Next Steps for Next-step Hints: Lessons Learned from Teacher Evaluations of Automatic Programming Hints

Benjamin Paaßen*
Institute of Informatics
Humboldt-University of Berlin
benjamin.paassen@
hu-berlin.de

Jessica McBroom
School of Computer Science
The University of Sydney
jmcb6755@
uni.sydney.edu.au

Bryn Jeffries
Grok Learning
bryn@groklearning.com

Irena Koprinska
School of Computer Science
The University of Sydney
irena.koprinska@
sydney.edu.au

Kalina Yacef
School of Computer Science
The University of Sydney
kalina.yacef@
sydney.edu.au

ABSTRACT

Next-step programming hints have attracted considerable research attention in recent years, with many new techniques being developed for a variety of contexts. However, evaluating next-step hints is still a challenge. We performed a pilot study in which teachers ($N = 7$) rated automatic next-step hints, both quantitatively and qualitatively, providing reasons for their ratings. Additionally, we asked teachers to write a free-form hint themselves. We found that teachers tended to prefer higher level hints over syntax-based hints, and that the differences between hint techniques were often less important to teachers than the format of the generated hints. Based on these results, we propose modifications to next-step hint strategies to increase their similarity to human teacher feedback, and suggest this as a potential avenue for improving their effectiveness.

Keywords

computer science education, next step hints, data-driven feedback, teacher evaluation

1. INTRODUCTION

To support students in solving practical programming tasks, many automatic feedback strategies provide *next-step hints*, i.e. they select a target program that is closer to a correct solution and provide feedback based on the contrast between the student's current program and the target program (e.g. [3, 6, 10, 11, 13–15]). Next-step hints are compelling because they do not require teacher intervention. Instead, they utilize historical student data and, as such, can be fully automated [10]. However, it remains challenging to evaluate next-step hints. Price et al. [14] found at least three different criteria to grade next-step hints: how often they are available (coverage), how they impact student outcomes, such as task completion speed and learning gain, and how well they align with expert opinions. Importantly, the relation between these criteria is not trivial and different ways to present next-step hints can influence their effect. For example, Marwan et al. [8] found that adding textual explanations improved hint quality in expert eyes but did not influence student outcomes.

Our main contribution in this paper is to combine quantitative ratings with qualitative explanations. In other words, we do not only investigate differences in teacher ratings, but also *why* teachers preferred some hints over others. To this end, we performed a survey with $N = 7$ teachers, asking them to grade next-step hints generated by three different methods across three programming tasks in Python. Our overarching research questions are:

RQ1 Do teachers' ratings differ between hint methods?

RQ2 Do automatic hints align with teacher hints?

RQ3 What are teachers' reasons for preferring some hints over others?

This paper is set out as follows: Section 2 discusses related work in more detail, Section 3 describes the setup of our study, Section 4 describes the results and, finally, Sections 5-6 discuss and summarize the implications of our work.

2. RELATED WORK

Prior work on evaluating next-step hints broadly falls into three categories: technical criteria, outcomes for students, and expert opinions [14].

Technical criteria are mostly concerned with the availability of hints and motivated by the *cold start problem*, i.e.

*Corresponding Author

Joint Proceedings of the Workshops of the 14th International Conference on Educational Data Mining (EDM 2021);

the problem that data-driven hint generation requires a certain set of data to become possible [1]. Over the years, this problem has arguably become less critical as multiple methods are now available which require very little training data, such as [6, 10, 11, 13, 15]. In this paper we restrict ourselves to these methods and therefore omit such criteria.

Regarding student outcomes, prior studies have already shown that data-driven, next-step hints can yield similar learning gains to working with human teachers [5], can improve solution quality [2], and completion speed [4]. The challenge in applying such criteria is that they require a study design in which an intervention group works on-line on a task with hint support, which was beyond the scope of our pilot study.

