality of students’ first hints affect their future help use?

5.3.1 Hint Use in iSnap

With 542 unique hint requests over 250 assignment attempts, students received on average 2.2 hints per assignment. This average is misleading, however, since the number of hints requested per student varied widely. The percentage of students who requested at least one hint on a given assignment ranged from 16.9% to 30.0%, with somewhat higher usage on homework assignments. Despite being introduced to the hint system in class and receiving reminders from iSnap before starting work, the majority of students never used the help on a given assignment. This is consistent with studies of help-seeking in other domains, where hint-usage rates were low. For example, Aleven and Koedinger found hints were used on 22-29% of steps in the PACT Geometry Tutor [AK00].

Across assignments, 55 attempts (22%) included at least one hint request. Many of these attempts included only 1-2 hint requests (47.3%), but those with 3 or more requests had a median of 16 hint requests per attempt, with a maximum of 68. Of all hints requested, 41.3% were followed. An additional 16.4% of hints were unfollowed but occurred within 1 minute of a later hint that was followed, indicating that a majority (57.5%) of help requests were closely followed by some resolution. For attempts on any assignment which included at least 1 hint request, we compared
the number of hints requested with the percent of hints followed. There is a significant, positive Spearman correlation between the two values ($\rho = 0.552; S = 12415; p < 0.001$), indicating that students who asked for more hints on a given assignment were also more likely to have followed them.

When a student chose to follow a hint, they took a median 9 seconds after viewing the hint before making their next edit, compared with 19 seconds for unfollowed hints. A Mann-Whitney $U$ test\(^1\) showed the difference was significant ($U = 18164; p < 0.001$). Students also viewed the hint window for hints they ended up following for less time (Med=5) than those they did not (Med=6), and the difference was significant ($U = 31550; p = 0.026$). This suggests that students needed less time to process a hint when they ultimately followed it.

5.3.2 Help Use and Performance

The course from which we collected our data did not include a pre- or post-test on Snap! programming, so we have no measure of learning. Instead, we focus on programming performance on the assignments to evaluate the impact of hints. While this is not a substitute for learning, it does provide insight into how well the hints played their intended role of scaffolding students during programming. Due to the presence of TAs for the in-lab assignments and the high grades students achieved (see Table 5.1), we focus on the two homework assignments. We defined meaningful hint usage as requesting and following at least one hint, and we labeled attempts with meaningful hint usage as F1, and those without as F0.

Figure 5.2 shows violin plots of the grade distributions for both homework assignments, comparing F0 and F1 attempts. We can see that F1 attempts perform at least as well as their counterparts, with equal median grades and higher average grades on both assignments. This was not a controlled study, so we can make no claims about the direct effect of hints on performance. However, previous studies have found that students with lower prior knowledge are more likely to request help [WW99], so if the hints had no effect, we would expect those using them to perform worse, rather than at least as well.

One of the justifications for including hints on a homework assignment is that they allow students to progress when they are “stuck” and do not know how to proceed, allowing them perform adequately on an assignment. We define adequate performance as missing no more than 1 assignment objective (out of 5-6 total). By this definition, 100.0% (9/9) of F1 students performed adequately on Squirrel, compared 60.8% (31/51) of F0 students. On Guessing Game 2, 87.5% (7/8) of F1 students performed adequately, as did 88.2% (45/51) of F0 students. Put another way, homework attempts with meaningful hint usage (F1) almost always achieved adequate grades (16/17), and many of the remaining homework attempts (F0) did not achieve adequate grades (26/102). However,

\(^{1}\)The Mann-Whitney $U$ test was used were data were not normally distributed.
different homework assignments do show different impacts of hint use.

The prevalence of students with poor performance and no meaningful hint usage indicates help avoidance [Ale06], where students could benefit from using help but choose not to. While there are many things we could do to encourage students to start using the help system, the remainder of this section focuses on how hint quality affects hint use, once students have started using the help.

### 5.3.3 Hint Quality and Immediate Hint Use

Our dataset of rated hints consists of 94 hints, the first and second hints requested for each assignment attempt. These were rated on three attributes from 1-3: Relevance, Progress and Interpretability, as described in Section 5.2.1. While these attributes were selected to be relatively independent, we first inspected the relationships between them. Each attribute pair showed a significant positive Spearman’s correlation: Relevance and Progress ($\rho = 0.664$; $S = 46454$; $p < 0.001$); Progress and Interpretability ($\rho = 0.777$; $S = 30842$; $p < 0.001$); Relevance and Interpretability ($\rho = 0.632$; $S = 50919$; $p < 0.001$). Due to the high correlations, we also computed a QUALITY RATING attribute, the sum of the other three, ranging from 3 to 9, and we use this for most of our analysis.

An assumption that underlies our investigation is that the quality of data-driven hints is not uniform. We attempted to verify this empirically on our sample of rated hints. While the median QUALITY RATINGS for the first and second hints received were high, 7 and 8 respectively, the standard deviations were also high: 2.1 and 2.2, respectively. Out of all 94 rated hints, 30 (31.9%) were rated below the middle possible rating of 6. This suggests that the quality of our sample did vary enough

---

**Figure 5.2** Violin plots comparing grade distributions for F0 and F1 attempts at both the Squirrel (left) and Guessing Game 2 (right) homeworks. Inside each plot is a boxplot, and a red dot indicating average grade.
to potentially impact students.

We then investigated the relationship between hint QUALITYRATINGS and whether or not a hint was followed. This was in part to confirm the validity of our ratings, since we hypothesize that valid hint ratings would be predictive of whether or not a hint was followed. However, we also note that this is an assumption. The theory of ideal help use put forward by Aleven et al. [Ale06] suggests a student should evaluate the helpfulness of a hint before applying it, but to our knowledge this has not been investigated empirically in ITSs.

First hints had a median QUALITYRATING of 7, and those with at least that rating were four times as likely to be followed (14/34 = 41.2%) as those with lower ratings (2/21 = 9.5%). Second hints had a median QUALITYRATING of 8, and those with at least that rating were three times as likely to be followed (11/21 = 52.4%) as those with lower ratings (3/18 = 16.7%). Put another way, first hints that were followed had higher QUALITYRATINGS (n = 16; Med = 8) than unfollowed hints (n = 39; Med = 7), and a Mann-Whitney U test showed that the difference was significant (U = 437.5; p = 0.018). For second hints, followed hints were also rated higher (n = 13; Med = 8) than unfollowed hints (n = 26; Med = 6.5), and the difference was significant (U = 276.0; p = 0.001).

We conclude that hint quality strongly affects the likelihood that a student will follow a hint, and that the QUALITYRATING metric is reasonable. Students also have quite high standards for the hints that they will follow, with median 8 and 8.5 QUALITYRATINGS out of a possible 9, for followed first and second hints, respectively. While the magnitudes of our ratings are somewhat relative, this is still strong evidence that students are capable of noting and disregarding flawed hints.

5.3.4 Hint Quality and Future Hint Use

It is encouraging that students can identify and disregard lower quality hints, but what effect does encountering such hints have on a student’s future hint use? To answer this question, we labeled each attempt that used hints as Low, Medium or High hint usage, where each label corresponds to a 3-quantile of the number of hints requested for a given assignment. For example, hint requests for the Guessing Game 1 had 3-quantiles [3, 11], so the labels were n ≤ 3 (Low), 3 < n ≤ 11 (Medium) and 11 < n (High), where n is the number of hints requested. Table 5.2 shows average attribute ratings for the first and second hints per attempt for each usage label. There is a consistent trend of ratings increasing with label, with only one exception for the Interpretability of the first hint.

There was a significant, positive Spearman correlation between an attempt’s label and the QUALITYRATING of the first hint received (ρ = 0.267; S = 20322; p = 0.049) and the second hint received (ρ = 0.409; S = 5838; p = 0.010). Additionally, hint QUALITYRATINGS had a significant positive Spearman correlation with the percentage of future hints that were followed for second hints (ρ = 0.413; S = 2384; p = 0.026) but not first hints (ρ = 0.201; S = 7892; p = 0.219). Together, these findings strongly suggest that better initial hints encourage students to make more use of
Table 5.2 Mean attribute ratings (with standard deviation) for first and second hints of attempts with each hint usage label. All hint ratings are summarized at the bottom.

<table>
<thead>
<tr>
<th>Hint</th>
<th>Label</th>
<th>N</th>
<th>Relevance</th>
<th>Progress</th>
<th>Interpretability</th>
<th>QUALITY RATING</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Low</td>
<td>23</td>
<td>1.96 (0.88)</td>
<td>2.13 (0.92)</td>
<td>1.83 (0.72)</td>
<td>5.91 (2.13)</td>
</tr>
<tr>
<td></td>
<td>Med.</td>
<td>15</td>
<td>2.27 (0.88)</td>
<td>2.47 (0.74)</td>
<td>1.80 (0.68)</td>
<td>6.53 (2.13)</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>17</td>
<td>2.35 (0.70)</td>
<td>2.65 (0.70)</td>
<td>2.12 (0.86)</td>
<td>7.12 (1.90)</td>
</tr>
<tr>
<td>2</td>
<td>Low</td>
<td>7</td>
<td>1.57 (0.79)</td>
<td>1.86 (0.90)</td>
<td>1.57 (0.79)</td>
<td>5.00 (2.38)</td>
</tr>
<tr>
<td></td>
<td>Med.</td>
<td>15</td>
<td>2.13 (0.83)</td>
<td>2.53 (0.74)</td>
<td>2.00 (0.76)</td>
<td>6.67 (2.06)</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>17</td>
<td>2.53 (0.72)</td>
<td>2.71 (0.69)</td>
<td>2.29 (0.77)</td>
<td>7.53 (1.97)</td>
</tr>
<tr>
<td>1, 2</td>
<td>All</td>
<td>94</td>
<td>2.18 (0.83)</td>
<td>2.43 (0.81)</td>
<td>1.97 (0.77)</td>
<td>6.57 (2.14)</td>
</tr>
</tbody>
</table>

hints in the future. Another way to interpret this result is that students can somewhat easily be deterred from using hints if they encounter one that does not seem useful. Looking only at the first 2 hints in an attempt, students stop asking for hints after receiving a hint that had at the least median QUALITY RATING only 18.2% of the time (10/55), while they stop 41.0% of the time after receiving a sub-median quality hint (16/39), over twice as often.

5.4 Discussion

RQ4.1 How frequently did students use and follow hints in iSnap? Consistent with prior results [AK00], we found that help use in iSnap was quite low overall. Few assignment attempts included help requests (17-30% across assignments), and many attempts which had help requests only had one or two hints (38-64%). A quarter of student homework attempts missed multiple objectives, but few of these had meaningful hint usage, suggesting help avoidance. We could help to address this with a more detailed introduction on iSnap and how to use it effectively, or by offering proactive hints. Less than half of hints were followed (41.3%), calling into question the hints’ overall utility. However, this result was expected to some extent due to iSnap’s interface, which provides students with buttons for all available hints at once, only some of which are likely to address the student’s original reason for requesting help.

RQ4.2 How did hint use relate to student performance on assignments? We saw that students who followed at least one hint on homeworks performed at least as well as those who did not and rarely performed poorly. This supports the idea that help in iSnap can improve student performance, at least on some assignments. However, also consistent with the literature, we saw evidence of potential help abuse, with some students requesting over a dozen hints on one assignment. Since
this behavior is negatively associated with learning [Ale06], improved performance may not lead to improved learning for these students.

**RQ4.3 How did a given hint’s quality affect whether it was followed?** We confirmed our assumption that hint quality does vary considerably among our data-driven hints. We also found empirical evidence to support the intuitive notion that students are more likely to follow higher-quality hints. We saw that the bar is quite high, with followed first and second hints having medium QUALITYRATINGS of 8 and 8.5 out of 9. This suggests that anything less than a great hint will go unused more often than not. That has serious implications for the design of data-driven hints, emphasizing the need for a vetting mechanism to ensure lower quality hints are not shown, perhaps using data on which hints have gone unfollowed in the past. However, just as there are many factors that influence a student’s willingness to seek help in an ITS [Ale03], we should also consider contextual factors besides hint quality that might encourage or discourage students from following requested hints, such as student affect.

**RQ4.4 How did the quality of students’ first hints affect their future help use?** The impact of hint quality appears to continue on to later hints, with a positive correlation between the QUALITYRATINGS of the first two hints a student receives and the student’s later level of hint usage. While we cannot speak directly to the mechanism at work here, it seems likely that a student’s interactions with hints establish the student’s level of trust in the system, which impacts future hint use. The first few interactions with help may be especially important, though we did not investigate later hints to verify this. Interestingly, we see that higher initial hint quality is predictive of High hint usage, which could indicate help abuse. One might hypothesize that students who abuse help are pre-disposed to do so, due to individual factors like prior knowledge and motivation; however, our results suggest that trust in the ITS could also play a role. Due to these findings, iSnap v2 was updated to include a warning for students using too many hints in a row (as discussed in Section 3.7 of Chapter 3).

