Progress Networks as a Tool for Analysing Student Programming Difficulties

Jessica McBroom
The University of Sydney
Sydney, NSW, Australia
jmcb6755@uni.sydney.edu.au

Benjamin Paaßen
The University of Sydney
Sydney, NSW, Australia
benjamin.paassen@sydney.edu.au

Bryn Jeffries
Grok Learning
Sydney, NSW, Australia
bryn@groklearning.com

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

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

ABSTRACT

The behavior of students during completion of a learning task can give crucial insights into typical misconceptions as well as issues with the task design. However, analysing the detailed trace of every individual student is time-consuming and infeasible for large-scale classes. In this paper, we propose *progress networks* as an analytical tool to make sense of student data and demonstrate the technique in large-scale online learning environments for computer programming. These networks, which are easily interpreted by teachers, summarise the progression of a student population through a learning task in a single diagram and, importantly, highlight locations where students fail to make progress. Using data from three different programming courses (N > 4000), we provide instructive examples of how to apply progress networks, including how to zoom in on areas of interest to identify reasons for student difficulty. In addition, we propose a simple technique for comparing progress networks across different cohorts of interest, for instance to analyse learning differences between older and younger students, and to investigate learning retention across tasks on the same programming concept. Finally, we discuss options to improve instructional design based on the insights from progress networks, and show that progress networks can also apply to smaller cohorts.

CCS CONCEPTS

• **Applied computing** → **Education**; • **Social and professional topics** → *Age*.

KEYWORDS

interaction networks, behaviour visualisation, student mistakes, student errors, programming education, course evaluation, age differences, paired exercises

ACM Reference Format:
Jessica McBroom, Benjamin Paaßen, Bryn Jeffries, Irena Koprinska, and Kalina Yacef. 2021. Progress Networks as a Tool for Analysing Student Programming Difficulties. In *Australasian Computing Education Conference (ACE '21), February 2–4, 2021, Virtual, SA, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3441636.3442366

1 INTRODUCTION

In order to maximise the effectiveness of programming courses, it is essential to develop techniques for identifying and understanding common issues that arise during learning. Previous work has mostly focused on misconceptions that generalise across courses and can thus be taken into account when designing new learning environments [14, 15, 21]. However, students may also experience difficulty in making progress due to the learning context, e.g. due to design problems in the learning environment, insufficient material, inappropriate feedback, or misalignment in attention between instructor and student [18, 19]. Analysing the data of students using a learning environment is a logical way to gain insight into such environment-specific learning difficulties. However, the sheer amount of student data makes it difficult to perform a detailed analysis that is still representative of the general student population (for example, even in a class of 20 students with only five steps per learning task and ten tasks, detailed analysis of all 1000 steps can be infeasible).

In this paper, we build upon prior work on interaction networks [1, 10, 12, 20] and introduce *progress networks* as a concise summary of the progression of a student population through a learning task. A progress network represents task-specific levels of progress as nodes and represents student transitions between these levels of progress as edges. We demonstrate the utility of progress networks by constructing example networks for programming tasks in a large-scale online learning environment for introductory computer programming. In particular, we use test cases as a measure of progress, show the tests at which students got stuck most often, and inspect program examples to investigate why this might have been the case.

Additionally, we provide a method to compare progress networks between subgroups of students and between related learning tasks, and demonstrate this method through two example applications. In particular, we compare younger and older students on a single task and also an entire population of students across two consecutive tasks. These analyses demonstrate how progress networks can help

ACE '21, February 2–4, 2021, Virtual, SA, Australia

ACM ISBN 978-1-4503-8976-1/21/02…$15.00
https://doi.org/10.1145/3441636.3442366

to support equitable course development by focusing teacher attention on key areas, and also provide insight into learning differences between student groups.

In summary, our contributions are as follows: (i) extending the concept of interaction networks to *progress networks*, (ii) providing a method to compare progress networks across groups and tasks, and (ii) providing examples of analyses using such networks on a large-scale programming education data set.

This paper is set out as follows: Section 2 discusses the background and related work. Section 3 describes the process of constructing, automatically scoring and analysing progress networks, as well as a method for comparing them. Next, Section 4 provides four example applications of progress networks on data from three online programming courses, showing how progress networks can provide insight into student behaviour and reveal potential places for course development. Finally, Section 5 investigates the impact of cohort size on progress networks and Section 6 concludes with a summary of the main ideas and suggestions for future work.

2 BACKGROUND AND RELATED WORK

Misconceptions in computer programming education: The review of [15] suggests that students' misconceptions in computer programming can be roughly grouped into syntactic, conceptual, and strategic misconceptions. Syntactic problems are generally easy to detect and fix, whereas conceptual and strategic misconceptions are more challenging. Swidan at al. [21] found that misconceptions are more common for younger students and students who only use block-based environments, compared to older students and students who program in more than one environment. Qian at al. [14] asked programming teachers about frequency and importance of misconceptions and found that some misconceptions may be difficult to detect due to their latent nature. Multiple works have attempted to automatically detect misconceptions that are frequent, e.g. by means of clustering [5–7, 10, 22] or code indexing strategies [11]. This has the advantage that teacher instruction for a misconception can be scaled easily by simply copying it whenever another instance of a certain misconception has been detected. Our work is similar in that we cluster student solutions as well, but we use test cases to construct the clusters and focus on the movement between clusters rather than the static analysis of cluster membership.

Automatic test cases in computer programming: An automatic test case consists of a pair of an example input and an expected output for a program. The input is fed into the student's current submission and the program's output is compared to the expected output. If both are equal, we define the test case as *passed*. Otherwise, we define it as *failed*. For example, if the student's program should add two numbers, an example test case could be the input $(3, 4)$ and the expected output 7. While a test case can never verify functionality for *all* possible inputs[1], failing a test case can be seen as an indication of a misconception. Indeed, [15] suggests that the failure to properly test their own code is among the common strategic misconceptions of novice programmers. Automatic test cases are also appealing because they provide an intuitive specification of what a program *should do*, inspiring students to think about their program's functionality rather than syntactic details [3]. Additionally, test cases are relatively fast to write and to execute, making them a popular tool in learning environments for programming to this day [8]. For example [17] select automatic hints for students in such a way that the hint is likely to guide students to more completed test cases. In this work, we apply test cases as well. However, we additionally consider them as ordered according to a teacher's notion of progress.