An alternative which requires less resources is offered by expert opinions, i.e. ratings by experienced programming teachers on the quality of hints. In particular, Price et al. [13] have suggested three scales (relevance, progress, and interpretability) to grade hint quality and have shown that expert ratings on these scales are related to the likelihood of students accepting hints in the future. Further, both Piech et al. [12] and Price et al. [14] asked teachers to generate next-step hints themselves and evaluated the overlap between the teacher hints and automatic hints as a measure of quality. Importantly, a next-step hint may be affected not only by the selected target program but also by how the hint is presented. For example, Marwan et al. [8] found that adding textual explanations improved expert quality ratings – but not student outcomes.

In our work, we combine aspects of this prior work with qualitative questions. In particular, we use a variation of the three scales of Price et al. [13] for quantitative ratings of hint quality and let teachers provide their own hints to evaluate overlap, akin to [12, 14]. Additionally, we ask teachers to provide a textual explanation for why they would give a hint and why they would choose *not* to give one of the automatic hints.

3. METHOD

In this section, we cover the setup for our survey, beginning with the programming data sets we used, followed by the mechanism to select specific examples, the hint methods, and the recruitment for the survey itself.

3.1 Programming data sets

In order to provide realistic stimulus material, we selected our programs from three real-world, large-scale data sets of program traces in introductory programming. Namely, we considered data from the 2018 (beginner challenge) and 2019 (beginner and intermediate challenges) National Computer Science School (NCSS)[1], an introductory computer science curriculum for (mostly Australian) school children in grades 5-10. 12,876 students were enrolled in the beginners 2018 challenge, 11,181 students in the beginners 2019 challenge, and 7,854 students in the intermediate 2019 challenge. Each challenge consisted of about twenty-five programming tasks in ascending difficulty, each of which were annotated with unit tests. In all cases, we only considered submissions, i.e. programs that students deliberately submitted for evaluation against unit tests.

[1] `https://ncss.edu.au`

3.2 Example selection

Our goal in this study was to evaluate the quality of automatic hints in a range of realistic situations where students were likely to need help and where feedback generation was non-trivial. For the purpose of this study, we considered a program as indicative of help-need if at least five students who submitted this program failed the same or more unit tests in the next step of their development. This is in line with prior work of [9] and [7], who both suggest that help is needed if students repeatedly fail to make progress.

As proxy for non-triviality we considered the tree edit distance to the top-100 most frequent submissions for the same programming task. If this tree edit distance is low, providing automatic hints is simple: we can retrieve the nearest neighbor according to tree edit distance and use a successful continuation of this nearest neighbor as a hint, as suggested by [6]. However, if this distance is high, we are in a region of the space of possible programs that is not frequently visited by students and, hence, harder to cover for an automatic hint system.

In the end, we selected for each of the three challenges the program which maximized the tree edit distance to frequent programs and indicated help-need. The resulting submissions are shown in Figure 1, alongside with a description of the respective programming task and an example solution.

3.3 Hint generation

We considered three techniques to produce next-step hints.

Firstly, we used one-nearest neighbor (1NN) prediction [6], i.e. we selected the nearest neighbor to the help-seeking program in the training data and recommended its successor. Distance was measured according to the tree edit distance, as used e.g. by [10, 15].

Secondly, we used the continuous hint factory (CHF) [10] which extends the one-nearest neighbor approach by computing a weighted average of multiple close neighbors and then constructs the program which is closest to this weighted average. Since this construction occurs in the space of syntax trees, it does not come with variable or function names attached. We therefore consider two versions: For the first two tasks, we present an 'abstract' program version where all variables and functions are named 'x'. For the last task, we instead use the nearest neighbor in the training data to the weighted average.

Finally, we used the ast2vec neural network [11] to first translate the student's current program into a vector, then predict how this vector should change via linear regression, and decode this predicted vector back into a syntax tree. To provide function names as well, we trained a classifier that mapped ast2vec encodings for subtrees to typical function names in the training data and we automatically copied variable names and strings from the student's current program, as suggested by [11].