### 5.5 Conclusion

In this chapter, we have presented evidence that, in practice, data-driven hints in an ITS for programming vary in quality, which has both an immediate impact on whether students follow hints, and a long-term impact on how many hints they request. This work is novel in that it directly measures hint quality using expert ratings and investigates the impact of hint quality on students. One hypothesis that would explain this impact is that hint quality affects a student’s trust in the ITS, and trust affects future help use. This has important implications for the design of ITSs, whether or not they rely on data, since all intelligent systems operate under some degree of uncertainty. Designers should carefully craft ITS help to avoid negative interactions with students, which might mean choosing not to offer help in uncertain situations.

This work has several limitations. Our dataset included only one programming course and four
assignments, and we cannot claim that the results will generalize to other domains. iSnap differs from traditional ITSs (e.g. Cognitive Tutors) in its lack of a student model and the open-ended nature of the task it supports. It also offers multiple hints at once, giving students some control over the hints they receive, so it is possible that student-specific factors played a confounding role in our analysis. Due to the small number of attempts with hint requests, we analyzed hints from different assignments together, but our results in Section 5.3.2 suggest that the assignment may impact how hints affect the student. Additionally, our hint ratings were made by experts, not students, and may be subject to the “expert blindspot.” Our analysis was exploratory, and our primary goal was to generate hypotheses, not conclusions. We therefore chose not to correct for multiple significance tests, but this means our results should be interpreted cautiously until they can be confirmed.

While the data analysis we present here is an important first step, more detailed methods are needed to understand the mechanisms by which hint quality impacts student help-seeking behavior. Chapter 7 addresses this through an analysis of interviews with students using programming help. While Aleven and colleagues’ model of ideal help-seeking behavior in a Cognitive Tutor is useful [Ale06], Chapter 7 also works towards developing a model of how students seek help in practice, to design systems which can leverage these behaviors for better learning.
CHAPTER 6

STUDY 2: THE IMPACT OF DATA QUANTITY AND SOURCE ON DATA-DRIVEN HINT QUALITY FOR PROGRAMMING

This chapter was adapted from a manuscript entitled:

The original text has been modified as follows: The Related Work section has been removed and integrated into Sections 2.3.1 and 2.3.3 of Chapter 2. Some references and the research questions have been updated to refer to relevant parts of this document. Note that this chapter analyzes data collected with both the original iSnap (CTD), and iSnap v2 (SourceCheck).
Abstract

In the domain of programming, intelligent tutoring systems increasingly employ data-driven methods to automate hint generation. Evaluations of these systems have largely focused on whether they can reliably provide hints for most students, and how much data is needed to do so, rather than how useful the resulting hints are to students. We present a method for evaluating the quality of data-driven hints and how their quality is impacted by the data used to generate them. Using two datasets, we investigate how the quantity of data and the source of data (whether it comes from students or experts) impact one hint generation algorithm. We find that with student training data, hint quality stops improving after 15-20 training solutions and can decrease with additional data. We also find that student data outperforms a single expert solution but that a comprehensive set of expert solutions generally performs best.

6.1 Introduction

Intelligent tutoring systems (ITSs) increasingly use student data to drive their decision making. Rather than relying on extensive knowledge engineering, authors can employ data-driven methods to automate the development of both “outer loop” [Van06] components of an ITS, such as constructing [Yud14] and improving [KS13] student models, and “inner loop” components, such as automatically generating hints [BS10] and worked examples [Mos15]. Authors of data-driven systems argue that these approaches avoid the need for experts to spend time constructing complex domain models [BS10; RK17] and can lead to additional insights that experts alone would not achieve [KS13].

However, empirical evaluations of the costs and benefits of data-driven approaches are still rare. This is especially true in the case of data-driven hint generation, where student data is used to automatically generate hints, rather than relying on expert models (e.g. model tracing [CA01] or constraint-based modeling [Mit98]). Many evaluations of data-driven hints have focused on whether a system can reliably provide hints to students [Per14; Wan17], as well as how much data is necessary to do so [BS10; Ped14; RK17], a challenge called the “cold start problem” [BS10]. However, a cold start analysis can only measure the availability of hints; it makes no claims about how useful the hints will be for students. This is particularly concerning in the domain of programming, given the results from Chapter 5, which suggest that data-driven hint quality varies considerably and that lower quality hints can deter students from seeking help when they need it.

In this chapter, we present a reframing of the cold start problem in which we evaluate the quality, rather than the availability, of data-driven hints for programming. We use this analysis to investigate RQ3: How can we evaluate and compare the quality of techniques for generating data-driven hints? Continuing the investigation of this RQ from Chapter 4, we focus on the following study-specific research questions:
RQ3.3 How does the quantity of available training data affect the quality of data-driven programming hints?

RQ3.4 How does data from students compare to expert-authored data for generating high-quality programming hints?

We address these questions using datasets from two different programming environments that offer data-driven hints. We find that with student training data, hint quality stops improving after adding 15-20 training solutions and that additional data may harm hint quality. We also find that student data can outperform a single expert solution, but that a comprehensive set of expert solutions generally performs best. Our results suggest that some data-driven hint algorithms can be oversensitive to individual training solutions. We show that this can be reduced by weighting hints using multiple training solutions, which also significantly improves hint quality. The primary contributions of this work are a new procedure for assessing the quality of data-driven hints and results from the first investigation of the impact of data quantity and source on data-driven hint quality.

6.2 Method

We used the SourceCheck data-driven hint generation algorithm (see Chapter 4) to investigate our research questions. Like many hint generation algorithms in the domain of programming (e.g. [RK17; Wan17; ZR15]), SourceCheck generates hints for a hint-requesting student in two phases. First, it searches a training dataset of correct solutions for the one that best matches the hint-requesting student’s code, using a code-specific distance metric. It then constructs a set of edits to transform the student’s code into the matching solution and suggests these edits as hints.

6.2.1 Data

We analyzed datasets from two programming environments that offer on-demand, data-driven hints: iSnap, a block-based programming environment (see Chapter 3) and ITAP [RK17], an intelligent tutoring system (ITS) for Python programming. Both datasets consist of log data collected from students as they completed multiple programming problems, including complete traces of their code and records of hints they requested. The iSnap dataset was collected from an introductory programming course for non-majors during the Fall 2016 and Spring 2017 semesters, with 120 total students completing up to 6 programming problems. In the Fall 2016 semester, students used the original iSnap and in Fall 2017 they used iSnap v2. The ITAP dataset was collected from two introductory programming courses in Spring 2016, with 89 total students completing up to 40 Python problems. See [Riv16] for a more detailed description of the ITAP data collection. Both datasets are available from the PSLC Datashop (pslcdatashop.org) [Koe10].
Our evaluation of data-driven hint quality required two sets of data: a set of hint requests for which to generate hints, and a set of training data used to generate those hints. From the iSnap dataset, we randomly sampled one hint request per problem from each student who used hints. This ensured that no student was overrepresented in the set of hint requests. From the ITAP dataset, we randomly sampled up to two unique hint requests per problem from each student who used hints, since there were fewer students who requested hints than in the iSnap dataset. We only sampled hint requests where the student’s Python code could be parsed, since this is required by SourceCheck. We also extracted a set of training data from each dataset consisting of the traces of each student who submitted a correct solution and did not request hints.

We selected a subset of problems from each dataset to analyze, since our method for evaluating hint quality (explained in Section 6.2.2) involves time-intensive hint generation by expert tutors. From the iSnap dataset, we selected two representative problems, GuessingGame and Squiral, which were also used in the evaluation of SourceCheck presented in Chapter 4. The two problems had 23 and 24 hint requests respectively (47 total). Common solutions to these problems are approximately 13 and 10 lines of code, respectively, and require loops, conditionals, variables and procedure definitions. From the ITAP dataset, we selected all 5 problems that had at least 7 associated hint requests, for a total of 51 hint requests (7-14 per problem). These simpler problems had common solutions with 2-5 lines of code, which included variables, API calls, arithmetic operators and, for one problem, loops. An important difference between the datasets is that the iSnap problems are considerably longer and more open-ended, while the ITAP problems often involve single-line functions, evaluated with test cases. Descriptions of both sets of problems can be found in Appendix A.

6.2.2 Assessing Hint Quality

Both of our RQs investigate data-driven hint quality, so we developed a method to measure it. As in Chapter 4, we used a group of experts to generate a set of “gold standard” hints for each hint request, and we labeled data-driven hints as high-quality if they matched these gold standard hints. First, three tutors independently reviewed each hint request, including the history of the student’s code leading up to the hint request. Each tutor then generated a set of all hints they considered to be valid, useful, and not confusing. Each hint was represented as one or more edits to the student’s code, making these hints comparable to the edit-based hints offered by most data-driven algorithms. Tutors were instructed to keep their hints to one edit (e.g. one insertion or deletion) unless multiple edits were needed to avoid ambiguity or confusion. Hints were designed to be independently useful, with the understanding that the student might receive any one hint, not the whole set.

Next, each tutor independently reviewed the hints generated by the other two tutors and marked each hint as valid or invalid by the same criteria used to generate hints. We included in our gold standard set any hint which at least two out of three tutors considered to be valid. Our goal was
not to determine a definitive, objective set of correct hints but rather to identify a large number of hints that a reasonable human tutor might generate. Requiring that two tutors agreed on each hint provided a higher quality standard than is used in most classrooms, while allowing for differences of opinion among tutors. This produced between 1 and 11 gold standard hints per hint request for the iSnap dataset (Med = 5) and between 1 and 5 for the ITAP dataset (Med = 2)\(^1\).

We can use this set of gold standard hints to automatically assign a \textsc{QualityScore} to a hint generation algorithm for a set of hint requests. For each hint request, we first generate a set of candidate hints, \(H\), using the algorithm. Since some algorithms (including SourceCheck) generate multiple hints for a given request, we require that the algorithm also assign a confidence weight to each hint, with the weights summing to 1. We then mark each candidate hint \(h \in H\) as valid if it matches one of the gold standard hints, after standardizing the names of any newly created variables, methods and string literals. We also detect partial matches where a candidate hint consists of a subset of the edits suggested by the tutor. The final \textsc{QualityScore} is the sum of the weights of all valid hints in \(H\). The \textsc{QualityScore} ranges from 0 (no valid hints) to 1 (all valid hints) and can be calculated to include partial matches or only full matches. We average this score over all hint requests to produce a final \textsc{QualityScore} for the algorithm. Note that this \textsc{QualityScore} should not be confused with the \textsc{QualityRating} presented in Chapter 5. While a hint’s \textsc{QualityRating} is determined by experts directly rating the hint, the \textsc{QualityScore} is determined by comparing a hint to other hints that experts have generated. This process is similar to the evaluation presented in Chapter 4, but rather than using the method to measure the similarity between human-authored and algorithmically generated hints, we use it here to create a single \textsc{QualityScore}.

In our analysis, we compared two different approaches for assigning weights to the set of hints generated by SourceCheck, \(H\), for a given hint request. The first, uniform weighting, simply assigns a weight of \(1/|H|\) to each hint. The second, voting-based weighting, uses multiple solutions in the training dataset to assign weights. Recall that SourceCheck uses a single, closest matching target solution in the training dataset to generate hints (a 1-nearest-neighbor approach). With voting-based weighting, hints are also generated using the top-\(k\) closest solutions in the training dataset. The weight of each hint \(h \in H\) is the percentage of these top-\(k\) solutions which, if used for hint generation, would also have generated \(h\). The weights are then normalized to sum to 1. In our analysis we chose \(k = 10\), though other values of \(k = 5\)…\(15\) produced similar outcomes. Note that the weighting approach does not change which hints are generated, only each hint’s weight in determining the \textsc{QualityScore}.

\(^1\)Full datasets and hint ratings are available at http://go.ncsu.edu/hint-quality-data.
6.2.3 Cold Start Procedure

To investigate RQ3.3, we developed a cold start procedure that measures the relationship between the quantity of available data and the resulting hint QualityScore. Unlike other cold start procedures (e.g. [BS10; Fos15; RK17]), our evaluation considers only actual student hint requests, rather than all states observed in historical data. For each dataset, we start with the full set of training traces, $T$, the set of hint requests, $R$, and a sample set of training traces, $S = \emptyset$. We repeatedly select a random trace in $T$, remove it and add it to $S$. After each addition, we use the SourceCheck algorithm to generate a set of hints for each hint request in $R$ and calculate a QualityScore for these hints. We repeat this until $T$ is empty. The end result is a QualityScore for each $i \in 1..|T|$, where $i$ is the number of traces used to generate hints. We repeated this whole procedure for the iSnap and ITAP datasets 200 times and averaged the results.