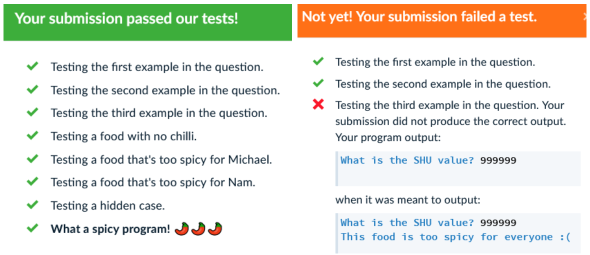
Interaction Networks: Our work is inspired by interaction networks [1, 10, 12, 20], which summarise student behaviour while completing a task. To construct an interaction network, we first record student behaviour in the form of traces, where each trace is a list of successive states. For example, for a simple 'Hello world' Python programming task, a student may start with the empty program, then write a print statement, and finally add the string `'Hello world!'`, which yields the trace “” (empty program), “`print()`”, and “`print('Hello world!')`”. Once such traces are collected for all students in a population, they are merged into an interaction network by representing all distinct states as nodes of the network and all transitions between adjacent programs as edges. Additionally, edges are labeled by the frequency of those transitions in the trace data.

A common challenge of interaction networks is that of *state matching* [1]. For the example of computer programming, we observe that students can write functionally equivalent programs in an infinity of different ways, which we might still want to recognise as representing the same state. Past research has tried to address this problem via *canonicalisation* techniques, which abstract from a student's program to a form that is easier to match, e.g. by using the abstract syntax tree instead of the source code, removing comments, and normalising variable names [16]. Another challenge is that there is no direct relationship between a state and the progress it represents. In other words, if we observe a certain student trace, it is not immediately clear whether the student's motion represents steady progress towards the correct solution or, rather, a failure to progress. Following prior work of [17], we argue that test cases offer a straightforward way to address both issues. We canonicalise programs by representing them in terms of the test cases they completed, which reduces the node set of an interaction network dramatically and solves the state matching problem. Additionally, we can let a domain expert order the test cases according to a notion of progress, starting from an initial program that fails all tests to a fully functional program that passes all tests. This notion of progress is the reason we change the name to *progress networks*.

3 METHOD: CONSTRUCTING, SELECTING AND ANALYSING PROGRESS NETWORKS

Our motivation for analysing student programming behaviour is to identify typical situations where students fail to make progress and to find reasons for such failures. Our analytical tool for this purpose (i.e. the *progress network*) is a network that summarises the progression of a student population through a learning task. In particular, for each student we record a trace of their solution states while working on the task, then map each of the states to a level of progress, resulting in a trace of progress levels, and finally summarise these progress traces in a network, where progress levels

[1] Such a general proof of semantic equality is undecidable [4].



Your submission passed our tests!

- ✓ Testing the first example in the question.
- ✓ Testing the second example in the question.
- ✓ Testing the third example in the question.
- ✓ Testing a food with no chilli.
- ✓ Testing a food that's too spicy for Michael.
- ✓ Testing a food that's too spicy for Nam.
- ✓ Testing a hidden case.
- ✓ **What a spicy program! 🌶🌶🌶**

Not yet! Your submission failed a test.

- ✓ Testing the first example in the question.
- ✓ Testing the second example in the question.
- ✗ Testing the third example in the question. Your submission did not produce the correct output. Your program output:

```
What is the SHU value? 999999
```

when it was meant to output:

```
What is the SHU value? 999999
This food is too spicy for everyone :(
```

Figure 1: Two examples of automated test feedback for a passing program (left) and a failing program (right). Note that the tests are run in order, so only feedback on the first failed test is given.

are nodes and edges indicate how many students moved from one level to another.

To construct a progress network we require two ingredients: student trace data and a mapping from student states to progress levels. In this paper, we focus specifically on the case of a large-scale learning environment for computer programming. In this case, our student traces consist of partially completed programs. Our mapping from programs to progress levels is provided by test cases. In particular, each learning task is associated with a list of test cases ordered by a teacher according to progress. Whenever a student submits a partially completed program, each test case in the list is run sequentially. As soon as one of them fails, testing halts and reports the current failure to the student, as shown in Figure 1. Accordingly, a natural measure of progress is the number of passed test cases.

Once student traces are converted into progress traces, constructing the progress network follows the same scheme as interaction networks [1]. In particular, we generate one node for each test case, and three special nodes for 'start', 'end', and 'pass'. Then, we iterate over all traces in our data set and consider, for each program in the trace, the test case where the program failed first. If all tests passed, we assign the 'pass' node. This yields a trace of nodes in our network, which we augment by putting the 'start' node at the start and the 'end' node at the end. Then, we draw an edge in our network between each node and its successor in the trace. If multiple students use the same edge, we draw it thicker and label it with the number of such students. As an example, consider Figure 2. The top shows three traces which have already been converted to progress traces. Student 1 only made a single submission which failed at test 1 and then quit; student 2 submitted a program that failed test 2, then changed the program but still failed test 2, and finally submitted a program which passed all tests; and student 3 first had a program which failed test 1 but could correct it to a program that passed all tests. These three traces are summarised in the network shown at the bottom. For each transition between tests that occurs in the trace data, we observe an edge in the network, where thicker edges correspond to edges that were used by more students (in Section 4, the specific number of uses for each edge is also labeled).

Once a progress network is constructed, we can pose our research questions more precisely, namely:

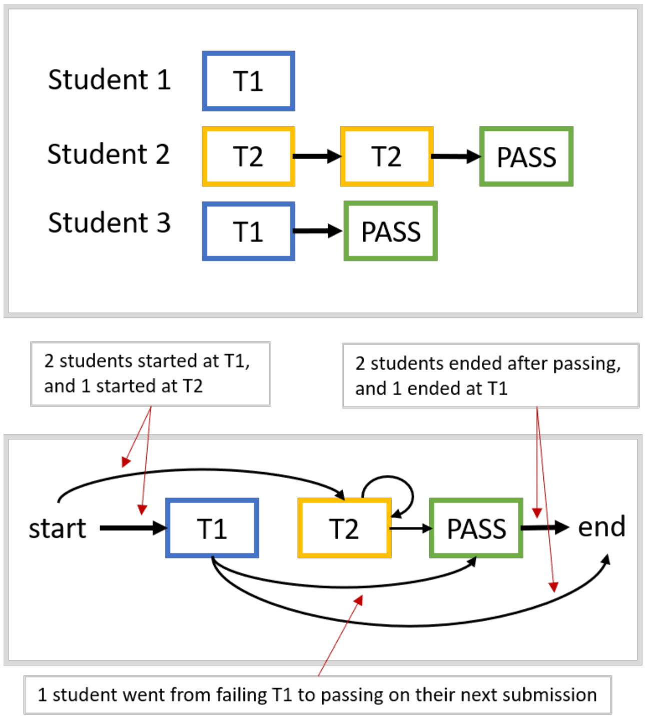

Student 1
T1
Student 2
T2
T2
PASS
Student 3
T1
PASS
2 students started at T1, and 1 started at T2
2 students ended after passing, and 1 ended at T1
start
T1
T2
PASS
end
1 student went from failing T1 to passing on their next submission

Figure 2: An example of a progress network construction. The traces (top) show the progressions of individual students through an exercise. The network (bottom) summarises the progress of all students on the exercise.