In all cases, the hint was formatted as a program which the

recipe task

You're opening a boutique pie shop. You have lots of crazy pie ideas, but you need to keep them secret! Write a program that asks for a pie idea, and encodes it as the numeric code for each letter, using the `ord` function. Print the code for each letter on a new line.

recipe submission

```
msg = input('Pie idea: ')
code = ord(msg[0])
code1 = ord(msg[1])
code2 = ord(msg[2])
code3 = ord(msg[3])
code4 = ord(msg[4])
code5 = ord(msg[5])
print(code)
print(code1)
print(code2)
print(code3)
print(code4)
print(code5)
```

recipe solution

```
msg = input('Pie idea: ')
for a in msg:
  print(ord(a))
```

anagram task

Let's make a computer program that only knows how to say 'hi': It doesn't matter what you type in, it should still print `'Hi, I am a computer'`! To make it a bit more exciting, though, we'll add an Easter Egg. The word `'anagram'` should trigger a secret message:

Hi, I am a computer!
What are you? anagram
Nag a ram!
Hi, I am a computer!

anagram submission

```
print('Hi, I am a computer!')
raining = input('What are you? ')
if raining == 'a dog':
  print('Hi, I am a computer!')
if raining == 'anagram':
  print('Nag a ram!')
  print('Hi, I am a computer!')
if raining == 'a person':
  print('Hi, I am a computer!')
```

anagram solution

```
print('Hi, I am a computer!')
word = input('What are you? ')
if word == 'anagram':
  print('Nag a ram!')
print('Hi, I am a computer!)
```

scoville task

The Scoville scale measures the spiciness of chilli peppers or other spicy foods in Scoville heat units (SHU). For example, a jalapeño has a range between 1,000 to 10,000, and a habanero is between 100,000 and 350,000! Different people have different tolerances to eating chilli peppers. Nam's parents cook with a lot of chilli, and so she enjoys eating foods with a SHU value less than 10000. Michael likes it less spicy, and only enjoys eating foods with a SHU value less than or equal to 120. Write a program to read in the SHU value for some food, and print out who will enjoy the food. For example:

What is the SHU value? 5000
Nam will enjoy this!

and another example:

What is the SHU value? 120
Michael will enjoy this!
Nam will enjoy this!

If neither Michael nor Nam will enjoy the food, your program should output: This food is too spicy for everyone :(

scoville submission

```
jalapeno = int(input("What is the SHU value? "))
if jalapeno <= 10000:
  print('Nam will enjoy this!')
else:
  print('This food is too spicy for everyone!')

sugary = int(input("What is the SHU value? "))
if sugary <= 120:
  print('Nam will enjoy this!')
  print('Michael will enjoy this!')
else:
  print('This food is too spicy for everyone!')
```

scoville solution

```
shu = int(input("What is the SHU value? "))

if shu < 10000:
  print('Nam will enjoy this!')
if shu < 120:
  print('Michael will enjoy this!')
if shu >= 10000:
  print('This food is too spicy for everyone :(')
```

Figure 1: The three program examples in our study (recipe, anagram, and scoville), each with the task description the student's received, the student submission, and a correct solution.

recipe hints

1NN

```
a = input('Pie idea: ')
b = [ord(x) for x in a]
print(*b, sep="\n")
```

CHF

```
x = x('<string>')
x = x(x[0])
x = x(x[0])
x = x(x[0])
x = x(x[0])
for x in x
x(x, keyword='<string>')
```

ast2vec

```
msg = input('Pie idea: ')
code = ord(msg[0])
code1 = ord(msg[1])
code2 = ord(msg[2])
code3 = ord(msg[3])
code4 = ord(msg[4])
print(code1)
print(code1)
```

anagram hints

1NN

```
print('Hi, I am a computer!')
computer = input('What are you? ')
if computer == 'anagram':
  print('Nag a ram!')
print('Hi, I am a computer!')
```