6.2.4 Comparing with Expert-authored Data

In order to investigate RQ3.4, comparing data from students and expert-authored data, we created two expert-authored training datasets to use as baselines. The first baseline dataset, ONEEXPERT, consists of a single solution, authored by an expert to reflect a straightforward approach to solving each problem. This is a minimal baseline, since almost any problem will already have at least one teacher-authored solution available. The second baseline dataset, ALLEXPERT, consists of all solutions considered by an expert to be useful for hint generation. To assign a QualityScore to these baselines, we still generate hints with the SourceCheck algorithm, but we provide it the expert training dataset.

To allow an expert author to generate a comprehensive set of correct solutions, we developed a simple solution templating language. Using this language, an author can write solution code that includes “branches,” indicating where there are multiple ways of writing an acceptable solution. These branches may represent different high-level programming strategies, or simply different ways of expressing the same idea programmatically. Branches can be nested to produce a wide variety of solutions. Additionally, authors can mark parts of the solution code as order-invariant and add wildcards that can match any code element. One researcher, familiar with the problems, generated the expert solutions for both baselines, using the set of students solutions as a reference. For the larger iSnap problems, ALLEXPERT solution sets took 60-90 minutes to author and test, and for the smaller ITAP problems, they took 30-60 minutes to author and test, making the process time consuming but feasible for classroom use. For GuessingGame and Squiral from the iSnap dataset, the ALLEXPERT solution sets consisted of 960 and 1,152 unique solutions, respectively. For the ITAP dataset, the expert author generated only 2 solutions for most problems, though one problem

\[\text{Over 200 trials, the standard error of the averaged QualityScores was always less than 0.01, and averaged less than 0.0025 across values of } i.\]
Figure 6.1 Cold start curves for the iSnap dataset (black) with uniform hint weights (solid) and voting-based weights (dotted). AllExpert (green) and OneExpert (blue) baselines are shown for comparison.

(KthDigit) had 12.

6.3 Results

Figure 6.1 shows cold start curves for the iSnap dataset, plotting how SourceCheck’s QUALITYSCORE (averaged over all trials) changes for both weighting approaches as the number of solutions in the training dataset increases. In many ways, the curves resemble a traditional cold start curve, with the QUALITYSCORE increasing rapidly as the first 10-15 solutions are added to the training data and then leveling off. Comparing the uniform and voting-based weighting approaches, we can see that the voting-based approach consistently performs better, achieving a 19.1% and 15.4% higher final QUALITYSCORE on the GuessingGame and Squiral problems, respectively. A Wilcoxon signed-rank test showed that this difference in the final QUALITYSCORE was significant for the sample of 47 iSnap hint requests ($V = 331$; $p = 0.015$; Cohen’s $d = 0.143$).

Considering RQ3.3, we see that unlike traditional cold start curves, which measure hint coverage and are guaranteed to increase monotonically with more data, our QUALITYSCORE curves show clear fluctuations as more training solutions are added, sometimes decreasing in QUALITYSCORE. Here too we see a difference between weighting approaches. The uniform weighting curves have a final QUALITYSCORE 8.4% and 8.3% lower than their peak score on GuessingGame and Squiral, respectively, while the voting-based weighting curves lose only 3.4% and 4.8% of their peak QUALITYSCORE, respectively. Recall that Figure 6.1 shows QUALITYSCORE averaged over all hint requests for a given problem. To further investigate why additional data may decrease hint QUALITYSCORE in the iSnap dataset, we calculated cold start curves for each of the 47 iSnap hint requests individually, and we
found a wide variety of curve shapes. For voting-based weighting, 39.1% and 33.3% of hint requests on GuessingGame and Squirrel showed a negative correlation between quantity of data and hint \textsc{QualityScore}, while 47.8% and 45.8% showed a positive correlation. This trend was similar for uniform weighting. This suggests that the fluctuations at the end of the cold start curves are due to competing positive and negative effects of data quantity on different hint requests.

To investigate RQ3.4, Figure 6.1 also shows lines indicating the \textsc{OneExpert} and \textsc{AllExpert} baseline \textsc{QualityScores}\(^3\). Since these values are calculated using a set of expert solutions, not student data, they have a constant value. Student data with uniform weighting fails to achieve the quality of the \textsc{OneExpert} baseline. However, with voting-based weighting, student data outperforms \textsc{OneExpert} with only 7 training solutions for both problems, achieving a peak \textsc{QualityScore} 9.8% and 7.5% higher than \textsc{OneExpert} on GuessingGame and Squirrel. Even with voting-based weighting, student data falls short of the \textsc{AllExpert} baseline, with the peak student \textsc{QualityScore} 73.3% and 94.3% as high as the \textsc{AllExpert} baseline. However, a Wilcoxon signed-rank test showed that this difference between the final student data \textsc{QualityScore} and the \textsc{AllExpert} baseline was not significant for the sample of 47 iSnap hint requests ($V = 181; p = 0.077$; Cohen's $d = 0.221$).

Figure 6.2 shows cold start curves for the ITAP dataset. Note that unlike in the iSnap dataset, \textsc{QualityScores} for ITAP curves were calculated including \textit{partial} matches (see Section 6.2.2), since Python ASTs are more complex, and many of \textsc{SourceCheck}'s hints matched only part of the gold standard hints. The IsPunctuation, KthDigit, and OneToN problems have a similar curve shape to the iSnap problems; however, rather than fluctuating, they increase monotonically in \textsc{QualityScore} with additional data. They also show almost no difference between uniform and voting-based weighting. Student data easily outperforms the \textsc{OneExpert} baseline and, in the IsPunctuation and OneToN problems, matches the \textsc{AllExpert} baseline. Note that these problems have much smaller training datasets, and it is unclear how the curves would continue with additional training data.

The other two curves, for HelloWorld and FirstAndLast, have unique shapes. For both problems, additional data has a clear negative impact on hint quality. Additionally, the \textsc{OneExpert} baseline has equal or better performance compared to the \textsc{AllExpert} baseline. HelloWorld and FirstAndLast are the simplest problems we investigated, each with a common, one-line solution that comprised 47.1% and 93.5% of the training dataset, respectively. This solution was also used in the \textsc{OneExpert} baseline. The curves suggest that the presence of other solutions in the training dataset lowered hint \textsc{QualityScore}. For FirstAndLast, the second most common solution comprised 35.3% of the training dataset. It used a more advanced Python feature (using $s[-1]$ to get the last character in a string), which the human tutors never suggested to struggling students, preferring a more straightforward approach ($s[\text{len}(s) - 1]$). SourceCheck often selected the more advanced approach as the target solution, since it is shorter, which accounts for the lower \textsc{QualityScore} for student data on this

\(^3\)Note that because the \textsc{OneExpert} baseline uses only one training solution, both weighting approaches produce the same results. For simplicity, Figure 6.1 shows only voting-based weighting for the \textsc{AllExpert} baseline.
problem. It is also important to note that QUALITYSCORES for these two problems were based on the fewest number of hint requests (7 each), which may not be fully representative of all likely hint requests.

6.4 Discussion

RQ3.3 How does the quantity of available training data affect the quality of data-driven programming hints? Our results suggest that increased training data has a positive impact on hint quality up to a point. For our datasets, hint quality stops increasing after 15-20 training solutions. However, for the more complex problems in the iSnap dataset, additional data had either no positive effect, or even a slight negative effect, on hint quality. While this does validate the claim of many data-driven
algorithms that little student data is needed to generate hints [BS10; Fos15], it is concerning that additional data fails to improve SourceCheck’s effectiveness. A similar problem was also noted by Rivers et al. [RK17] on some problems, where ITAP’s hint generation performance decreased with additional data. We could interpret the cold start analysis presented here as a method to measure a data-driven algorithm’s ability to effectively make use of all of its data. Ideally, a data-driven algorithm should have monotonically increasing hint quality as it is given additional training data, selecting the useful solutions and ignoring unhelpful ones. By this interpretation, the SourceCheck algorithm has mixed success, especially on the iSnap dataset.

In considering why additional data may be harmful, recall that this evaluation relied on the SourceCheck algorithm to generate all the hints we evaluated. Like many data-driven hint generation algorithms (e.g. [RK17; Wan17; ZR15]), SourceCheck generates hints using the single best-matching target solution from the training dataset (a “1-nearest neighbor” approach). This makes it sensitive to additional training data. Our training data consists of all correct student solutions, and some of these will contain unnecessary code, unique design choices, or advanced features (e.g. $s[-1]$ in FirstAndLast) that make them problematic for hint generation. This is especially true for the more open-ended problems found in the iSnap dataset. As more of these problematic solutions are added to the training dataset, it becomes increasingly likely that one of them will be the closest match for the hint-requesting student (e.g. if the hint-requesting student added similar unnecessary code). Since only the closest match in the training dataset is used for hint generation, a single one of these problematic solutions can eclipse others that are more useful.

Our comparison of uniform and voting-based hint weighting lends some support to the hypothesis that this “1-nearest-neighbor” approach to hint generation is overly sensitive to additional data. When SourceCheck uses the top 10 best matches in the training dataset to weight hints, its final QUALITYSCORE improves significantly and less quality is lost from additional training data. However, this weighting does not change the hints themselves, which may still suffer from a bad match. Another way to reduce this sensitivity to additional data is the approach of Gross et al. [Gro14a], which selects a matching cluster of solutions, rather than a single target, and uses an exemplar solution from that cluster for hint generation. Rivers and Koedinger [RK17] address this in part by discovering new target solutions through path construction, which better match the hint-requesting student’s code. We could also attempt to filter training solutions (manually or automatically) to retain only those useful for hint generation. Importantly, both the decrease in QUALITYSCORE with additional training data and the difference in QUALITYSCORE between weighting approaches appeared only in the iSnap dataset. This may be because the iSnap dataset featured larger and more complex problems than the ITAP dataset, but it is also possible that these trends result from more specific features of the iSnap dataset itself.

RQ3.4 How does data from students compare to expert-authored data for generating high-quality programming hints? The use of student data for hint generation makes the implicit assumption that
it will yield better hints than using a single expert solution, and our results suggest that this is not always the case. In the iSnap dataset, student data failed to outperform the ONEEXPERT baseline when using uniform hint weighting. However, we also showed that this can be addressed with voting-based weighting. On the simpler ITAP dataset, student data seems to perform better, outperforming ONEEXPERT with only 1-3 training solutions on 3 problems, regardless of the hint weighting. However, our results also show that for the simplest problems (HelloWorld and FirstAndLast), a single expert solution may be all the data that is required.

The ONEEXPERT baseline represents the simplest use of expert data. The more robust ALLEXPERT baseline outperformed student data on the three problems where more than 2 expert solutions were generated as training data (GuessingGame, Squirrel, KthDigit), as well as on FirstAndLast. While this difference was not significant on the iSnap dataset, this may be due to our relatively small sample size (47 hint requests). On the remaining problems, student data has similar performance to the ALLEXPERT baseline. These results suggest that for all but the simplest problems, a comprehensive set of expert solutions is likely to be more useful for hint generation than student data. This is not surprising, since our expert solutions were authored by a researcher familiar with the problems (as most instructors would be), using student solutions as a reference, so in many ways they represent an ideal training dataset. It would also be possible to author expert solutions without the benefit of referencing student solutions, but this may yield lower-quality hints.

The template-based data generation technique used in the ALLEXPERT baseline is not the focus of this chapter, but we do note some advantages to this approach. In addition to outperforming student data in terms of QUALITYSCORE, this approach solves the cold start problem entirely. Also, if a problem changes from one semester to the next (e.g. adding an additional objective), it is easy to modify the corresponding template, while any student solutions would be rendered useless. The results from Chapter 5 suggest that even a few low-quality hints may deter students from using a help system, so the consistency of expert-authored data is desirable. However, like expert models, the templates are time intensive to produce, taking 30 to 90 minutes per problem and likely longer for larger or more complex problems. This process may therefore not scale well to larger problem sets.