(1) On which tasks do students face particular difficulty to make progress?
(2) At which locations in the tasks do students have this difficulty?
(3) Are there differences in these locations across groups?
(4) Are there differences in these locations across tasks?

We first address questions one and two, and then move on to questions three and four.

3.1 Identifying Key Exercises and Difficult Steps

Since programming courses often involve many programming tasks and teachers may not have time to analyse every task, one important challenge is identifying the key exercises to focus analysis on. As such, it is useful to have an automated method of ranking progress networks so that the most important networks are considered first.

We propose the following simple measure for scoring progress networks, where *back steps* are steps that cause students to fail *more* tests than they did in the previous submission and *repeat steps* are steps where students fail the same number of tests:

$$score(x) = back\_steps(x) + repeat\_steps(x)$$

Note that we collectively refer to back and repeat steps as *non-improving steps*. We chose this measure for two key reasons:

(1) Firstly, back steps and same steps are a useful indication of difficulty, because they are caused by students repeatedly failing the same test or regressing to an earlier failure. This can be seen as an instance of unproductive struggle or wheel-spinning [2] on the student's path toward a solution, which we would like to avoid if possible.

(2) Secondly, this score takes into account both the *number* of students completing an exercise, and the *average mistakes* per student, which can be seen mathematically as follows: $score(x) = \frac{back\_steps(x)+repeat\_steps(x)}{num\_students(x)} \times num\_students(x)$. This means that exercises with a high score will either have a large number of students or a very high error rate, so improving these exercises will have the largest absolute impact.

We prefer this measure to other simple measures, such as the number of steps overall, since we still regard it as progress if a student fails a test but then is able to correct their program to not fail the same test again. We also note that this scoring method utilises the key properties of progress networks, since it requires a dynamic understanding of student progress through a task that cannot be captured by only analysing a student's current state.

Once the most interesting exercises are identified, teachers can then analyse these exercises using progress networks to gain insight into student behaviour. Section 4.2 shows two example applications of this process.

3.2 Comparing Networks

To analyse differences between two progress networks we employ a simple three-step procedure. First, we normalise the edge counts in both networks by the total number of submissions. Second, we match the nodes between both networks, e.g. based on semantic equivalence. Third, we compute the difference in normalised edge counts. This yields a new progress network where edges are labeled with difference values, where positive values (blue) represent edges that occur more frequently in the first network, and negative values (orange) represent edges that occur more frequently in the second network. Consider the example shown in Figure 3. Here, we compare two networks where the second network has a frequent edge from *start* to *pass* which is entirely absent from the first network. Accordingly, if we compute the difference in edge strength between both networks, this edge emerges as the most prominent difference. This indicates that students in network 2 were faster in progressing to the *pass* state compared to students in network 1. This comparison can provide insight into more specific behavioural differences than existing techniques such as Markov models [9].

In Section 4.3, we provide two example applications of this comparison technique to investigate 1) differences between younger and older students on the same task and 2) differences across two consecutive but similar tasks for a single group of students.

4 RESULTS AND DISCUSSION

4.1 Data

In this section, we apply progress networks to data from three Australian Python programming courses for school students, offered online through the Grok Learning platform[2]. Each of these courses involved a series of programming exercises interleaved with notes on programming concepts, and students could submit their solutions to receive automated feedback from test cases.

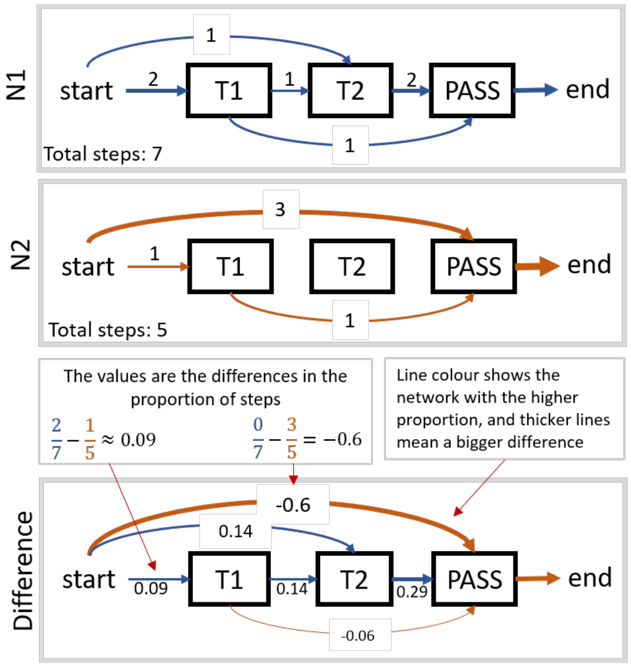

N1
start
2
T1
1
T2
2
PASS
end
1
1
Total steps: 7
N2
start
1
T1
T2
PASS
end
3
1
Total steps: 5
The values are the differences in the proportion of steps
$\frac{2}{7} - \frac{1}{5} \approx 0.09$
$\frac{0}{7} - \frac{3}{5} = -0.6$
Line colour shows the network with the higher proportion, and thicker lines mean a bigger difference
Difference
start
0.09
T1
0.14
T2
0.29
PASS
end
-0.6
0.14
-0.06

Figure 3: An example for the comparison of two progress networks. Top: Two network examples. Bottom: The difference between networks, where blue indicates that the edge was more frequent in network 1 and orange indicates that the edge was more frequent in network 2. Thickness corresponds to the amount of difference. The largest difference lies in the edge from start to pass, which is much more prominent in network 2 compared to network 1.