CHF

```
x('<string>')
x = x('<string>')
if x == '<string>':
  x('<string>')
if x == '<string>':
  x('<string>')
  x(x)
if x == '<string>':
  x('<string>')
```

ast2vec

```
raining = input('What are you? ')
if raining == 'anagram':
  print('Nag a ram!')
  print('Hi, I am a computer!')
if raining == 'a person':
  print('Hi, I am a computer!')
```

scoville hints

1NN

```
msg = int(input('What is the SHU value? '))
if msg < 10000:
  print ('Nam will enjoy this!')
else:
  print('This food is too spicy for everyone :(')
if msg <= 120:
  print ('Michael will enjoy this!')
```

CHF

```
shu = int(input('What is the SHU value? '))
if shu < 10000:
  print('Nam will enjoy this!')
if shu <= 120:
  print('Michael will enjoy this!')
else:
  print('This food is too spicy for everyone :(')
```

ast2vec

```
sugary = int(input('What is the SHU value? '))
if sugary >= 10000:
  print('Nam will enjoy this!')
if sugary < 0:
  print('What is the SHU value? ')
if sugary < 120:
  print('Nam will enjoy this!')
```

Figure 2: The hints of all three methods (1NN, CHF, and ast2vec) for all three student submissions from Figure 1.

student might try next to improve their current program. We used a random sample of 30 student traces from the same task as training data to simulate a 'cold start' with only a classroom-sized training data set.

The resulting hints of all three methods for all three tasks are shown in Figure 2.

3.4 Survey and recruitment

For our study, we recruited $N = 7$ teachers from programming courses in Australia and Germany. Recruitment was performed via e-mail lists with a survey link. Teachers could then voluntarily and anonymously complete the survey in Microsoft forms. Participants were first asked about their experience as programming teachers. Six participants responded that they had taught more than three courses, and one participant that they had taught between one and three courses. Participants with no experience were excluded from the study.

We acknowledge that our recruitment strategy has limitations: While we can be reasonably certain that only experienced programming teachers took part, we have no information about the specific courses they taught and whether that matches up with the kind of programming task we investigated in our study. Further, self-selection bias may have occurred as we did not employ a random recruitment strategy.

Next, we presented the first programming task (the recipe task) with the official task description from the National Computer Science School, the example solution, and the student's submission (refer to Figure 1, top left). We asked the teachers whether they thought the student needed help in this situation (on a four point scale), why (as a free text field), and what edit they would recommend to guide the student (free text field). Further, we presented the three automatic hints in Figure 2 (top left) and asked the teacher how relevant each hint was, how useful it was, and how much the student could learn from it, all on a five-point scale. We defined 'relevant' as 'addressing the core problem of the student's program' and 'useful' as 'getting closer to a correct solution'. These scales correspond to the scales of relevance and progress proposed by [13]. We replaced the 'interpretability' scale of [13] with 'learning' to encourage the teachers to reflect on the learning impact a hint may have.

Finally, we asked the teachers whether they would prefer *not* to give any of the hints and why (free text). We repeated all questions for the anagram and scoville tasks. To avoid ordering bias, the order of hint methods was shuffled randomly for each participant.

4. RESULTS

We present our results beginning with the teacher's assessment of whether hints were needed at all, followed by the hints given by teachers, and we conclude with the assessment of teachers for the automatic hints.

4.1 Help-need assessment

For each of the programs in Figure 1, we first asked the teachers how much help a student in this situation would need on a four-point scale, ranging from 0 ("The student is on the way to a correct solution and does not need help.") to 3 ("The student seems to have crucial misconceptions and should start from scratch.").