### 6.4.1 Considerations and Limitations

Automatically assessing the quality of hints is a difficult and inherently subjective task. Our method for generating gold standard tutor hints strikes a balance between including many hints (generated by three separate tutors) and ensuring hint quality (all hints were “seconded” by another tutor). However, our results may have been different if we had used more than 3 tutors or a higher standard of agreement (consensus among all tutors). The process of matching an automatically-generated hint to a gold standard hint is also imperfect, and it is likely that some hints were marked invalid.
Despite being similar to a gold standard hint. For all of these reasons, the QUALITYSCORES reported here may seem low. However, raw QUALITYSCORES are difficult to interpret, so we use them primarily for comparing hint generation approaches.

An important limitation of this work is that the QUALITYSCORE metric was calculated based on a moderate number of hint requests for each problem (23-24 for iSnap; 7-14 for ITAP). The QUALITYSCORE metric is therefore only as accurate as these hint requests are representative of all hint requests. The tutors generating hints for these requests noted that students’ code often contained a range of strange misconceptions and creative mistakes and often did not resemble other students’ solutions. However, we see this as an important feature of our data and analysis. The set of code states where students request hints is not representative of students’ code generally, since it reflects students’ need to request outside help. Hint requests are full of strange design choices and incorrect code, and this is exactly what a hint generation algorithm should be designed to address and overcome. A related limitation of this work is that these hint-requesting students were not included in the training dataset, even if they ended up with a correct solution, which diminished the diversity of our training datasets. Lastly, most problems in the ITAP dataset had a limited number of training solutions (7 to 22), and it is unclear how the cold start curves would have continued with additional training data.

6.5 Conclusion

In this chapter, we presented a detailed method for evaluating the quality of data-driven hints, and we applied that method to investigate how hint quality is impacted by the quantity and source of data used to generate hints. Our results highlighted ways that student data may detract from hint quality, especially when hint generation uses a single, best-matching training solution. We introduced a voting-based hint weighting approach that reduces this reliance and significantly improves hint quality. With this weighting, we showed that student data outperforms a single expert-authored solution for hint generation but falls short of a comprehensive set of expert solutions. Finally, we presented a new, template-based method for generating this set of expert solutions. We compared our results across 2 distinct datasets that used different programming languages – to our knowledge, the first evaluation of data-driven hints to do so.

While our quality evaluation procedure comes with limitations, we argue that research on data-driven hint generation should hold itself to the more rigorous standard of evaluation presented here, rather than relying on simpler availability evaluations. We have suggested that part of that standard should be an algorithm’s ability to effectively leverage additional training data, which can easily be assessed using the cold start curves presented here. In future work, we hope to use our method to benchmark and compare hint generation algorithms on common datasets to better understand their strengths and weaknesses. We also hope to further explore how to better leverage multiple
training solutions, rather than a single best match, to generate higher quality hints.
CHAPTER 7

STUDY 3: FACTORS INFLUENCING STUDENTS’ HELP-SEEKING BEHAVIOR WHILE PROGRAMMING

This chapter was adapted from:

The original text has been modified as follows: The Related Work section has been removed and integrated into Section 2.4 of Chapter 2, and the section describing iSnap has been removed, since this is covered in Chapter 3. Some references and the research questions have been updated to refer to relevant parts of this document. Note that this chapter presents an evaluation of the original iSnap (using the CTD algorithm).

Abstract

When novice students encounter difficulty when learning to program, some can seek help from instructors or teaching assistants. This one-on-one tutoring is highly effective at fostering learning,
but busy instructors and large class sizes can make expert help a scarce resource. Increasingly, programming environments attempt to imitate this human support by providing students with hints and feedback. In order to design effective, computer-based help, it is important to understand how and why students seek and avoid help when programming, and how this process differs when the help is provided by a human or a computer. We explore these questions through a qualitative analysis of 15 students’ interviews, in which they reflect on solving two programming problems with human and computer help. We discuss implications for help design and present hypotheses on students' help-seeking behavior.

7.1 Introduction

Programming is a challenging skill to learn, particularly for novices. Getting “stuck” on an assignment can place an emotional toll on students [KS10b] and may result in the student giving up. Ideally in these situations, a student will be able to seek help from a professor or teaching assistant and receive one-on-one, expert tutoring, arguably the most effective form of instruction [Blo84]. However, experts – or even knowledgeable peers – are not always available or accessible, especially in overcrowded classrooms, and they may be absent altogether in informal learning settings. To address this, researchers and educators are increasingly augmenting programming environments with help features to aid students who are stuck when no human is available.

There are many forms of embedded help found in programming environments, including enhanced error messages [Den14], program visualization tools [Sor13] and debugging aides [KM04]. Particularly promising are Intelligent Tutoring Systems (ITSs), which attempt to play the role of the human tutor [Van06]. In the domain of computing, ITSs can offer intelligent, contextual help through hints (e.g. [Pri17d; RK17]), feedback (e.g. [Ger16]) and curated examples (e.g. [WB01]). Working with an ITS can improve computing students’ performance and decrease their programming time on problems, both when help is available and afterwards, outside of the tutor [CA01], leading to claims that ITSs can match the learning gains of human tutors [Cor01].

However, researchers have noted that students’ use of help features in ITSs is far from ideal, with students avoiding help when they need it, or abusing help to expediently solve the problem [AK00; Ale03], behaviors negatively correlated with learning [Rol06]. ITSs for learning programming are not immune to these problems, as shown in Chapters 3 and 5, which presented evidence of help avoidance and abuse. This leads us to ask, what do we know about how and why students seek programming help from computer tutors – or from human tutors? With a few exceptions [KF13; Vae14], little work has investigated help-seeking in the domain of computing. Existing models of help-seeking focus on general classroom learning [Nel81; New94] or on desired help-seeking behavior in an ITS [Ale06]. However, educational psychology results from other domains do not always apply to computing [Mor15b; Mor15c], and we are interested in how and why students
actually seek help, not just how they should. In this work, we investigate factors that influence novice programmers’ help-seeking behavior and how this differs with human and computer tutors. We present a qualitative analysis of 15 students’ interviews, in which they reflect on solving two programming problems: one with a human tutor and one with intelligent, computer-based help. We discuss design implications and hypotheses that arise from these results.

7.2 Method

This work was motivated by RQ4: What factors impact students’ willingness to seek and use programming help? Continuing the investigation of this RQ from Chapter 5, we focus on the following study-specific research questions:

RQ4.5 Why do novices seek and avoid help when programming?

RQ4.6 How is the process of help-seeking different when novices are working with human help and computer-based help?

We focus on novice programmers because that is the target population for many programming ITSs [Ger16; RK17], including iSnap. The analysis presented in this chapter is part of a larger study to investigate help-seeking and help use in computing, the procedures of which are explained below. This analysis represents an initial attempt at understanding our data and formulating hypotheses that lay the groundwork to answer these research questions.

7.2.1 Population

We recruited 15 undergraduate students at a large U.S. research university to participate in our study. We recruited students who had completed or enrolled in an introductory programming course (AP CS, Java, Matlab or Python), but not a more advanced course. We made short, in-class announcements about the study, and we accepted the first 15 participants to volunteer. To encourage less eager programmers to participate, we compensated participants with a $20 gift card. Our participants included 4 females and 11 males, with most students identifying as White (11) or Asian (2), one as Hispanic/Latino and one as Other. Participants’ majors were primarily CS (5) or other engineering fields (8), and 9 had taken or were enrolled in the introductory Java course for CS majors. None reported having used Snap! or Scratch more than a few times.

7.2.2 Procedure

Each participant completed the study individually in a single session. Participants completed two programming assignments, one with a human tutor available and one with iSnap help available,
and then participated in a semi-structured interview, which is the focus of our analysis. Participants each attended an individual research session with two researchers: an experimenter and a tutor. In this study, both tutors were female. The participant began by completing a short logic puzzle for 5 minutes to practice thinking aloud, with the tutor available for help if needed. The tutor then led the participant through a short tutorial on the core Snap! programming concepts the participant would need for approximately 20 minutes. To reduce bias, no indication was given whether the research team was involved with the development of Snap!, and the whole system, including the iSnap help features, was simply referred to as Snap!. The participants then worked on two programming tasks, each for up to 30 minutes. Students were prompted to think aloud while working on these tasks. Before starting, the participant was asked to read the assignment instructions out loud and to ask the tutor any questions they had just on the instructions.¹

For the first task, the participant was encouraged to ask the tutor for help if they needed it, or to work independently if not. For the second task, the tutor left the room, and the participant was similarly encouraged to ask iSnap for help if they needed it. Before starting, participants were shown a 3 minute video on the help features of iSnap. In both cases, if the student appeared to be struggling with the same issue for at least 45 seconds, the tutor (in the first task) or experimenter (in the second task) intervened by saying, “remember, you can ask me/snap for help.” Assignments were selected from the first two units of the Beauty and Joy of Computing curriculum (bjc.edc.org) [Gar15] to be appropriate for novice Snap! programmers. In the GuessingGame assignment, the program should greet the player by name and then repeatedly ask them to guess a stored random number, telling them if it is too high, too low or correct. In the BrickWall assignment, students create custom blocks (procedures) to draw a brick wall with an arbitrary number of alternating rows. Students were given detailed instructions on how to complete the assignments, given in Appendix C. Note that the GuessingGame assignment used here is the same assignment that has been referenced in other chapters of this document; however, the instructions students received were more detailed than the brief description given in Appendix A. The order of the two assignments was randomized, with 10 receiving GuessingGame first with human tutor help. Because iSnap was not designed to answer questions about the Snap! programming interface itself, we ordered human tutor help first for all participants so that they had an opportunity to ask the tutor any additional interface questions during their first assignment.

After completing both assignments, the experimenter conducted an interview for 10-15 minutes, and the tutor remained out of the room. The experimenter used a semi-structured interview protocol to investigate the student’s perceptions of the tutor’s help, iSnap’s help, comparisons between the two and any suggestions the participant had for iSnap. The questions explored why the student did or did not seek help, the quality of the help and whether the student trusted the help for both assignments. The experimenter also followed up on interesting points and inquired about specific

¹Full study materials available at: http://go.ncsu.edu/icer2017-materials
behaviors observed during the programming exercises using notes. The full set of interview questions is available in Appendix D. While future work will explore the various types of data collected (think aloud protocols, survey responses, finished programs), we focus in this work on the interview data, as it provides a good starting point for understanding participants’ experiences.

7.2.3 Qualitative Analysis

We investigated our research questions using a qualitative analysis, based in Grounded Theory (GT). GT is a set of techniques for building theory that is “discovered, developed and provisionally verified through systematic data collection and analysis” [SC90]. These techniques, discussed in the next section, were appropriate because of the qualitative nature of our data, the open-endedness of our research questions and the lack of existing, satisfying theory addressing help-seeking in programming. GT has also been applied in CS education research in a number of contexts [Fit05; KS10b; KS11; Lew11; Lew16]. We focused on the variants of GT detailed by Strauss and Corbin [SC90; CS08] and Charmaz [Cha08; CB12], and drew on the detailed work of Kinnunen and Simon [KS10a] to apply GT to CS education.

GT is more than just a set of analysis techniques; it is an iterative approach to “gathering, synthesizing, analyzing and conceptualizing qualitative data” [Cha08]. As in previous work [KS10a], we primarily applied these techniques to data that had already been collected, so we recognize that our work borrows from GT but does not embody it. The analysis we present is an initial attempt to understand our data and research questions. Due to the limitations of our sample size and methods, our goal is not to definitively answer these questions, but rather to describe and interpret the phenomena we observed, discuss how our results relate to existing theory, and generate hypotheses. This is an expected first step in qualitative research, which can take many iterations to generate a full theory.

7.2.3.1 Coding

We began by performing line-by-line, open coding [Cha08] on transcriptions of the interviews we collected. We started with two intentionally selected interviews, from P12 and P15, which the experimenters identified as containing rich content that directly addressed our research questions. The four researchers that conducted the study served as coders. They coded the interviews independently, with two coding each of the two chosen examples, generating a total of 191 independent codes. The researchers met to discuss these codes and collectively sorted them (written on cards) into conceptual groups. Each group of similar codes was refined and named to produce an initial set of 36 focused codes. Examples include “trust” (a participant’s trust, or lack of trust, in the tutor or provided help) and “independence” (a participant’s beliefs about independence and actions taken based on these beliefs).
One researcher compiled these focused codes into a codebook and wrote brief descriptions for the other researchers to reference. The codes were not tutor-specific (e.g. “trust in human” vs “trust in iSnap”) and could be applied in reference to either the human or computer tutor. The referent (“human” or “iSnap”) was noted when the code was applied to a segment. Three of the original researchers then performed open coding on the full set of 15 interviews, during which the codes were further refined for a final set of 43 focused codes. They coded a total of 713 segments (over 30,000 words), with an average 1.8 codes per segment. Since the primary purpose of coding was to help us organize and understand our data, not to quantify and reduce it, each interview had one primary coder, and we therefore did not calculate inter-rater reliability. Instead, as in previous qualitative studies, we worked to establish rigor by discussing and refining codes throughout the analysis [KS10a; Lew11].