The first course, *DT Challenge Python - Chatbot*, is a project-based course comprising 62 beginner exercises of a similar structure, and teaches students the skills to program a simple chatbot. Our data set includes student enrolments from May 2017 to August 2020, and we apply progress networks to a task attempted by N = 23,381 students over this time. The other courses, *NCSS Challenge (Beginners) 2019* and *NCSS Challenge (Intermediate) 2019* were shorter courses offered as part of the 2019 National Computer Science School (NCSS) Challenge [3], a 5-week challenge held from July-August in 2019. For these courses, we analyse data from N = 5994 and N = 4123 students respectively. Note that some students completed these courses as part of a school class, while others were participated individually.

4.2 Insights from Example Networks

In this section, we apply progress networks to two example exercises from the *DT Challenge Python - Chatbot* and *NCSS Challenge (Intermediate)* courses. In each case, the exercises were selected using the scoring technique described in Section 3.

[2] https://groklearning.com

[3] https://groklearning.com/challenge/

4.2.1 Example 1 - Chatbot: Echo! Echo! Figure 4 shows the progress network for students attempting the fifth exercise (*Echo! Echo!*) of the *DT Python Challenge - Chatbot* course. This exercise, which required students to print input from the user, received a score of 36,450 under our system (i.e. 32, 092 back steps + 4, 358 repeat steps), which was the highest for the course. The network summarises the behaviour of 23,381 students who attempted the exercise.

The problem requires the student's code to prompt the user for a message, and then print this message back. The first four tests applied to the code check that that the input prompt contains the correct words (T1), punctuation (T2), case (T3) and whitespace (T4). The remaining tests check the output is as expected for different inputs: `'Echo!'` (T5); `'I am having a great day!'` (T6); and for a final "hidden" case where the input is not disclosed to the user (T7), to ensure generality.

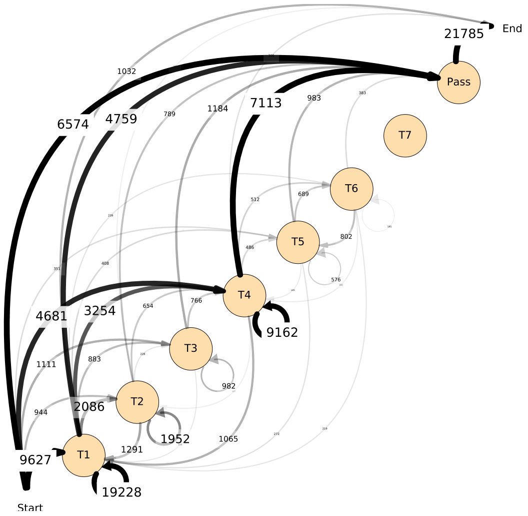
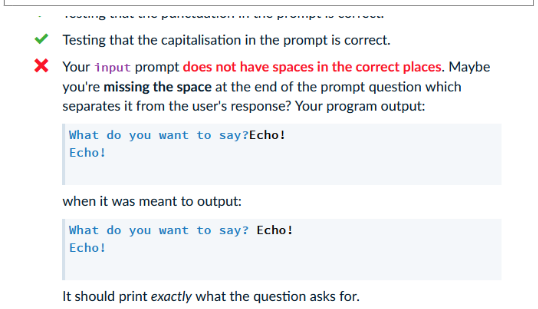


Figure 4: The progress network for 23,381 students attempting the exercise *Echo! Echo!*.

From this network, we can see that T1 and T4 are particularly interesting test cases. This is because a large number of students fail these tests on their first submission (i.e. 9627 or 41% for T1 and 4681 or 20% for T4) and many students who have already failed these tests fail them again on their next submission (this occurs 19,228 and 9162 times respectively). In addition, the most common places for students to give up during this exercise are after failing these tests (this happens for 1032 and 300 students respectively) and many students cycle between failing these tests (T4 was failed after T1 3254 times and T1 was failed after T4 1065 times). This indicates that students often find this test difficult, even after receiving feedback from the system.

After identifying interesting places in the network, we can use samples of student work to gain insight into why they may have had difficulty here. For example, to see why many students failed T4 repeatedly, we can look at an example of a student who submitted two programs in a row that failed T4 and the feedback they received, as shown in Figure 5. In this case, the problem with the student's first program was that they did not have a space after the question mark in the prompt (i.e. the prompt should have been `"What do you want to say? "`). However, they seem to have misinterpreted the feedback and instead removed the space on line 2, resulting in a failure of the same test on their next submission.

```
1   msg = input('What do you want to say?')
2
3   print(msg)
```



Testing that the punctuation in the prompt is correct.

✓ Testing that the capitalisation in the prompt is correct.

✗ Your `input` prompt **does not have spaces in the correct places**. Maybe you're **missing the space** at the end of the prompt question which separates it from the user's response? Your program output:

```
What do you want to say?Echo!
Echo!
```

when it was meant to output:

```
What do you want to say? Echo!
Echo!
```

It should print *exactly* what the question asks for.

```
1   msg = input('What do you want to say?')
2   print(msg)
```

Figure 5: An example of a pair of consecutive programs submitted by a student that both failed T4. Also shown is the automatic feedback provided to the student after the first submission.

This is a particularly interesting example, because the feedback explicitly identifies the problem ("Maybe you're missing the space at the end of the prompt question ... ?"), but the student does not respond as expected. One potential reason for this could be that the student did not understand the feedback. In this case, course designers could try rewording the feedback to improve clarity. Alternatively, another reason could be that the student did not read this feedback, perhaps because their attention was drawn to the general message in red about incorrect spacing. In this case, perhaps shortening the feedback, or changing which parts are emphasised could help to address this issue.

Figure 6 shows another example of a student who failed T4, but where their next program failed T1 instead of T4. From the student's first submission, it is likely that they were unsure how to print user input on line 2. In their subsequent submission, they try to correct line 2, but end up introducing an error that causes T1 to fail.

```
1   name=input("What do you want to say?")
2   Input=("Echo!")
```

```
1   name=input("What do you want to say?")
2   print(speech)
```

Figure 6: An example of a pair of consecutive programs submitted by a student that failed T4 then T1.