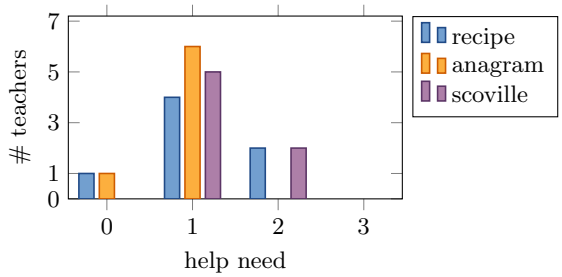

Figure 3: teacher's assessment of help-need for the three submissions from Figure 1. 0 corresponds to 'no help needed', 3 to 'the student should start from scratch'. The y axis corresponds to the number of teachers.

Figure 3 shows the distribution of responses for each task. Most teachers agreed on response 1 for all tasks ("The student is on the way to a correct solution but could benefit from a hint."). This indicates that our automatic selection indeed identified examples which indicated help-need.

We also asked teachers why they believed that the student did or did not need help. In response to this question, most teachers appeared to analyze which high-level concepts the student had already understood - judging from their program - and which concepts were still missing or were misunderstood. For example, one teacher responded for the recipe task: *"They know how to input, they know how to do the ord, they know they need to move through the string, they have just forgotten that there is a 'short cut' to do this in a loop."*, and another teacher responded for the anagram task: *"The student is not seeing the general rule of the program and is trying to cover possible cases by hand."* Further, multiple teachers responded to this question with suggestions on how to provide further guidance and help to the student, such as *"It just hasn't clicked that there are other possible inputs that they need to account for. Keeping this fresh in their mind should quickly lead to a solution."* or *"I think that they should mess around a bit more, but they should get a hint that the input function should only be used once for this problem."*.

4.2 Categories of teacher Hints

Next, we asked teachers how they would recommend editing the student's program next. Interestingly, teachers generally did not give hints on a syntactic level. In fact, some of them stated explicitly that they thought this was not helpful (e.g. "I would not give students exact syntax because then they blindly follow without understanding"). However, in some cases, they did suggest lines to delete. We found that teacher feedback tended to fit into four general categories:

A suggesting a missing concept, such as a for-loop, an else statement, or a combination of if-statements. For example, *"When you have a line of code that you are*

Table 1: teacher hint types

	teacher						
task	1	2	3	4	5	6	7
recipe	A	A	B	B	A	C,D	A,B
anagram	B	B	A	D	B	B	B
scoville	A	A	A	D	B	D	A,D

Table 2: Average teacher ratings (± std.) for each of the hints from Figure 2.

method	relevant	useful	learning
	recipe		
1NN	1.43 ± 0.73	0.86 ± 1.12	-0.14 ± 1.12
CHF	-1.71 ± 0.45	-1.86 ± 0.35	-1.86 ± 0.35
ast2vec	-1.29 ± 0.88	-1.71 ± 0.45	-1.43 ± 0.73
	anagram		
1NN	1.86 ± 0.35	1.43 ± 0.73	0.14 ± 0.99
CHF	-2.00 ± 0.00	-2.00 ± 0.00	-1.86 ± 0.35
ast2vec	-0.29 ± 1.28	-0.43 ± 0.90	-0.71 ± 1.03
	scoville		
1NN	1.14 ± 0.35	1.14 ± 0.83	0.14 ± 1.12
CHF	1.43 ± 0.49	1.57 ± 0.73	0.57 ± 1.29
ast2vec	0.29 ± 1.28	-0.29 ± 1.58	-0.29 ± 1.28

repeating it can be useful to use a loop like 'for' or 'while'. This also allows you to repeat the code within the body of the loop for a flexible number of times."

