Evaluation of a Data-driven Feedback Algorithm for Open-ended Programming

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

Rui Zhi
North Carolina State Univ.
Raleigh, NC, USA
rzhi@ncsu.edu

Tiffany Barnes
North Carolina State Univ.
Raleigh, NC, USA
tmbarnes@ncsu.edu

ABSTRACT

In this paper 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.

1. INTRODUCTION AND BACKGROUND

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 [11] or data-driven methods [5, 6, 9]. However, existing techniques (including our own work [6]) have two notable limitations: the type of feedback they can provide and the methods with which they are evaluated.

Existing work 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. [6, 9, 11]) or small classroom studies that use case studies (e.g. [7]). 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 [10]. However, recent work suggests that programming hints can vary widely in quality and that low-quality hints may deter students from later asking for help when they need it [8]. A meaningful first step would therefore be to better understand and evaluate the quality of the feedback we generate. Piech et al. [5] 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.

2. FEEDBACK GENERATION

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 *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 *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 [1] and was successfully implemented by Rivers and Koedinger for programming hints [9]. 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 *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 [6]. 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.

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 [9] and approximations of tree edit distance [11], though more complex metrics have been proposed [4]. The problem with edit distances, however, is that they heavily penalize differences in the position of code fragments [3, 4]. 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 [3]. We build on this idea, along with our previous work decomposing ASTs using root paths [6], 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.

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 [9] or following edits from previous students [6]. 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.

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 [6, 9] 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) [5]. 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 focus on the following research questions:

RQ1: How well does SourceCheck's feedback agree with ideal human tutor feedback?

RQ2: 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 [2] (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[1] [7], an extension of the block-based, novice programming environment Snap*!* [2]. iSnap supports students working on open-ended assignments with data-driven, on-demand hints [6].

We selected one homework assignment (Squiral – SQ) and one in-lab assignment (The Guessing Game – GG) for analysis. In SQ, 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 GG, 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 GG and SQ 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 SQ and GG for a total of 22 and 29 hints respectively, 51 altogether.

[1]Demo and datasets available at http://go.ncsu.edu/isnap

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 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 RQ1. 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 RQ2.

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

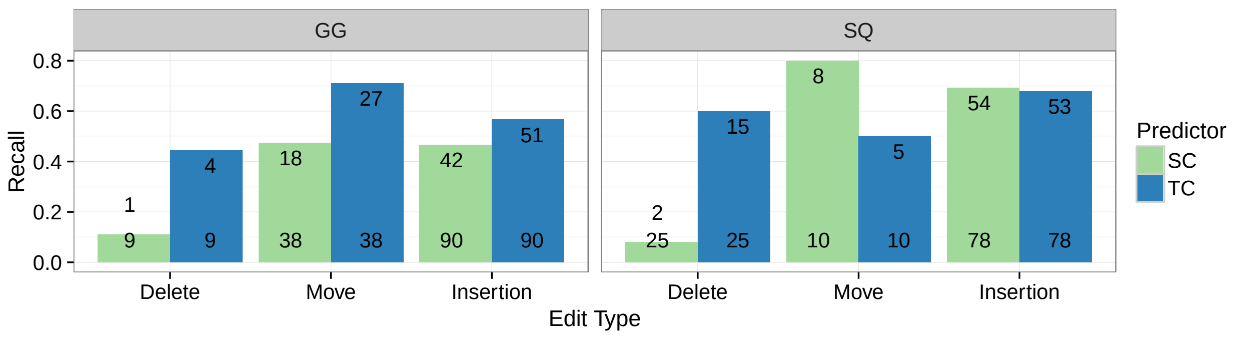

Figure 1: The percentage of tutor Phase I edits predicted by SourceCheck (SC) and TC (the recall) for each edit type on the GG and SQ assignments. Bars are labeled with the total number of Phase I edits of each type (bottom) and the number of correctly labeled edits (top).

(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 RQ1 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 RQ2, 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.

4.1 Results

We first look at predicting the targeted feedback that tutors provided in Phase I. Figure 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 GG, and 69% and 80% of Moves and Insertions for SQ, where it even outperforms TC. Totalling all edits, SourceCheck had a recall of 0.45 and 0.57 on GG and SQ 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 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 GG and SQ respectively, while TC achieved 0.57 and 0.54.

However, SourceCheck's precision is much lower, particularly for GG, 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 GG and SQ 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 GG AST nodes where the tutors also did not suggest an edit in Phase II and for 662/703 (94%) of SQ nodes. Totalling all edits, SourceCheck had a precision of 0.27 and 0.38 on GG and SQ respectively, while TC achieved 0.57 and 0.54[2].

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

[2]Note 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.