This example highlights a potential opportunity to improve the feedback system. When this student submitted the first program, they received feedback similar to that in Figure 5, which suggested they fix the whitespace in the prompt. Since this was not the core issue with the student's program, it may have been helpful to reorder the tests to check the printing was done correctly first. In this way, the student could receive feedback on the main issue, which may have helped them to correct it more easily.

Figure 7 shows an example of a student who repeatedly failed T1. Their programs failed the test because they were missing the word "to" in their input prompt. In each case, they were given the feedback "Your input prompt does not use the correct words.", followed by their program's output and the expected output. Looking at their code, the student did not respond to this feedback as expected: first, they attempted to swap the `"` for `'`. Then, after failing the test again, they removed the (correct) space after the question mark. This suggests that they had trouble identifying the incorrect word and so were trying other things. To improve this feedback, perhaps explicitly mentioning which word was missing, or more strongly highlighting the differences between the actual and expected output could help to address this issue.

```python
Echo = input("What do you want say? ")
print(Echo)
```

```python
Echo = input('What do you want say? ')
print(Echo)
```

```python
Echo = input('What do you want say?')
print(Echo)
```

Figure 7: An example of three consecutive programs submitted by a student that all failed T1.

In summary, this example shows how we can use a progress network to visualise the behaviour of thousands of students at once and thereby identify the areas where they experience the most difficulty. We can then "zoom in" on these areas by analysing samples of student programs and feedback that correspond to them to reveal meaningful and practical steps for improving the exercise.

4.2.2 Example 2 - Intermediate - Scoville Scale. As a second example, we consider an exercise from the *intermediate* course of the 2019 NCSS Challenge. This course acts as an interesting second example because it not only involves more advanced programming concepts, but it also had a time limit of five weeks for completion, which may have prompted different numbers and types of mistakes.

Figure 8 shows the progress network for the fourth task of the course, *Scoville Scale*, which was attempted by 4123 students and scored highest with 564 back steps and 1738 repeat steps. This exercise involved writing a program to ask for the spiciness level of a food (i.e. its SHU value) then printing who would like the food based on this score. The student therefore needed to implement conditional logic (using, e.g., `if ... else`) with two comparisons. All seven tests checked the input prompt and returned message for a variety of SHU values. Tests T1 to T3 intentionally covered the three expected code branches. Tests T4 to T6 tested edge cases where a student might have incorrectly specified the required inequality tests, while T7 used a hidden input value for generality.

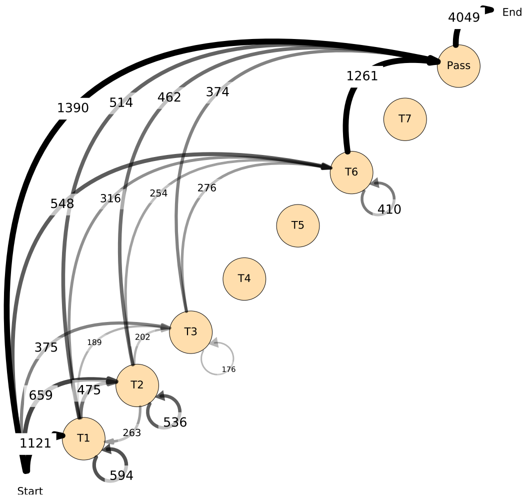


Figure 8: The progress network for the *Scoville Scale* task of the 2019 intermediate programming challenge.

From the network, the most interesting test cases appear to be T1, T2 and T6 since many students repeatedly failed them (there were 594, 536 and 410 cases respectively). In addition, T2 is particularly interesting because roughly half (263) of all back steps start at T2.

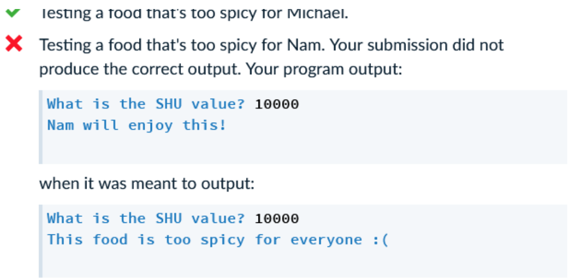
To better understand why these test cases pose challenges to students, we begin by considering examples where students repeatedly failed the same test. Figure 9 shows an example of a student who failed T6 twice in a row. Note that the first program is correct except the `100000` on lines 2 and 6 should have been `10000` (i.e. with one less 0). After receiving feedback that their program failed T6 (which tests the input 10000), they did not notice this extra 0 and instead assumed the problem was with the boundaries in their if statements. As such, they changed these boundaries, which moved them further away from the correct solution.

Since this student had difficulty distinguishing between 10000 and 100000, one possible way to minimise this issue in the future could be to use spacing or a comma in the task description (i.e. instead of "Nam... enjoys eating foods with a SHU value less than `10000`.", the 10000 could be written as 10 000 or 10,000). Another option could be to swap the question for an equivalent one with numbers that are less easily misread, or to add a check for this kind of issue when testing.

Moving on to T2, Figure 10 shows an example of a student who failed this test on consecutive submissions. Note that the first program is almost correct except that `<` on line 4 should be `<=` and `>` on line 6 should be `>=`. Due to these errors, when T4 tested input on a boundary condition (i.e. 120) the program produced incorrect output. Without realising the real issue, the student attempted to correct their program by switching the order of the statements, which resulted in the test failing again (note that the task description explicitly states that the order of printing does not matter). A similar analysis of T1 → T1 mistakes shows that students also respond to feedback here by switching the order of statements without changing the output in a functionally relevant way.

These findings point to a typical conceptual misconception [15], namely to assume that all ordering changes influence code behaviour, which is true in many, but not all cases. As such, one

```
shuv = int(input('What is the SHU value? '))
if shuv < 100000:
  print('Nam will enjoy this!')
if shuv <= 120:
  print('Michael will enjoy this!')
if shuv >= 100000:
  print('This food is too spicy for everyone :(')
```



✓ Testing a food that's too spicy for Michael.

✗ Testing a food that's too spicy for Nam. Your submission did not produce the correct output. Your program output:

```
What is the SHU value? 10000
Nam will enjoy this!
```

when it was meant to output:

```
What is the SHU value? 10000
This food is too spicy for everyone :(
```

```
shuv = int(input('What is the SHU value? '))
if shuv <= 100000:
  print('Nam will enjoy this!')
if shuv <= 120:
  print('Michael will enjoy this!')
if shuv > 100000:
  print('This food is too spicy for everyone :(')
```

Figure 9: An example of a pair of consecutive programs submitted by a student that both failed T6.