B explaining or hinting at situations when the program will not work as expected. E.g. *"You're almost there. However, it will only do the right thing when the user writes 'a dog', 'anagram' or 'a person'. You can improve it so that it says 'Hi, I am a computer!' every time, no matter what the user says."*

C suggesting the student solve a simpler problem first. E.g. *"Suggest that they delete all but the first line and try and print out each letter one at a time."*

D suggesting that the student has something unnecessary in their program, or directly telling them to delete it. E.g. *"Remove the two irrelevant if statements leaving only the correct 'easter egg' statement"*

Table 1 shows a classification of the teacher hints into these four categories. We observe that each hint could be classified in at least one category.

4.3 Assessment of automatic hints

Our third set of questions for each task concerned the rating of automatic hints (refer to Figure 2) according to relevance, usefulness, and learning, each on a five-point scale from -2 ("not at all") to +2 ("very"). Table 2 shows the average ratings (± standard deviation) given by the teachers.

We observe that 1NN is generally regarded as relevant and useful, which can be explained by the fact that it always recommended a correct solution for the tasks in our examples (refer to Figure 2 and 1). However, teachers believed that students would not learn particularly much from these hints (rating around zero). For CHF, we observe strongly negative ratings in all three criteria for the recipe and anagram task, where CHF did not provide function and variable names (refer to Figure 2), but positive scores for the scoville task where it selected a correct solution. On that task, it received even higher scores than 1NN. Ast2vec received negative scores on the recipe task, and scores around zero for all criteria on all other tasks.

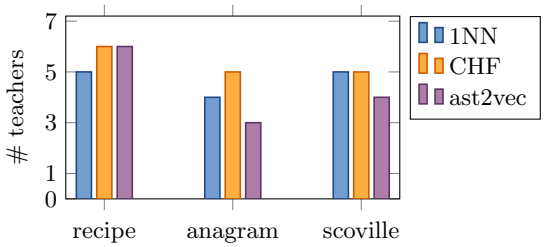

Figure 4: The number of teachers (y-axis) who would prefer *not* to give a certain hint according to hint method (color) and task (x-axis).

Finally, we asked teachers if there were any hints they would prefer *not* to give, and why. Figure 4 shows how often each method was named for each task (where lower is better). Regarding the reasoning, one teacher always responded that *"I would not give students exact syntax because then they blindly follow without understanding."*, which excludes all automatic hints. Further, 1NN was often named because it *"shows a valid solution. Students may copy the hint code and then use it - without understanding what it does."*. The hints provided by CHF for the first two tasks did not include variable and function names (refer to Figure 2), which lead teachers to exclude it because it *"does not look syntactically valid, and is incomprehensible."*. Ast2vec was named least often, albeit by a narrow margin. Reasons for naming it were that it is *"not helpful for developing student understanding"*, would require additional explanation, or even *"harm the learning of the student."*

5. DISCUSSION

In this section, we interpret the results in light of our three research questions: Do ratings differ between methods? Do automatic hints align with teacher hints? And what are teachers' reasons for preferring some hints over others?

RQ1: Do teacher's ratings differ between hint methods?. We do observe systematic differences between hint methods in terms of ratings. However, these differences appear to be driven less by the underlying algorithm, but rather by two factors: a) whether a correct solution was selected or a partial solution, and b) whether the hint was presented as a program with function and variable names or not. teachers only gave high scores for usefulness and relevance if a correct solution was given and gave very low scores if function and variable names were missing. They

Table 3: Abstracted hints based on the four hint types from Table 1.

	method	abstracted hint
recipe hints		
A	1NN, CHF	Maybe you could try a for-loop.
B	1NN	What happens when the user types 'apple'?
C	1NN, CHF	Try doing this task on for-loops first
D	1NN, CHF, ast2vec	Can you think of a way to reduce the number of print statements?
anagram hints		
B	1NN	What happens when the user types 'cat'?
D	CHF, ast2vec	Can you think of a way to use fewer if statements?
scoville hints		
B	1NN, CHF	What happens when the user types 120?
D	1NN, CHF, ast2vec	How can you reduce the number of input calls? (and/or) Try using only one variable.

never gave high scores for learning.