Near the end of coding, the researchers again physically sorted the codes, this time into broader conceptual categories. Drawing on the paradigm model of GT [SC90], we identified a central phenomenon of interest, “Help-seeking Behavior,” and related other categories to that phenomenon as causal conditions, context, and consequences. For this chapter, we narrowed our focus to factors influencing help-seeking behavior (rather than the help-seeking itself or its outcomes), which included 3 of the 8 categories we identified. We compiled, summarized and discussed the interview segments coded with each code in these categories, which we present in the next section. We applied constant comparison [SC90] throughout the process, writing extensive memos that contrasted segments and explored relationships among the codes and the categories.

### 7.3 Results

Our results focus on three categories of codes and their relationship to students’ help-seeking behavior. We briefly explain these categories and then discuss the most prominent codes from each:

**Inputs** are static factors that exist outside of the problem-solving session, such as a student’s previous experiences, the student’s existing knowledge and beliefs, and attributes of the programming assignment. The tutor (or ITS designer) has no direct control over these factors, and they are unlikely to change in the course of problem solving. While they may not influence help-seeking behavior directly, Inputs set a baseline for the Student Mindset.

The **Student Mindset** is how a student interprets their relationship to the programming problem, the tutor and the help the tutor offers. Examples include perceptions of the tutor’s trustworthiness, the accessibility of help and how stuck the student is. Unlike Inputs, the mindset is dynamic and can change throughout problem solving, especially but not exclusively when the student seeks help. A student’s help-seeking behavior is a response to this mindset.

**Attributes of Help** are qualities of a tutor’s help which determine its effectiveness and perceived value, such as specificity, interpretability and utility. Unlike codes in the Student Mindset category,
these attributes can only be experienced by the student after seeking help from the tutor. They affect the outcome of receiving help, possibly leading to changes in the Student Mindset.

7.3.1 Inputs

7.3.1.1 Previous Experiences

Previous Experiences encompass students’ past encounters with human or computer help and how these experiences shape their behavior during problem solving. Participants related the human tutor to past experiences with TAs and professors. When asked what it was like working with the human tutor, one participant said it was “just like office hour[s]... I don’t know how to do my homework, and I ask the TA for help” [P2]. Another participant expressed how their previous experiences in the classroom made help-seeking unfamiliar: “I’m also just not used to asking for help in [programming] classes” [P15]. Participants discussed avoiding human help during the study due to negative past experiences receiving programming help: “I’ve definitely experienced before where I had my- the design of my program in mind for a project... and then someone would be like, “oh no do it this way”... And I’m not saying [the tutor] did this. I’m just saying this is my experience that made me quick to dismiss her at first.” [P11]. P11 rarely asked for help from the human tutor, even after explicit reminders. These previous experiences, appear to strongly influence students’ later help-seeking behaviors.

Participants attempted to relate iSnap to previous experiences with computer help, compilers and IDEs. One participant noted how iSnap defied those expectations for the better: “in Python or in Matlab you don't have little help buttons like that... I was expecting syntactic help... but [iSnap] gives you pretty exact advice... that was pretty cool ” [P15]. Another wished iSnap had behaved more aggressively like a compiler: “in eclipse or any other IDE [if] you enter a... statement where you... have an argument that isn't compatible it will immediately flag that as red” [P5]. Other participants explicitly linked previous experiences with computer help to their help-seeking behaviors. One participant’s confidence rejecting one of iSnap’s suggestions was “not based on... my experience in this [study] but rather what I felt in my experience prior” [P5]. Another participant never used iSnap’s help, and also noted “I don’t like using Java help” [P7]. We hypothesize that prior experiences played an important role in shaping participants’ expectations of the computer tutor, as discussed in Section 7.3.1.2.

7.3.1.2 Expectations of the Tutor

Expectations of the Tutor refers to a student’s preconceptions of how a human or computer will provide help and how these expectations are confirmed or defied. Students had generally high expectations of the human tutor’s knowledge and experience, assuming “she’s an expert” [P1] and
“she understands it” [P13]. These expectations were largely met during the study: “she had the answer right away” [P10].

In contrast, participants’ expectations for iSnap were quite low. Many noted the fact that the help came from a computer as a specific source of low expectation, saying “it’s just a computer” [P12], “listening to a computer... is hard” [P3], and more exact, perceptive help is “not really something you should expect from little computer help buttons” [P15]. For many participants, these low expectations produced a satisfactory experience, even for some who rarely used the help or described it as being generally unhelpful. iSnap was “good enough” [P1], and it “does everything well enough” [P4]. For others, however, these low expectations meant never using the help in the first place, as was the case for one who saw iSnap as helping with the “basics of programming” [P7]. As noted in Section 7.3.1.1, iSnap defied these low expectations in some cases.

### 7.3.1.3 Independence

Independence refers to students’ beliefs and preferences about working without help. When asked what discouraged them from asking for help, nearly all participants expressed some form of the sentiment that they wanted to “figure it out myself” [P12], often using very similar language. They cited the belief that independent work leads to self-improvement or learning: “it’s more beneficial for me to just figure it out on my own” [P11], or that asking for help lowers self-efficacy: “if they ask for help, they feel like they can’t do it themselves” [P13]. One was willing to “keep trying myself until I’m too frustrated to keep trying” [P10], and another expressed an unwillingness to ask for help on math errors, “something I should be responsible for” [P5]. Two students mentioned that programming in particular may lend itself to independent work because “you can just kinda puzzle your way through it and figure it out on your own” [P15]. Some of these responses may reflect a selection bias in our participants, who willingly participated in a programming study and may be more motivated learners.

While participants also cited a desire to “figure this out myself” [P13] as a reason not to use computer-based help, it may not trigger the same threat to independence that human help does: “I’d probably click on the little computer help things quicker than going to an actual tutor just because I’m not asking someone else for help [laughs] per se” [P6]. Another participant got deep into debugging a problem without the human tutor’s help, but said “maybe if help indicators had been turned on for that program I would have gone back” [P5]. However, independence may also lead students to invest less effort in understanding hints: “I don’t think I really gave it a good enough chance... I didn’t actually over-analyze the structure [of the hint]... because I wanted to figure it out myself” [P10].
7.3.2 Student Mindset

7.3.2.1 Trust

Trust refers to a student’s confidence in the tutor’s ability and the quality of their help. Participants clearly trusted the human tutor in our study, citing assumptions about the tutor’s experience (see Section 7.3.1.2), the fact that the tutor was “encouraging... friendly and approachable” [P11] and provided affective support, the perceived quality of the advice itself, which “made sense” [P10], and the perceptiveness of the tutor (see Section 7.3.2.4). Trust did not always lead to help-seeking: “I... trust that she has the answer, but I felt it did not really motivate me to ask [for help]” [P1], indicating that it may be a necessary but not sufficient condition for help-seeking. Two participants’ negative previous experiences with human tutors did lead to initial mistrust and few help requests (see Section 7.3.1.1), but when asked directly the same students still described the study’s tutor as “definitely” [P11] trustworthy, again indicating that trust alone does not lead to help-seeking.

Participants generally expressed Trust in iSnap, though not as unanimously. When asked whether they trusted iSnap’s help, one participant expressed wariness because a computer could not adapt to the “many different ways that you can go about [solving the problem]... [iSnap] can’t know everything that I’ve done” [P12]. One trusted iSnap, but “a little less than an actual tutor” [P6]. Others expressed confidence that iSnap had more experience than they did with Snap! programming “because its... Snap itself that gives me advice” [P8]. Another found the directness of the advice trustworthy: “of course I trust it because... it just tells you... what needs to be done” [P7]. Trust may be less important if students feel confident evaluating help quality: “if you’re a programmer you should probably know... what advice is good and what advice is bad” [P15].

There was a small breach of procedure for P1, in which the tutor remained in the room during the second programming exercise. After being prompted to use iSnap’s help, P1 asked the tutor how the help was generated, and she explained. This changed P1’s trust of iSnap: “I only trusted... it after I asked [the tutor]... where is it getting this information and she said it’s based on previous students” [P1]. This may suggest that the human tutor “lent” iSnap some of her trustworthiness by explaining it, or that simply understanding the help mechanism makes it more trustworthy.

7.3.2.2 Stuck

A student is Stuck when they are no longer making progress on the problem. Many students noted this as a primary motivator for seeking help. This was true for both human help: “whenever I got stuck and like literally didn’t know what the next step was... then I was like, okay I need to ask for help” [P15], as well as computer-based help: “I was motivated to click on the hint bubbles when I couldn’t figure it out on my own” [P11]. For iSnap in particular, participants noted that the presence of the help was reassuring when stuck: “it was kind of a good feeling that you knew you had that little
bit that could help you if you were stuck” [P12]. Some participants also expressed reluctance to ask for help, even when stuck, and in retrospect recognized this as a harmful behavior that “comes back and bites me because... I really needed to ask a question, but I didn’t” [P13]. Participants described Stuck as a state one enters for a duration with degrees of intensity, rather than a single incident: “sitting there thinking too long” [P13] or “just sitting there stuck” [P15]. When this reaches a threshold, it can prompt help-seeking: “then I was like okay, I need to ask for help” [P15].

While participants were generally positive about the human tutor’s role in getting them unstuck, they had mixed opinions on iSnap’s ability to do so. Some mentioned its ability to nudge them in the right direction, “if you’re stuck, kinda getting me on the right path” [P12], or to help them solve the problem without wasting time: “this kind of help is better on... programming efficiency” [P15]. Two mentioned that iSnap should intervene in these situations: “if you’re having issues too many times, then maybe it can provide some suggestions” [P13], while others thought this could be disruptive. Some participants noted that iSnap was useful even when they weren’t stuck, as a means of “confirming that I was doing the right thing” [P15]. Others found that iSnap “wasn’t really helping” [P3] them get unstuck, due to difficulty interpreting the hints or iSnap’s lack of perceptiveness.

7.3.2.3 Accessibility and Salience

Accessibility refers to the perceived availability of the tutor and their help, and how convenient it is to access help\(^2\). Salience is often intertwined with Accessibility and refers to how noticeable the help is, or how present it is in the minds of students. Participants spoke of Accessibility and Salience almost exclusively in regards to iSnap, which had a “convenient feature to ask for help” [P14]. The help buttons were considered a “low risk, high reward” [P4] option because they “were there, and it’s really quick and easy just to click on one and see what it says” [P12]. Many noted that iSnap’s help buttons “were kind of there” [P4], subtly pervasive “from the very start” [P15], though one did recall forgetting about iSnap until prompted by the experimenter. While salient, the buttons were also appreciated for their subtlety; they were “non-intrusive and non-intrusive; I think is a very good way of having it” [P4]. iSnap’s Accessibility and Salience were important factors in enabling help-seeking for some, especially when stakes were otherwise low: “I figured I might as well at least see what it’s trying to suggest” [P4]. Participants noted for both tutors that they sometimes asked for help because the experimenter or tutor prompted them to (as described in Section 7.2.2), indicating that the help was not salient: “I was motivated to click on the hint bubbles... when you reminded me I could have help if I wanted” [P11].

Our human tutor’s help was not described as inaccessible; however, when asked about useful aspects of the human tutor, participants rarely mentioned Accessibility or Salience. One participant did say that “it wasn’t really an issue having to ask [the tutor]” [P9]. However, another participant

\(^2\)We are not referring here to accessibility as a system’s ability to support users regardless of physical or mental ability.
did describe previous experiences with professors' inaccessibility, having “to wait on... the professor to come help you” [P15], and others mentioned that iSnap would be useful when humans were unavailable: “the little automated stuff is helpful... especially if I'm doing something in my room” [P6]. One noted that the inaccessibility of human help can be an intentional aspect of coursework: “I wouldn't be allowed to access [help] because it's a project for school” [P10].

7.3.2.4 Perceptiveness

Perceptiveness refers to the student’s belief that a tutor is aware of the student’s code, current objectives and possible mistakes, and can provide help based on this knowledge. There was consensus that the human tutor was highly perceptive. The human tutor knows “what you're trying to do” [P14], “what your goal is” [P7] and “where I’m going wrong” [P1]. The human was also perceived to understand the context of the programming assignment. Participants did not directly link the human tutor’s perceptiveness to help-seeking, but it was cited as a reason to trust the tutor: “[her help] was very reliable, since she’s right here can see exactly what I have and what I've done” [P12].