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

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

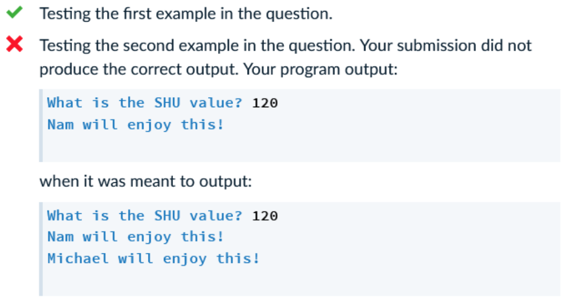
Figure 10: An example of a pair of consecutive programs submitted by a student that both failed T2.

potential opportunity for improving the course could be to dedicate more time to discussing the relationship between statement order and program output before students attempt this exercise.

At this point, we can also observe another potential avenue for improving the exercise. To see this, notice that some tests are hardly ever failed according to the progress network in Figure 8. For example, T4 is failed so infrequently that it is not drawn in the figure. On the other hand, T2 is failed very frequently. Interestingly, these tests both expect the same output, but T2 checks a more complicated input (i.e. a boundary condition). Considering this, it would make sense to switch the order of T4 and T2 so that T4 checks the easier case first, and then T2 checks the more complicated boundary case later. This could help to reduce student confusion and also allow the feedback to be better-tailored to the student's specific situation.

Figure 11 shows an example of a student who first failed T2 then failed T1 on the next submission. Note that T1 checks the input 5000, which should yield the output `Nam will enjoy this!`, whereas T2 checks the input 120, which should yield the output lines '`Nam will enjoy this!`' and '`Michael will enjoy this!`' in either order. The first program passes the former but not the latter test because once the first `if` condition applies, the second one is not checked anymore due to the `elif` construct. In the second program, the student has found this problem and corrected it, but failed to anticipate that the `else` branch would now apply for any input above 120, such that now the first test fails with the unexpected output '`Nam will enjoy this!/This food is too spicy for everyone :(`'. This can be seen as a 'productive' mistake, in the sense that it can lead to a deeper understanding of the `if`, `elif`, and `else` constructs in Python, and does not necessarily require changes in the task.

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



✓ Testing the first example in the question.

✗ Testing the second example in the question. Your submission did not produce the correct output. Your program output:

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

when it was meant to output:

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

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

Figure 11: An example of a pair of consecutive programs submitted by a student that first failed T2 and then T1.

Overall, we observe that it is possible to identify tasks where students face difficulty, to drill down into specific locations where students fail to make progress, and to further inspect the students' submissions to search for misconceptions that may explain their failure to make progress.

4.3 Insights from Comparing Progress Networks

When analysing student data, teachers may wish to compare two different groups of students, or the same group of students in different contexts. For example, they may wish to compare students from different gender or age groups, or the same students on two different exercises.

In this section, we demonstrate how progress networks can support the comparison of students in two such example cases. Specifically, we first compare the progress graphs of older and younger students on the previously discussed *Echo! Echo!* task. Secondly, we compare the progress graphs of the overall student population between two comparable tasks that use semantically equivalent unit tests. Note that if the tasks do not use similar unit tests, more sophisticated matching techniques may be required. Also note that progress graphs are not intended as a test for differences between students, but instead a tool to provide insight into these differences.

4.3.1 Comparing students of different grade levels. Of the 23,381 students who attempted the *Echo! Echo!* task described in Section 4.2.1, we had school grade information for 23,327 of these students. Hypothesising that younger students would experience more difficulty on this exercise, we divided the students into two groups (older and younger) to compare their behaviour. In particular, group 1 included all students in Year 7[4] and under (10,411 students), and group 2 included all students in Year 8 and above (12,916 students). Note that the overwhelming majority of students (88.2%) were in Years 7 and 8, so this cutoff was chosen to balance the group sizes as much as possible.

In order to test our hypothesis, we compared the number of students in each group who passed the exercise on their first attempt. Specifically, we modelled the distribution of first-attempt successes by a Bernoulli distribution and performed a one-sided test on the null hypothesis that younger students did not perform worse than older students. The result was a p-value of 0.0004, which corroborated our hypothesis that younger students experienced more difficulty, encouraging further investigation into the difference and thereby motivating the application of progress graphs.

Figure 12 displays the difference graph between the older students (Group 1) and the younger students (Group 2) on the task. Positive edges, marked in orange, indicate steps that were more frequent among younger students, while negative steps (blue) were more frequent among older students.

Note that values in difference graphs will generally be limited to a small range, so meaningful values are expected to be small. For example, consider the step from *start* to *pass*, which has a weight of -0.0093. Among older students, this step occurred 3738 times out of 46152 submissions in total, which is $\frac{3738}{46152} = 0.081$. Thus, the minimum weight this edge could possibly have had was -0.081, no matter what the value was for the younger students. Similarly, since the fraction for younger students was $\frac{2808}{39142} = 0.072$, the maximum the weight could have been was 0.072, even if *no* older students had taken that step. In this case, the value was $0.0717 - 0.0810 = -0.0093$, indicating that this step was more common among older students (specifically, $1 - \frac{0.0810}{0.0717} = 13\%$ more common).

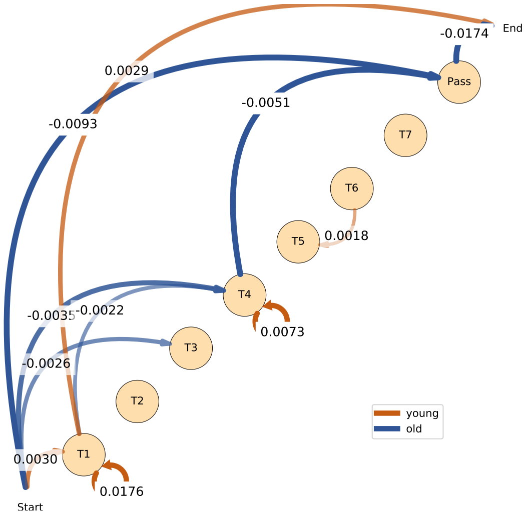


Figure 12: The difference graph between older (blue) and younger (orange) students on the *Echo! Echo!* task.