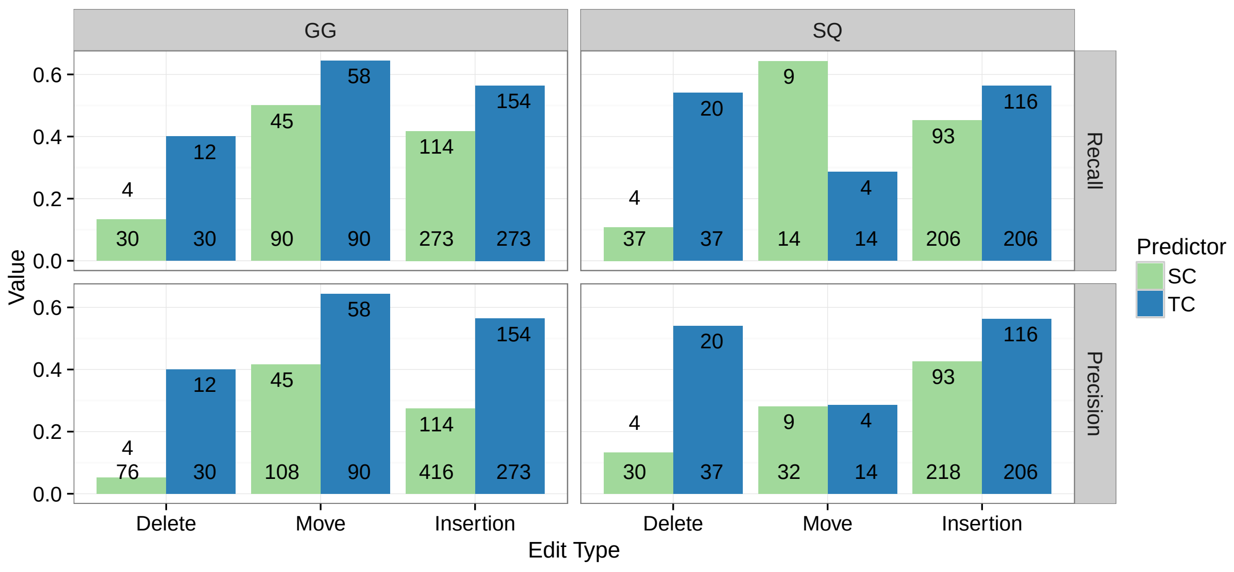

Figure 2: The recall for Phase II edits (top), as well the the percentage of SourceCheck (SC) and TC edits that agreed with a human tutor (the precision, bottom).

the Phase I deletions that SourceCheck failed to predict were instances of a tutor deleting a variable, such when a student used the wrong variable in an expression. This is largely due to SourceCheck's canonicalization process [6], 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.

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 how to reset the sprite and canvas before drawing the "Squiral" in the SQ 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.

5. DISCUSSION

RQ1: *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 GG and SQ 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. [8]). 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.

RQ2: *How does the agreement between SourceCheck and a human tutor compare to the agreement between human tutors?* Our results for RQ2 are mixed. In Phase II, SourceCheck was 72-76% as likely to agree with a given tutor's edit as another tutor was on GG and SQ (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 GG. 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, recent work suggests that students seek less help after receiving poor quality hints [8]. 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 GG and SQ assignments respectively. In fact, tutors only agreed completely on 8 out of 22 SQ snapshots (36%) and 7 out of 29 GG 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 [5].

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 [3, 9] 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 [5] 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 as similar to human tutors are they are to each other. We show that SourceCheck approaches this target in some ways and falls well short in others.

Based on our results, iSnap has been 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 GG and SQ. 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.

7. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under grant 1623470.

8. REFERENCES

[1] T. Barnes and J. Stamper. Toward Automatic Hint Generation for Logic Proof Tutoring Using Historical Student Data. In *Proc. of Int. Conf. on Intelligent Tutoring Systems*, pages 373–382, 2008.

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

[3] B. Mokbel, S. Gross, B. Paaßen, N. Pinkwart, and B. Hammer. Domain-independent proximity measures in intelligent tutoring systems. In *Proc. of Int. Conf. on Educational Data Mining*, 2013.

[4] B. Paaßen, B. Mokbel, and B. Hammer. Adaptive Structure Metrics for Automated Feedback Provision in Java Programming. In *European Symp. on Artificial Neural Networks, Computational Intelligence and Machine Learning*, page 312, 2015.

[5] C. Piech, M. Sahami, J. Huang, and L. Guibas. Autonomously Generating Hints by Inferring Problem Solving Policies. In *Proc. of ACM Conf. on Learning @ Scale*, pages 1–10, 2015.

[6] T. W. Price, Y. Dong, and T. Barnes. Generating Data-driven Hints for Open-ended Programming. In *Proc. of Int. Conf. on Educational Data Mining*, 2016.

[7] T. W. Price, Y. Dong, and D. Lipovac. iSnap: Towards Intelligent Tutoring in Novice Programming Environments. In *Proc. of ACM Technical Symp. on Computer Science Education*, 2017.

[8] T. W. Price, R. Zhi, and T. Barnes. Hint Generation Under Uncertainty: The Effect of Hint Quality on Help-Seeking Behavior. In *Proc. of Int. Conf. on Artificial Intelligence in Education*, 2017.

[9] K. Rivers and K. R. Koedinger. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education*, 16(1), 2015.

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

[11] K. Zimmerman and C. R. Rupakheti. An Automated Framework for Recommending Program Elements to Novices. In *Proc. of Int. Conf. on Automated Software Engineering*, 2015.