There was a general sentiment that iSnap was not perceptive. iSnap “only can respond to what I'm putting on [the screen]” [P11], and so it misses important visual and audio queues that the human would pick up, especially since the participant is thinking out loud. It was perceived as being unresponsive to changes in student code: “It doesn't know when you make an error” [P7] and it “couldn't anticipate what I wanted so it couldn't really help” [P14]. When asked what discouraged them from using iSnap, one participant explained, “it doesn't know what I want to do so it can't tell me what to... do” [P14]. Another responded that in the instructional video for iSnap, the speaker advised students to use their best judgement when receiving help, noting that one suggestion in the demo was not useful in this situation. This possibility for error made the participant “feel like the computer doesn't understand” [P1]. It is clear that a lack of perceptiveness directly dissuaded students from using iSnap. One student did describe iSnap as perceptive, “I realize it knows what what I’m trying to get at” [P3], but thought iSnap was unable to translate this knowledge into an interpretable hint. Another, who had a more positive experience with iSnap, experienced some perceptiveness: “it looks at your code and then it tells what you should probably do” [P15].

7.3.3 Attributes of Help

7.3.3.1 Specificity

Specificity refers to how precise a tutor’s help was, whether it consisted of directly actionable suggestions or a higher-level outline. Participants appreciated the human tutor’s ability to provide both exact advice: “it was more precise... she would be able to say exactly what I needed” [P12], and high-level advice: “tutors have always been helpful... they don't just give you the answers... they actually try to walk you through how to find the answer” [P6]. The former was appreciated for its
efficiency and helpfulness, while the latter was appreciated because it allowed participants to retain a sense of independence and agency (see Section 7.3.1.3). As one participant noted, “*instead of straight-up just telling me where the commands were, she provided a path for me to think... I think that was helpful*” [P8]. While participants did not directly link Specificity to help-seeking or avoidance, it was tied to the perceived “helpfulness” of the tutor’s advice.

Participants also perceived iSnap as providing a variety of types of help. Participants noted appreciating high-level advice: “*it was really good to get the outline of like ‘oh this is what needs to be done, not what I have’*” [P12], as well as “*pretty exact advice*” [P15]. Others perceived the help to be so low-level that it was too basic, “*more syntactic than anything else*” [P11]. Participants who never used iSnap’s help had mixed expectations of how specific it would be, inferring either that it helps with low-level things “*like the formatting*” [P7] or high-level things like the “*skeleton of the program*” [P9]. Since all participants viewed the same instructional video on iSnap’s help, these diverse opinions may have been reflections of expectations based on previous experiences with computer help.

### 7.3.3.2 Interpretability

Interpretability refers to how easily the help provided by the tutor can be understood and applied by the student. Participants rarely discussed the interpretability of the human tutor, but when they did their comments were positive and terse. The tutor’s help simply “*made sense*” [P10], and this influenced participants’ trust of the tutor: “*I would trust in her opinion... whenever I followed what she did it made sense*” [P15]. Participants were much more critical of iSnap’s interpretability: “*half the things you would click on didn't make sense for the program*” [P10]. A common reason was that iSnap offered no explanation for its suggestions: “*I found it unhelpful, mostly because I wasn't quite sure why it was offering the help that it was offering. It was just like, here's a suggestion- but why?*” [P4]. One participant followed the advice but found the results difficult to interpret: “*I did that and nothing different happened*” [P1]. Others chose not to follow the advice because of difficulty interpreting: “*I saw it, I just wasn't quite sure how to go about implementing it*” [P12].

While some students found the hints too difficult to interpret, others found the need for active interpretation to be worth the time it took: “*I would kind of have to interpret and figure out what it meant... If I was completely lost it would be a really good idea to figure out what it was*” [P12]. Participants recalled that hints that did not make sense at first could become more clear in retrospect after progressing further in the program: “[it] gave me a hint as to where I need to go later, even though I didn't think it was applicable when I first saw it” [P15]. One participant even realized, in the middle of explaining why a hint did not make sense, that its suggestion was in fact applicable but hard to understand in the moment: “*they wanted me to change it to an equals sign... um come to think of it I should have set that equal to something... it just wasn't like saying things clearly*” [P14]. Sometimes this interpretation can lead to a student making an unintended discovery. One participant added
a custom length parameter to their “draw brick” block, a more advanced feature, based off of a misinterpreted, unrelated hint.

7.4 Discussion

We now return to our research questions. To address RQ4.5, we identified key factors which influenced participants’ willingness to seek help. Many of these correspond to factors identified in previous research, such as the importance of Trust in a tutor’s ability [Van88] or the Accessibility of help [Rya01]. Independence, and the threat that help can present to it, are also well established reasons for help avoidance [But98; Kar04]. Students’ experiences of being Stuck leading to help-seeking fit nicely into Newman’s dynamic self-efficacy level (SEL), which can prompt help-seeking if it falls below a threshold [New94]. However, our results place more emphasis than existing work on students’ Previous Experiences and Expectations, as well as their beliefs about the tutor’s qualities, such as Perceptiveness. Our results show these factors have direct and indirect influence on help-seeking behavior. Additionally, while previous work mentions “evaluating help” as an important step in help-seeking [MF08; Nel81; New94], our results explore specific attributes of help, such as Specificity and Interpretability, that may be particularly relevant when evaluating the quality of computer-based help.

Our results also speak to how the domain of programming specifically impacts students’ help-seeking behavior. Programming students are more likely to have had Previous Experiences with computer-based help (e.g. compilers and documentation) that shape their Expectations. Programming problems have many valid solutions and require students to make design choices, which may make it difficult to Trust help that contradicts those choices: “that way would work but my way will also work too” [P11]. That same open-endedness makes clear explanations particularly important when offering help, and participants noted that iSnap’s lack of explanation hindered its Interpretability. Some participants believed that one can “puzzle your way through” [P15] programming problems with enough time and effort, which justified their decision to continue working Independently. Each of these attributes of programming present specific obstacles that should be addressed to provide effective, computer-based help in this domain.

7.4.1 Differences and Design Implications

In this section, we address RQ4.6, which deals with differences in how students experience help-seeking factors with human and computer tutors. We focus on how these differences have design implications for programming ITSs. While not every student needs help, many students do avoid help when they need it [Ale03; Pri17c], so it is important that ITSs are designed to facilitate help-seeking. Our methods and analysis were not intended as an evaluation of iSnap; rather, we use
students’ experiences with iSnap to highlight the challenges and opportunities that arise when providing computer-based, on-demand help to novice programmers.

Our study highlights some key challenges faced specifically by computer-based help. Unlike with human help, few students have experienced intelligent computer tutoring, which can lead to very low expectations for computer tutors. These expectations will shape how students interact with help, whether or not their preconceptions are accurate. Designers may be able to address this by explaining how their help system works, how it can be used effectively and how it differs from other help technologies (e.g. compilers, documentation). iSnap, for example, could explicitly state that its hints contain advice on achieving a correct solution based on previous students’ work, which P1 noted as important in changing their perceptions and trust. Compared to human tutors, participants described trusting a computer as difficult, in part because of preconceptions about its inherent limitations. This was closely tied to a feeling that the computer was not, or could not be, perceptive or adaptive in the way that a human could. For help systems that rely on intelligent algorithms, it may be possible to mitigate this by subtly displaying what the help system does perceive and believe about the student’s current state. iSnap could accomplish this by displaying the assignment objective it believes the student is currently pursuing, along with hints for this objective. In short, transparency and explanations of the computer’s behavior may help to avoid confusion and unfounded mistrust.

Participants also noted some key advantages of computer-based help, describing it as efficient, salient and accessible. iSnap achieved efficiency in part by keeping help light-weight and visual, avoiding the need for “3 paragraphs for each [hint]” [P4], which one participant cautioned against. However, the simplicity of the help’s presentation may have also led students to believe that the help was not very sophisticated (i.e. perceptive/intelligent). iSnap’s accessibility and salience were enhanced by embedding the help buttons directly into the code, with help constantly visible and available. iSnap also seemed to enable some students to seek help but still feel independent. This may be because computer-based help is private and does not risk impacting how others perceive one’s competence or social status, as human help does [Rya01]. Just as students may be more willing to seek help from peers than teachers [Van88], they may be even more willing to seek help from computers. Designers should craft help which minimizes the threat to students’ independence and avoids unnecessary intervention, e.g. like “Microsoft Word and Clippy... popping up” [P4]. iSnap addressed this in part with the subtlety of its hints, but it could go further by adapting the quantity and specificity of hints to a student’s ability level as other ITSs do [LD99; WW99]. Overall, our results suggest that computer tutors are not simply a “worse” version of human tutors; rather, they offer distinct advantages and disadvantages, and may fill an important niche for students unable or unwilling to seek human help.
7.4.2 Hypotheses

In addition to addressing our research questions, our results also prompted us to propose new hypotheses about how and why students seek programming help from humans and computers:

**H1: The same factors shape students’ help-seeking process with both human and computer tutors.** We found that few of our initial line-by-line codes directly referenced the human or computer, and when they did, they were often pairs (e.g. “human sees what I’m doing”; “iSnap doesn’t know what I’m doing”). We therefore selected focused codes that were independent of the tutor itself (e.g. “Perceptiveness”). Our results suggest that the same factors (e.g. Independence, Trust, Accessibility) were relevant with human and computer tutors, but they may matter more for one type of tutor. Our presentation of results reflects this, as the description of each factor includes references to both human and computer tutors. For some factors, one type of tutor was referenced more frequently, perhaps indicating increased importance.

**H2: Help-seeking is not simply a triggered response to difficulty; it can be a problem-solving strategy.** It is clear from our results that being Stuck, on its own, did not always lead to help-seeking, in part due to the participants’ desire for Independence. However, participants also reported asking for help for reasons besides perceived difficulty, such as the Salience of help (“buttons were kind of there” [P4]), a desire to check one’s work (“confirming that I was doing the right thing” [P15]), or expediency (“I just want to get it right quickly” [P7]). For these students, help was not simply a means of overcoming difficulty, but rather a tool to use in problem solving. We hypothesize that the extent to which students view help-seeking as a viable strategy is the product of a constant negotiation of many factors, which we call the Student Mindset. This contrasts with existing models of help-seeking, in which a help request always arises from a student’s recognition of difficulty [MF08; Nel81; New94]. We believe that difficulty is a key factor in prompting a student to seek help, but it is neither necessary nor sufficient.

7.4.3 Limitations

This qualitative analysis of a small dataset has inherent limitations, and our results therefore lend themselves to recommendations and hypotheses rather than definite conclusions. This work is a necessary and expected first step in a larger research effort to understand help-seeking in computing. Our analysis required thoughtfulness and creativity on the part of the researchers, which means it is also a reflection of the researchers and their biases [SC90]. We attempted to minimize this by recognizing our biases, discussing results among the coders, and consistently supporting our claims with quotations.

The assignments we selected, especially the GuessingGame assignment, were designed for novice programmers and may have been too easy to produce a genuine need for help in some of our more advanced participants, a few of whom reported having done a similar assignment in class.
Participants were also new to Snap!, and some help requests centered on its interface, rather than programming generally, which may confound our results. Lastly, we have intentionally avoided a discussion of gender in this chapter, as we felt that our dataset included too few females (4) to produce a meaningful comparison, but we acknowledge that it is also a major factor in help-seeking [Ale03; But98].

7.5 Conclusion and Future Work

In this work we have presented a qualitative analysis of 15 students’ interviews after solving programming problems with both human and computer-based help. We identified 3 categories of factors that directly and indirectly impact students’ help-seeking behavior: Inputs, Student Mindset, and Attributes of Help. Our results suggest that students have very different expectations of human and computer tutors, rooted in previous experiences. These Inputs set a baseline for how the student perceives the tutor and the role of help (the Student Mindset), which impacts their help-seeking behavior. If and when students experience help, the Attributes of Help may alter their perceptions of the tutor. We contrasted students’ experiences of these factors to understand the differences in how human and computer tutors are perceived. For our participants, a human tutor seemed more trustworthy, perceptive and interpretable, while a computer tutor seemed more accessible and less threatening to a student’s sense of independence. These initial results have important implications for how we provide help to students, especially through the design of ITSs for computing, as discussed in Section 7.4.1.

Future work will address aspects of the help-seeking process that we identified but did not discuss here: the ways students seek and avoid help, the types of help the tutor provides, the interface through which help is mediated, outcomes of receiving help and how the students’ perceptions are updated after seeking or avoiding help. Our goal is to construct a preliminary model of help-seeking for both human and computer tutoring in the domain of programming, which can serve as a theoretical basis for designing more effective help systems. We plan to use the think-aloud and log data we collected while students programmed to help construct this model and address the hypotheses raised earlier.
8.1 Research Questions

In this section I discuss how this work has addressed the following research questions:

**RQ1** How can we effectively integrate the support features of intelligent tutoring systems into a novice programming environment?