Table 1: Starting steps for students in each group

First Step	G1 (young)		G2 (old)	
T1	4477	43.0%	5139	39.8%
T3	455	4.4%	656	5.1%
T4	2069	19.9%	2600	20.1%
pass	2808	27.0%	3738	28.9%
other	602	5.8%	783	6.1%
Total	10,411 students		12,916 students	

From Figure 12, one general observation is that younger students tended to start with less complete programs. This is because the step *start* → *T1* (which indicates that the first submission passed no tests) is more common among younger students, while other steps such as *start* → *T3*, which indicate more progress, are more common in older students. Table 1 shows more information on the proportion of students in each group that started at the most interesting locations in the graph.

From the difference graph, we can also observe that younger students were more likely to give up after failing T1. Indeed, 536 younger students (5.1%) gave up here compared to 496 older students (3.8%). Furthermore, back steps (such as *T6* → *T5*) and repeat steps (such as *T4* → *T4* and *T1* → *T1*) appear to be more frequent among younger students. This suggests that younger students may have more difficulty correcting their programs than older students, resulting in them giving up or repeatedly failing tests more often.

While the progress network for the whole population in Section 4.2.1 leads to general insights for improving this exercise, this difference graph provides additional information that could be helpful in the context of equity. For example, since younger students were more likely to start at, repeatedly fail and give up at T1, this suggests that focusing on improving the feedback here first could be better than starting with other tests.

In summary, differences of progress graphs can help teachers to compare the behaviour of different populations of students. This can offer more insight into their learning than simple measures, such as the number of students who pass on their first attempt, and can thereby help to direct attention when improving the course to maximise equitable outcomes.

[4] from a K-12 classification (i.e. a median age of 13)

4.3.2 Comparing paired exercises. To analyse differences across tasks, we consider the two consecutive tasks *That Band Rules!* and *Misprefixed* from NCSS Challenge (Beginners) 2019 course. These are an example of a paired exercise, in which two problems requiring very similar solutions are provided. The first problem is scaffolded with initial code that the student must modify.

In the *That Band Rules!* task, the program should ask the user for their favourite band and output '$Band rules!', where '$Band' is replaced by the name of the band. Similarly, in the *Misprefixed* task the user is given some word and must return the output 'mis$word'. One may suspect either that the first task is more difficult, because it is the first time students encounter this type of task, or that the second is more difficult because it does not provide a starting program and interactive steps as scaffold.

Figure 13 displays the difference graph between both tasks with edges coloured blue if the edge is more common in *Misprefixed* and coloured orange if it is more common in *That Band Rules!*. Note that it is simple to match the test cases between these two tasks because they are semantically equivalent: T1 checks the user prompt roughly, T2 precisely, T3 the program output roughly, T4 precisely, and T5 checks a second example to make sure that the program does not hard-code a response. For other pairs of tasks, more sophisticated schemes may be necessary to match test cases. Also note that we compare the tasks on the same student population.

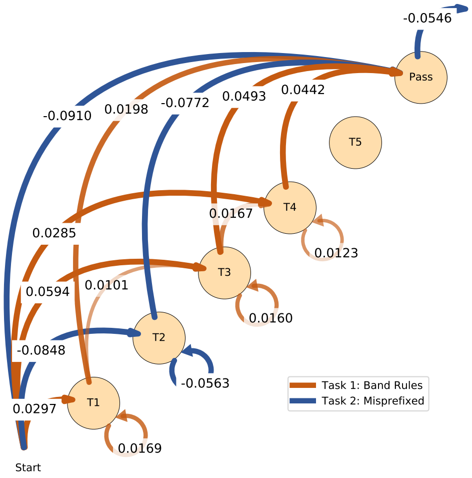


Figure 13: The difference graph between the *That Band Rules!* (orange) task and the *Misprefixed* (blue) task.

We observe that the most prominent difference (value −0.09) is that students were on average more able to complete the *Misprefixed* task in one step compared to the *That Band Rules!* task. This indicates that many students could successfully transfer their knowledge from the *That Band Rules!* task to the *Misprefixed* task, making the second task easier to complete. However, we also observe that students were more likely to repeatedly fail T2 for the *Misprefixed* task. One example of such a repeated failure is shown in Figure 14.

T2 fails because the student's prompt `'Word?'` does not end with a space as it should. Accordingly, the student receives the feedback "Your input prompt **does not match exactly**. Maybe you're **missing a space or punctuation**?" (emphasis in original). However, instead of adding a space in the prompt, the student removes a space in the response, which does not solve the issue. In this case, the emphasis on space and punctuation may have distracted the student from the key information that the problem is in the input prompt rather than the output message.

```
word = input('Word?')
print(f'{word} take')
```

```
word = input('Word?')
print(f'{word}take')
```

Figure 14: An example of a pair of consecutive programs submitted by a student that both failed T2 in the *Misprefixed* task.

Another possible explanation is that the student's current focus of attention is on the second line, such that the student intuitively tries to solve the issue there. Indeed, the second line also reveals a key misconception of the student, namely that the program puts 'take' after the input word rather than 'mis' in front of it. This problem would be revealed by T3, but this test is not executed because T2 already fails. To better align the student's focus of attention with the system message, one could re-order the tests and rather put T3 before T2.

5 IMPACT OF SAMPLE SIZE ON PROGRESS NETWORKS

In the previous sections, we applied progress networks to examples with large numbers of students (N > 4000 in each case). However, for some courses the number of students might be significantly less. As such, we performed one final investigation to explore how sample size impacted the progress network for the *Echo! Echo!* exercise from Section 4.2.1. We performed the investigation as follows:

(1) We selected a random sample of 200 student traces from the set of 23,381, then generated 200 progress networks from these traces. In particular, the first network was made from the first trace, the second from the first two traces, the third from the first three traces and so on.
(2) We repeated this process 1000 times. This gave us 1000 random examples of progress networks made from 1 student, 1000 examples made from 2 students and so on up to 200 students. This is similar to the sampling procedure in [13].
(3) We then compared these samples with the network we analysed in Section 4.2.1 (made from all 23,381 students) to see how different they were. We did this by first calculating the difference network between the sample and the original network, exactly as in Section 4.3. Then, we took the absolute of all the edge weights and summed these up to get the total difference. Finally, since the maximum possible difference is 2 if all the edges are different, we divided by 2 then multiplied by 100 to get the percentage error.