RQ2: Do automatic hints align with teacher hints?. We observe that teacher hints do not directly align with automatic hints because teachers generally suggested hints which were higher-level, like adding missing concepts or deleting lines to get to a more compact program. More generally, we identified four categories (refer to Section 4.2) which cover the kinds of edits teachers would have given themselves. To align teacher hints to automatic hints, we could employ automatic heuristics which post-process next-step hints, such as:

A - Missing Concepts Compare the nodes of the student's current AST to that of the next step, then suggest concepts corresponding to missing nodes.

B - Mishandled Situations Apply test cases to the current and the predicted program, then suggest the student focus on inputs that work for the next step but not the current step

C - Simpler Problem Similarly to A, first identify missing concepts in the student's work, then suggest easier programming tasks that contain this concept (e.g. from earlier in the course).

D - Deletions Compare the current step to the next step and, if lines are deleted, ask the student if these lines are necessary.

Applying these strategies, the automated hints for the tasks in Figure 2 might then become the hints shown in Table 3, which align better with the hints given by teachers.

We note in passing that abstraction may also make it easier to provide helpful hints because the hint method does not need to get every detail (such as function or variable names) right, merely the rough direction. For example, we notice that ast2vec uses the wrong string in line 5 of its hint and the wrong comparison constant in line 4 of Figure 2 (bottom). This would not be an issue in the abstracted hint.

Still, we acknowledge that this approach has limitations: For strategy A and C, we implicitly assume that 'concepts' coincide with syntactic building blocks, e.g.: 'have you tried adding a for loop?'. This assumption likely breaks down in more advanced programming classes.

RQ3: What are teachers' reasons for preferring some hints over others?. Teachers mostly explained hints by concepts that were still missing in a program (like loops), undesired functional behavior for additional cases, or superfluous code. These reasons align with the hints the teachers gave. Importantly, many teachers also emphasized that the student had already gotten many things right, indicating that teachers were motivated to preserve the progress that had already been made. The main reason for *not* providing a hint appeared to be that the teachers were concerned that the hint may not lead to any learning, either because the hint was syntactic, and hence not abstract enough, because the hint was a correct solution, or because the hint was 'incomprehensible' due to missing function or variable names. Overall, the reasons provided by teachers underline our finding that teachers do not only care about the content of a next-step hint but also – perhaps mainly – how it is communicated, i.e. with a high-level explanation instead of a syntactic edit and without revealing the solution.

6. CONCLUSION

We performed a survey with $N = 7$ teachers to evaluate hints from three hint methods (1NN, CHF, and ast2vec) to investigate three research questions: Do quantitative ratings differ between methods? Do automatic hints align with teacher hints? And what are teachers' reasons for preferring some hints over others?

We found that teachers generally had a low opinion of syntactic next-step hints, irrespective of the method. Differences in ratings could be explained by two factors: Whether the hint was a correct solution (then it was regarded relevant and useful) or not (then all ratings were around zero) and whether the hint used human-readable variable and function names or not (then the ratings became strongly negative).

Instead of syntactic hints, teachers preferred higher-level hints which suggested a missing concept, pointed out inputs which were not appropriately covered by the current program, suggested a simpler problem first, or proposed to remove superfluous lines.

Finally, the main concern of teachers for not giving hints was students' learning. They disregarded both syntactic hints as well as showing a correct solution because they were concerned that students might naïvely apply the hint without reflecting on it sufficiently. This points to a potential gap in current next-step hint approaches, which are mainly focused

on suggesting changes, but less on inviting reflection or abstracting to a higher level. For introductory programming courses, it may be sufficient to just post-process syntax-level hints with simple heuristics – such as the ones proposed in the previous section. Additionally, one can introduce textual explanations as suggested by [8]. However, further research is needed to investigate whether it is sufficient to change how next-step hints are communicated or whether deeper changes in hint methods are necessary to achieve a better alignment with the pedagogical expertise of programming teachers. Finally, an evaluation study with students is still required to make sure that any refined hint strategy yields better student outcomes compared to current hint strategies.