**RQ2** How can we leverage student data to automatically generate hints for open-ended programming problems?

**RQ3** How can we evaluate and compare the quality of techniques for generating data-driven hints?

**RQ4** What factors impact students’ willingness to seek and use programming help?

8.1.1 Research Question 1

RQ1 asks, *How can we effectively integrate the support features of intelligent tutoring systems into a novice programming environment?* In the context of this research question, effective integration means preserving, or even amplifying, the useful attributes of a novice programming environment (NPE), while augmenting it with the adaptive support of an intelligent tutoring systems (ITS). Chapter 3 discusses the design and evolution of iSnap, a programming environment created to demonstrate that such integration is possible. However, each chapter in this document also contributes to answering
this question by presenting evaluations of the system and its underlying hint generation algorithm. Reflecting on this process of creating and evaluating iSnap, I have developed the following design recommendations to address RQ1:

**Design Flexible Support Features:** One of the strengths of NPEs is that they can be easily adapted by instructors to support a variety of learning settings and computing curricula, including computational thinking [Rut10], computer science [MS10], data analysis [Bar17a] and STEM+Computing [Rep15]. Any support feature in an NPE, therefore, needs to be just as flexible, allowing instructors to easily adapt it to new assignments. This is not possible with the existing support features of ITS, which have historically relied on experts to construct complex domain models (e.g. model tracing [CA01] or constraint-based modeling [Mit98]) that are only useful for a limited set of problems. Instructors lack the time and expertise to extend these models to support new and different assignments. iSnap addresses this by using assignment-agnostic, data-driven hint generation (Chapter 4), that requires only a set of example solutions to generate programming hints and feedback. Instructors can easily collect a semester’s worth of student solutions, or author their own, with minimal time and no special expertise required. In this way, iSnap maintains the core flexibility of NPEs, while supporting students with hints and feedback.

**Build on Existing Tools:** Learning environments are only effective if students use them. One of the strengths of NPEs is that they are already popular in K-12 classrooms [Bro14; Coo17; Hub15; Gar15], online and informal learning settings [Fie13; Mal08] and, to a lesser extent, undergraduate classrooms [Gar12]. As a result, they have an enormous potential for impacting students. The best way to leverage this popularity of NPEs is to extend existing systems, which have already been integrated into classrooms and curricula, rather than building new ones. iSnap does this by extending Snap!, a popular NPE that was used by over 3,500 high school students taking the Beauty and Joy of Computing (BJC) curriculum [Gar15] in 2017. More importantly, hundreds of K-12 teaches have received training on how to use Snap! during BJC professional development (PD) workshops [Pri16c]. iSnap was used in the 2017 BJC PD, and has been presented multiple times at SIGCSE [Pri17d; Pri18a], the primary conference for practitioners of CS education. When iSnap is ready to scale to reach more learners, this tight integration with Snap! and BJC will make that possible. This process has already begun in a few BJC high school classes, where teachers reported little difficulty transitioning from Snap! to iSnap.

**Embrace Visual, Interactive Feedback:** A core strength of NPEs is their visual and interactive nature. Many, including Snap!, enable students to write programs with complex output (e.g. games, animations). They also employ novel programming interfaces, such as block-based programming, which allow students to visualize and directly manipulate the abstract syntax tree of their code. Rather than relying on the traditional, text-based help messages of many programming help systems (e.g. [Cho17; Ger16; RK17]), I recommend that NPEs should provide support in a graphical and interactive manner that matches the rest of the programming environment. iSnap does this by
embedding help directly in the programming interface, highlighting blocks and annotating them with help buttons. While feedback is active, it updates dynamically as students modify their code. In Chapter 7, I showed that this visual, interactive feedback was considered accessible and salient by students.

**Support Creative Curricula:** Arguably the most definitive attribute of NPEs is that they enable students to make interesting, creative artifacts, such as games [Köl10; Mal08], stories [Kel07] and music [Fre14]. Any support feature for an NPE should be compatible with these more open-ended assignments. This is a challenge for the more rigid expert models employed by ITSs, which describe correct behavior, rather than creative behavior. While iSnap falls well short of providing support for truly open-ended assignments, it makes some progress in this direction. As discussed in Chapter 4 and in the next section, iSnap's data-driven hints are designed to support assignments that enable customization and design choices where students submit a diversity of solutions. As a result, iSnap is capable of supporting assignments that require 20 or more lines of code, including more creative tasks, such as creating a simple game of Pong.

### 8.1.2 Research Question 2

RQ2 asks, *How can we leverage student data to automatically generate hints for open-ended programming problems?* Chapter 4 presents the SourceCheck data-driven hint generation algorithm for programming, which was designed to answer this research question. SourceCheck extends my own previous research [Pri16b] on generating hints, specifically for the more open-ended programming problems found in NPEs. The foundational finding of this research is that these open-ended programming problems include enough design choices and customization that students’ code is unlikely to overlap exactly. This means that for any student requesting a hint, it is very unlikely that iSnap has ever seen a student with the same code (even ignoring differences in identifiers and literals, which students can name). Considering the Guessing Game problem, completed by nearly 300 students in iSnap over 5 semesters, less than 10% of the code states iSnap has recorded are ever seen in multiple students’ work. The challenge for SourceCheck is how to generate a data-driven hint for a student without any match in the historical data.

SourceCheck and its predecessor, CTD [Pri16b], both address this sparsity of data by focusing on the parts of students’ programs that do overlap, even when the full programs do not match, and using these overlapping segments to generate hints. Rather than looking for an exact match for a hint-requesting student, SourceCheck looks for the best match in its dataset of correct solutions. In order to provide hints using this inexact match, SourceCheck breaks both the student’s code and the matching solution down into hierarchical levels, and it pairs these levels independently, generating hints at each level. In doing so, SourceCheck is leveraging the insight that there may be multiple acceptable ways to order and arrange the code elements that make up a correct solution. By
breaking the code down into smaller parts, SourceCheck is able to provide hints to a student using a best-matching solution, even if those two programs have differences in code structure. However, this choice to break code down into constituent parts for hint generation comes with a tradeoff. It provides greater potential for overlap between students’ solutions, facilitating hint generation, but it deemphasizes the larger structure of the code that puts each part into context. The CTD algorithm [Pri16b], previously used in iSnap, largely ignored this larger program structure, sometimes leading a student to a solution with correct pieces but an incorrect overall structure. SourceCheck attempts to strike a better balance between low-level and high-level program structure, ultimately pointing a student towards a single, best-matching correct solution.

Chapter 6 also addresses RQ2 by highlighting how data-driven hint generation for iSnap’s more open-ended problems may pose challenges that are not encountered in generating hints for programs to solve simple data processing problems. I found that for iSnap problems, using additional student data to generate SourceCheck hints did not always improve hint quality, and that it can even be harmful, though this was not the case for the shorter problems from the ITAP dataset. While these results were initially unexpected, a closer inspection of how SourceCheck interacts with student data showed that SourceCheck’s reliance on a single target solution for hint generation plays an important role in this sensitivity, especially for assignments that encourage more customization and design choices. This choice for SourceCheck to use a single target solution for hint generation, rather than integrating multiple solutions as CTD does, is part of the tradeoff between emphasizing high-level and low-level program structure, discussed above. It ensures that students are always directed towards a fully correct solution, but it makes the algorithm more sensitive to a single poor match. However, this tradeoff seems to matter less for the the shorter problems in the ITAP dataset, where using a single solution for hint generation was not a problem, suggesting that this challenge may be exacerbated by the open-ended nature of the iSnap problems. In Section 8.3, I discuss how future work can better support these open-ended problems.

8.1.3 Research Question 3

RQ3 asks, How can we evaluate and compare the quality of techniques for generating data-driven hints? This work addresses RQ3 by presenting three different methods for assessing the quality of data-driven hints using experts. Each of these methods yields additional insight into the strengths and weaknesses of iSnap’s hints. In Chapter 5, I asked experts to directly rate the quality of iSnap’s hints (generated using SourceCheck’s predecessor [Pri16b]) using a rubric. The results suggested that iSnap’s hints were generally high-quality (rated a median 7 or 8 on a scale that ranged from 3 to 9), but that there was also a high variance in hint quality.

In Chapter 4, I had SourceCheck, as well as two human tutors, generate hints for a set of student hint requests and asked whether SourceCheck’s hints could be distinguished from those of the
two human tutors. SourceCheck was able to replicate human tutor hints at nearly the same rate of the human tutors (76-88% as well); however, human tutors were much less likely to replicate SourceCheck's hints than each others' hints (47-70% as likely). This suggests that SourceCheck can somewhat reliably produce tutor-quality hints, but it also produces many less useful hints.

In Chapter 6, I again compared SourceCheck's hints against human tutors' hints. This time, however, the goal was to compare different approaches for hint generation and to better understand the factors that impact data-driven hint quality. I found that using more student data to weight the importance of each hint can significantly improve hint quality and that the quantity and source of training data have a meaningful impact on data-driven hint quality.

Importantly, all three of these hint rating procedures determined hint quality using actual student hint requests. Previous work on evaluating data-driven hint quality has used a sample of all program states observed in the dataset (e.g. [Pie15]) or expert-authored buggy states (e.g. [Wat12]) to evaluate hint generation algorithms. As discussed in Chapter 6, there are important differences between the code that students have at the time of a hint request and the code of the general student population for a given problem. Anecdotally, our hint authors noted that students who request hints often have code that diverges starkly from a correct solution, usually due to multiple interacting misconceptions, in a way that the code of the general student population does not. Future work should attempt to quantify and understand these differences more thoroughly.

One limitation of the evaluation methods presented in this document is that they rely on experts, rather than students, to determine the quality of data-driven hints. This was an intentional choice, designed to make these quality metrics more scalable and reusable. The expert-authored hints used in the evaluations in Chapters 4 and 6 can be compared with any number of newly generated hints to determine their quality. Any evaluation using students would have to be reproduced to rate each new batch of hints. Additionally, in Chapter 5, I found that expert hint ratings strongly predicted students' willingness to follow a hint and their willingness to request additional help from iSnap, suggesting that these expert quality ratings are fairly representative of students' perceptions of hint quality.

8.1.4 Research Question 4

RQ4 asks, What factors impact students' willingness to seek and use programming help? I identified two classes of barriers that can deter students from seeking and using help when programming. The first is the quality of that help the student receives. In Chapter 5, I identified a significant, positive correlation between the quality of the first and second hints a student receives and the amount of additional help that the student requested. I found that students also hold hints that they follow to a high standard of quality (followed hints were rated 8 to 8.5 on a scale that ranged from 3 to 9). The notion of quality used in this evaluation considered not just how well the hint progressed
a student towards a correct solution, but also how relevant and interpretable the hint was. These results suggest that the quality of help is an important factor influencing students’ willingness to seek help, which is especially important for automatically generated hints, where quality cannot be guaranteed by an expert.

The second class of barriers is more general and includes factors that influence help-seeking even before the student has received help. These factors may be even more important than the quality of help, given the results in Chapter 5 that the majority of students never ask for a first hint in iSnap, despite many students struggling on assignments. Chapter 7 discusses these barriers in detail. Some important factors are students’ desire for independence, their trust in the help system, their previous experiences with computer-based help and the accessibility and salience of the help. I suggest that some of these barriers can be overcome by designing a help system differently, for example by reframing the help features as a “tool” that savvy programmers use, rather than a source of outside help that may diminish a student’s feelings of independent accomplishment.

8.2 Contributions

This work makes the following research contributions:

1. The iSnap programming environment itself is a major contribution of this work. It is the first programming environment to combine the support features of block-based programming and automatically generated hints, both of which have been shown to improve student learning (see Sections 2.1.2 and 2.2.2 of Chapter 2). Its automated support features make it easy to adapt to new classrooms, and it has been deployed in an undergraduate course and a large-scale K-12 teacher professional development program. It also supports the Beauty and Joy of Computing (BJC) curriculum [Gar15], used by thousands of high school students each year, making a larger-scale deployment feasible.

2. The SourceCheck data-driven hint generation algorithm is another contribution of this work. While I present the algorithm primarily in the context of its use in iSnap, SourceCheck is programming-language agnostic. It is the first data-driven hint algorithm to be evaluated on multiple programming datasets, generating hints in two different programming languages (see Chapter 6). Additionally, SourceCheck and its predecessor [Pri16b] were the first data-driven algorithms designed specifically to generate hints for more open-ended programming problems, as found in the BJC curriculum.