The results are shown in Figure 15. From this figure, we can see that the error score drops off quickly, reaching an average error of 24.5% with a sample of 25 students, 17.6% with 50 students, 12.7%

with 100 students and 9.1% with 200 students. In addition, the lower and upper quartiles are close to the mean, suggesting that there was not much variation between samples.

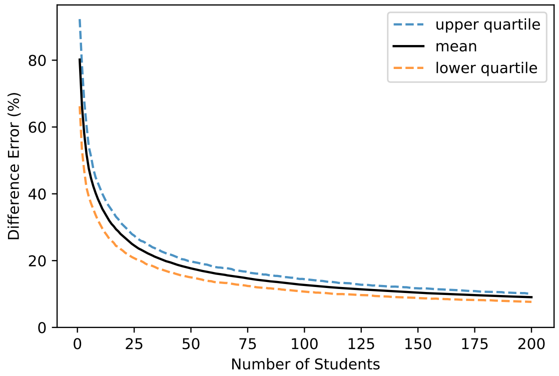


Figure 15: The impact of sample size on the *Echo! Echo!* progress network. The black line shows the mean error compared to the original network with 23,381 students.

These results indicate that even if we had only had a relatively small number of students, the progress graphs would have been similar to the original progress graph discussed in 4.2.1 from 23,381 students. This suggests that progress graphs are not only a useful tool for large-scale courses with many thousands of students, but can also be applied to smaller courses, and are thus useful in a wide variety of learning contexts.

6 CONCLUSION

In this paper, we have introduced the notion of *progress networks*, which summarise in a concise visualisation how a population of students progresses through a learning task. Importantly, this visualisation also highlights where students fail to make progress and rather get stuck or even move backwards. We used this visualisation to answer four types of research questions for introductory programming tasks, namely: Which tasks are difficult? Which step inside the task is difficult? Are there differences across groups? Are there differences across tasks? In all cases, we used purely quantitative measures to initially guide our attention and then "zoomed in" to the level of individual programs to reveal reasons why students may have failed to progress. Our examples revealed both typical misconceptions of students as well as potential issues with the learning environment, such as the order of tests, the emphasis in feedback messages, or a misalignment in the student's focus of attention and the reference of the feedback message.

A limitation of our current analysis is that we can retrieve examples for a certain typical node or edge in a progress graph, but we do not yet have a quantitative measure of how representative a certain program is for a node or edge. Future work could develop such a measure to use further quantitative information to guide our attention during analysis. It would also be interesting to explore additional test architectures and metrics for comparing progress graphs.

While we focused specifically on learning computer programming, our proposed analysis technique via progress networks can readily be applied to other fields as well. The only two required ingredients are student data in form of traces of partial solutions to a task, and an automatic classification of states into levels of progress. Our method is not specific to large data sets, either. In principle, even a single class of 20 students would be sufficient to generate meaningful visualisations. As such, we hope that progress networks become a helpful tool for many teachers to make sense of student data, identify difficulties and identify areas of improvement in their instructional design.

REFERENCES

[1] Tiffany Barnes, Behrooz Mostafavi, and Michael J Eagle. 2016. *Data-driven domain models for problem solving.* US Army Research Laboratory, Orlando, FL, USA, 137–145.
[2] Joseph E. Beck and Yue Gong. 2013. Wheel-Spinning: Students Who Fail to Master a Skill. In *Proc. AIED 2013.* 431–440.
[3] Stephen H. Edwards. 2003. Rethinking Computer Science Education from a Test-First Perspective. In *Proc. OOPSLA 2003.* 148–155.
[4] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L Thomas van Binsbergen. 2017. Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback. *IJAIED* 27, 1 (2017), 65–100.
[5] Elena Glassman. 2016. *Clustering and visualizing solution variation in massive programming classes.* Ph.D. Dissertation. Massachusetts Institute of Technology.
[6] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. *ACM SIGPLAN Notices* 53, 4 (2018), 465–480.
[7] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, and Jeremy Ernst. 2018. Misconception-Driven Feedback: Results from an Experimental Study. In *Proc. ICER 2018.* 160–168.
[8] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proc. Koli Calling 2010.* 86–93.
[9] Bryn Jeffries, Timothy Baldwin, Marion Zalk, and Ben Taylor. 2020. Online Tutoring to Support Programming Exercises. In *Proc. ACE 2020.* 56–65.
[10] Jessica McBroom, Kalina Yacef, Irena Koprinska, and James R Curran. 2018. A data-driven method for helping teachers improve feedback in computer programming automated tutors. In *Proc. AIED 2018.* 324–337.
[11] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. 2014. Codewebs: scalable homework search for massive open online programming courses. In *Proc. WWW 2014.* 491–502.
[12] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. 2015. Autonomously Generating Hints by Inferring Problem Solving Policies. In *Proc. L@S 2015.* Association for Computing Machinery, 195–204.
[13] Thomas W Price, Rui Zhi, Yihuan Dong, Nicholas Lytle, and Tiffany Barnes. 2018. The impact of data quantity and source on the quality of data-driven hints for programming. In *Proc. AIED 2018.* Springer, 476–490.
[14] Yizhou Qian, Susanne Hambrusch, Aman Yadav, Sarah Gretter, and Yue Li. 2020. Teachers' Perceptions of Student Misconceptions in Introductory Programming. *Journal of Educational Computing Research* 58, 2 (2020), 364–397.
[15] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Transactions on Computing Education* 18, 1 (2017).
[16] Kelly Rivers and Kenneth R. Koedinger. 2012. A Canonicalizing Model for Building Programming Tutors. In *Proc. ITS 2012.* 591–593.
[17] Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *IJAIED* 27, 1 (2017), 37–64.
[18] Valerie J Shute. 2008. Focus on formative feedback. *Review of educational research* 78, 1 (2008), 153–189.
[19] Jill Slay, G Quirchmayr, F Kurzel, and K Hagenus. 2003. Adaptive learning environments for CS education: from AMLE to live spaces. In *Proc. ACE 2003.* 257–262.
[20] John Stamper, Michael Eagle, Tiffany Barnes, and Marvin Croy. 2013. Experimental evaluation of automatic hint generation for a logic tutor. *IJAIED* 22, 1-2 (2013), 3–17.
[21] Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. 2018. Programming Misconceptions for School Students. In *Proc. ICER 2018.* 151–159.
[22] Hezheng Yin, Joseph Moghadam, and Armando Fox. 2015. Clustering Student Programming Assignments to Multiply Instructor Leverage. In *Proc. L@S 2015.* 367–372.