7. ACKNOWLEDGMENTS

Funding by the German Research Foundation (DFG) under grant number PA 3460/2-1 is gratefully acknowledged.

8. REFERENCES

[1] T. Barnes and J. Stamper. Toward automatic hint generation for logic proof tutoring using historical student data. In B. P. Woolf, E. Aïmeur, R. Nkambou, and S. Lajoie, editors, *Proceedings of the International Conference on Intelligent Tutoring Systems (ITS 2008)*, pages 373–382, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[2] R. R. Choudhury, H. Yin, and A. Fox. Scale-driven automatic hint generation for coding style. In A. Micarelli, J. Stamper, and K. Panourgia, editors, *Intelligent Tutoring Systems*, pages 122–132, Cham, 2016. Springer International Publishing.

[3] S. Chow, K. Yacef, I. Koprinska, and J. Curran. Automated data-driven hints for computer programming students. In *Adjunct Publication of the 25th Conference on User Modeling, Adaptation and Personalization (UMAP 2017)*, page 5–10, 2017.

[4] A. T. Corbett and J. R. Anderson. Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, page 245–252, 2001.

[5] D. Fossati, B. Di Eugenio, S. Ohlsson, C. Brown, and L. Chen. Data driven automatic feedback generation in the ilist intelligent tutoring system. *Technology Instruction, Cognition and Learning*, 10(1):5–26, 2015.

[6] S. Gross and N. Pinkwart. How do learners behave in help-seeking when given a choice? In C. Conati, N. Heffernan, A. Mitrovic, and M. F. Verdejo, editors, *Proceedings of the 17th International Conference on Artificial Intelligence in Education (AIED 2015)*, pages 600–603, 2015.

[7] M. Maniktala, C. Cody, A. Isvik, N. Lytle, M. Chi, T. Barnes, et al. Extending the hint factory for the assistance dilemma: A novel, data-driven helpneed predictor for proactive problem-solving help. *Journal of Educational Data Mining*, 12(4):24–65, 2020.

[8] S. Marwan, N. Lytle, J. J. Williams, and T. Price. The impact of adding textual explanations to next-step hints in a novice programming environment. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, page 520–526, New York, NY, USA, 2019. Association for Computing Machinery.

[9] J. McBroom, B. Paassen, B. Jeffries, I. Koprinska, and K. Yacef. Progress networks as a tool for analysing student programming difficulties. In C. Szabo and J. Sheard, editors, *Proceedings of the Twenty-Third Australasian Computing Education Conference (ACE '21)*, page 158–167. Association for Computing Machinery, 2021.

[10] B. Paaßen, B. Hammer, T. Price, T. Barnes, S. Gross, and N. Pinkwart. The continuous hint factory - providing hints in vast and sparsely populated edit distance spaces. *Journal of Educational Datamining*, 10(1):1–35, 2018.

[11] B. Paaßen, J. McBroom, B. Jeffries, I. Koprinska, and K. Yacef. ast2vec: Utilizing recursive neural encodings of python programs. *Journal of Educational Datamining*, 2021. in press.

[12] C. Piech, M. Sahami, J. Huang, and L. Guibas. Autonomously generating hints by inferring problem solving policies. In G. Kiczales, D. Russell, and B. Woolf, editors, *Proceedings of the Second ACM Conference on Learning @ Scale (L@S 2015)*, page 195–204, 2015.

[13] T. Price, R. Zhi, and T. Barnes. Evaluation of a data-driven feedback algorithm for open-ended programming. In X. Hu, T. Barnes, and P. Inventado, editors, *Proceedings of the 10th International Conference on Educational Data Mining (EDM 2017)*, pages 192–197, 2017.

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

[15] 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*, 27(1):37–64, 2017.