3. I demonstrated evidence that highlights the importance of data-driven hint quality and its impact on students’ willingness to seek and use help when they need it. While not surprising, this is an important finding for the growing field of automated programming hint generation,
which still relies heavily on technical evaluations, rather than evaluations of hint quality (see Chapter 6).

4. I presented novel methods for evaluating the quality of data-driven hints using experts, which allow researchers to benchmark and compare hint generation algorithms. I have applied these methods across multiple datasets to identify challenges in data-driven hint generation and solutions to those challenges.

5. I identified other factors besides hint quality that may be even more important in influencing students’ willingness to seek and use help when programming, and I showed how these results can inform the design of better-utilized help features.

8.3 Future Work

The work presented here lays a foundation for integrating the support features of ITSs into NPEs. While my research with iSnap demonstrates that this is possible, it also highlights two primary barriers that must be overcome for iSnap to be used effectively. The first barrier is the inconsistent quality of data-driven programming hints, as demonstrated in Chapters 4, 5 and 6. Ensuring hint quality is a challenge for SourceCheck, but as more data-driven hint generation algorithms are introduced, it is also important for the research community as a whole to set standards for data-driven hint quality. I am in the process of adapting the Quality evaluation procedure introduced in Chapter 6 to benchmark and compare a set of published hint generation algorithms on common datasets, which will be released publicly. This will allow the research community to address this challenge directly, and to develop clear understanding of what constitutes the state of the art in hint generation.

The second barrier to iSnap’s effective classroom use is students’ unwillingness to seek and use help when they need it. While higher-quality hints should address this somewhat, as discussed in Chapter 5, many students never ask for hints in the first place. My investigation of help-seeking in Chapter 7 points to some design choices that may improve help-seeking, which should be evaluated empirically. However, additional investigations of students’ help-seeking will likely be needed to fully understand how to make support features most accessible.

Once these two barriers are overcome, the most important piece of future work for iSnap will be a controlled study to evaluate the impact of data-driven hints on student learning. While iSnap is built on support features with clear empirical support, NPEs and adaptive hints, an important limitation of this work is that I have not measured the impact of these two support features working together. However, because of the barriers of inconsistent hint quality and low hint usage, I chose to leave such an evaluation for future work.
The long-term future of intelligent support for programming goes beyond my work with iSnap and SourceCheck. Most existing data-driven hint generation algorithms, including SourceCheck, focus on edit-based hints that tell a student exactly how to edit their code (e.g. [LB14; Per14; Pie15; Pri16b; RK17; Wan17]. While I have advocated for better methods for evaluating and comparing these next-step hints, my research also sheds light on many of the weaknesses of these next-step, edit-based hints. Chapters 3 and 5 highlight how students can easily abuse “bottom out” hints to complete a problem without trying. Chapters 3 and 7 also show how next-step hints can be difficult to interpret, even when they are ostensibly correct. The gold standard hint generation procedure discussed in Chapter 6 also revealed that it can be quite difficult for human tutors to express the advice they think is most appropriate in the form of a set of edits to a student’s code, without any natural language.

For these reasons, I believe that future work will need to find better, data-driven ways to support students, going beyond next-step hints. As Suzuki et al. [Suz17a] argue, there is a need to explore the design space of data-driven hints for programming. Additionally, beyond hints, what are other forms of feedback that could be automatically generated from student data? Some examples of future data-driven programming support might include data-driven worked examples, showing a student a step-by-step explanation of how to solve a related problem, and higher-level hints, which point the student to a more general future goal, rather than an immediate next step. Both data-driven worked examples [Mos15] and higher-level hints [Eag12] have already been implemented in the domain of logic proofs. Data can also be used to determine effective times to prompt students for self-explanation, which has a strong potential to promote learning if done correctly [Ale16; Shi08].

Ideally, these new forms of help will be used to support even more open-ended, creative projects in NPEs. The most compelling of these projects are those that connect with students’ interests, values and learning goals. Data-driven help should be able to support computing contexts like EarSketch [Mag16], which allows students to remix sound clips using a Python API to create new music that connects with their interests, and the CORGIS project [Bar17b], which lets students investigate meaningful questions by analyzing large, real-world datasets through introductory programming. In these contexts, students can bring their own design goals to the project and help to define what a successful outcome looks like, making it difficult to support students with existing help systems that require a model of correct problem-solving behavior. Future work on data-driven support will need to leverage higher-level forms of help that work with students to address their challenges and objectives, rather than simply offering a path to a solution. The work outlined in this document takes an important step towards supporting these more creative and open-ended programming projects, with the hope that future work will make this goal a reality.
BIBLIOGRAPHY


A.1 **iSnap Assignments**

Various chapters in this document refer to the following assignments, which were completed by students working in iSnap.

A.1.1 **Polygon Maker**

**Step 1:** Create a new block that can draw any polygon. It should have a total of 3 input areas for:

- the size of each side
- the number of sides
- how thick the pen is

**Step 2:** Perform these tests:

1. # of sides = 4, size = length of 100, thickness = 5
   This should draw a square

2. # of sides = 10, size = length of 50, thickness = 6
   This should draw a decagon
3. # of sides = 50, size = length of 7, thickness = 2

What do you think this will draw?

A.1.2 Squiral

In this activity you will build a block in Snap! that makes your sprite draw a squiral like the one below:

Give the block an argument that allows the user to set the number of times the sprite completes a full rotation around the center. (The picture has at least 5 rotations).

A.1.3 Guessing Game 1 (GuessingGame)

[Guessing Game 1] Item 1: The computer chooses a random number between 1 and 10 and continuously asks the user to guess the number until they guess correctly.

Item 2: Make sure that your program contains the following:

- Welcome the user to the game
- Ask the user's name
- Welcome the user by name
- Tell the user if their guess was too high, too low, or correct

A.1.4 Guessing Game 2

Please make sure that all items from Guessing Game 1 are included in this assignment, as well as the following:

1. Allow the user to choose minimum and maximum numbers that the random number will be between
2. Keep track of how many guesses the user used
3. Report the number of guesses when the user wins

A.2 ITAP Assignments

These assignments were completed by students using the ITAP [RK17] intelligent tutoring system. A dataset consisting of attempts at these problems was analyzed in Chapter 6.

A.2.1 HelloWorld

Write a function, hello_world, which takes no parameters and returns the string "Hello World!".

A.2.2 FirstAndLast

Given a string, s, return the combination of the first letter and the last letter of the string. You can assume that s is at least two characters long.

A.2.3 OneToN

Given a number n, return a string that contains the numbers from 1 to n. (So 5 results in "12345")

A.2.4 IsPunctuation

Given a one-character string, write a function, is_punctuation, which returns whether that character is a punctuation mark. Hint: you can do this by importing the string module and using the built-in punctuation value.

A.2.5 KthDigit

Given a number, x, return the kth digit (from the back) of x. In the number 1234, the 1st digit is 4, the 2nd is 3, etc. You can assume that x is positive. Note that // is truncating integer division in python.
The following rubric was used in the hint rating procedure in Chapter 5. Ratings were made while looking at the student's code at the time of the help request, along with the content of the hint dialog. Raters had access to the student's code history as well.

**Relevance**: Think of all the goals that the student may have had when they asked for the hint, based on what they have accomplished, the errors they have and what they have left to do. How likely is it that the hint they received addresses one of these goal.

1. The hint is not at all relevant to the student's likely goals.
2. The hint is somewhat relevant to a likely goal, or addresses a goal that is possible but not likely for the student to be pursuing.
3. The hint is directly relevant to a likely student goal.

**Progress**: Will following this hint bring the student closer to correctly completing the assignment, by progressing the code or removing erroneous code? Note that if the student has a non-traditional solution, the hint should be judged with respect to the student's code, not an "ideal" solution.

1. The hint suggestions is either not useful or removes useful code.
2. The hint suggestion is technically or partly correct, but may not be fully necessary. For example, the hint may suggest a better solution for code that is already correct. Alternately, the hint may contain some edits which are useful and others that are not.

3. The hint moves the student towards a correct solution, by adding useful code or removing erroneous code.

**Interpretability:** How easy is it for a novice to understand the value of the suggestion?

1. The hint is either incorrect or the value of the suggestion would not be clear without further explanation. Following this hint would not likely get the student "unstuck" without further hints.

2. The value of the hint may or may not be apparent to the student. For example, the hint may be too vague, or might come too early for the student to understand it.

3. The hint should be easy to interpret and understand, assuming the student understands the objectives of the assignment.

Important note: The first three numeric ratings should be largely independent. Hints can be:

- Correct and interpretable but not relevant: such as saying to add a variable at the start of a program, when the student has no code to use the variable.

- Relevant but not correct: such as telling the student to add a needed block in the wrong place.

- Relevant and correct but not interpretable: such as telling the student to delete a large chunk of bad code that’s causing a bug, without explaining what to save or move elsewhere.

However, a hint that is incorrect may be uninterpretable as a result, since there is no value in it to understand.
The following detailed instructions were given to students working on the GuessingGame and BrickWall programming assignments described in Chapter 7.
The Guessing Game

In this exercise, you will create a guessing game where the computer chooses a random number between 1 and 10 and continuously asks the user to guess the number until they guess correctly.

Before we get started, here are some of the blocks you might need in this exercise:

1. Welcome the player to the guessing game.
2. Ask the player for their name and then greet them by name.

Guessing the Secret Number

The most important part of the guessing game is thinking of a secret number. Your game should generate a random secret number between 1 and 10 and store it for the player to guess.

Once the game has generated the secret number, we need to give the player a chance to guess it. The player should be able to continue making guesses until they get the secret number correct. The game should only end when they guess it correctly.

Giving Feedback

To help the player out, have the game tell the player if their guess is too high, too low or correct.

Once you've tested your game and it works, you're done!
Brick Wall

In this exercise, you will write a Snap program to draw a brick wall, like the picture below:

Before we get started, here are some of the blocks you might need in this exercise:

Drawing One Brick

The first step in making a brick wall is to draw a single brick. Since we need to draw many bricks, it makes sense to make a custom block to do this.

A single brick is rectangle, filled in red. There's no draw rectangle block in Snap, but we can get the same result by thinking of a rectangle as a very thick line.
Try creating this custom block and use it to draw a few bricks:

```
draw brick
pen down
set pen color to
set pen size to 10
move 30 steps
pen up
```

**Drawing Rows**

If we put multiple bricks together, we get a *row*. There are two kinds of rows in our wall, so we can make blocks that specialize in each:

Row A: `---------------------`
Row B: `---------------------`

Try it:
1. Make blocks `rowA` and `rowB` that draw the pictures above.
2. Think about what custom blocks besides `draw brick` you might want.
3. The two kinds of rows should be exactly the same length, and the space in between bricks should be the same in each row.

**Drawing the Whole Wall**

Once you have rows A and B the same length, you can write the `Draw a Brick Wall with Rows` block.

Try testing your `Draw a Brick Wall with Rows` with 5, 6 and 7 rows. Make sure it works correctly for even and odd numbered rows.

Once you've tested your program and it works, you're done!
APPENDIX

D

SEMI-STRUCTURED INTERVIEW QUESTIONS

Instructions: The experimenter should use these questions to guide and structure the interview but allow the participant to take the conversation in unforeseen directions where appropriate. Approximate times are given for guidance, which the experimenter may adjust as necessary. An effort should be made to cover all questions. The tutor should leave the room, so the participant can be open about their experience.

1. [1-3m] First, the experimenter should take time to clarify any remarks that were unclear during think-aloud portion of the study.

2. [4m] Tell me what it was like working with [tutor name]. Think about the times when you asked for help.

   (a) What motivated or discouraged you to ask for help?
   (b) What was useful about the help and how could it be improved?
   (c) How did you feel about the help provided? Did you trust it or follow it?

3. [4m] Tell me about what it was like working with the automated Snap help.

   (a) What motivated or discouraged you to ask for help?
(b) What was useful about the help and how could it be improved?
(c) How did you feel about the help provided? Did you trust it or follow it?

4. [3m] What were the biggest differences between the you experience with [tutor name] and the automated help?

(a) Describe some situations when [tutor name] did something that you wish that automated Snap help had done more of. Why was it better?
(b) Describe some situations when the automated Snap help did something that you wish [tutor name] had done more of. Why was it better?

5. [2m] Imagine that Snap could do more than just give you suggestions on what to do next, like in this study.

(a) What types of help would you like to see?
(b) When should the help be provided?
(c) What would motivate you to use the help when you need it?

6. Is there anything else you want to tell us?