A Data-Driven Method for Helping Teachers Improve Feedback in Computer Programming Automated Tutors

Jessica McBroom[1(✉)], Kalina Yacef[1], Irena Koprinska[1], and James R. Curran[2]

[1] The University of Sydney, School of Information Technologies, Sydney, Australia
jmcb6755@uni.sydney.edu.au, {kalina.yacef, irena.koprinska}@sydney.edu.au
[2] Grok Learning, Sydney, Australia
james@groklearning.com

Abstract. The increasing prevalence and sophistication of automated tutoring systems necessitates the development of new methods for their evaluation and improvement. In particular, data-driven methods offer the opportunity to provide teachers with insight about student interactions with online systems, facilitating their improvement to maximise their educational value. In this paper, we present a new technique for analysing feedback in an automated programming tutor. Our method involves first clustering submitted programs with the same functionality together, then applying sequential pattern mining and graphically visualising student progress through an exercise. Using data from a beginner Python course, we demonstrate how this method can be applied to programming exercises to analyse student approaches, responses to feedback, areas of greatest difficulty and repetition of mistakes. This process could be used by teachers to more effectively understand student behaviour, allowing them to adapt both traditional and online teaching materials and feedback to optimise student experiences and outcomes.

Keywords: Data-driven teacher support · Automated tutoring systems
Feedback improvement · Tutoring system evaluation

1 Introduction

In recent years, steady progress has been made towards developing new data-driven methods for feedback generation in automated tutoring systems. These are online systems which provide automated marking of submissions and feedback to students. This work has led to an increase in the sophistication of tutoring systems, with great potential to improve educational outcomes. However, it has also increased the difficulty of analysing and assessing the automated feedback given by these systems. As the systems become more complex, teachers become more distanced from the direct feedback and instruction given to individuals, and this is exacerbated by the increasing number of students. It is therefore very challenging for teachers to identify and

C. Penstein Rosé et al. (Eds.): AIED 2018, LNAI 10947, pp. 324–337, 2018.
https://doi.org/10.1007/978-3-319-93843-1_24

diagnose issues with the system and monitor whether the instructional material they have created is appropriate or could be improved.

Thus, to ensure that automated tutoring systems meet their potential to enhance student experiences, it is essential to close the loop between teachers, systems and students, by ensuring that information about student interactions with these systems is passed back to teachers. In particular, it is important to develop data-driven methods for assessing automated tutoring systems alongside methods of extending them. These methods can allow teachers to use the wealth of data collected by the tutoring systems to modify and improve these systems. They can also provide useful insights into student learning, which can be applied in other contexts, such as inside the classroom, to improve learning.

In this paper, we present a new data-driven method for assessing automated tutoring systems in the context of a computer programming course. This method helps teachers to understand the kinds of programs students write and how they change over time in response to feedback. This allows them to: (i) gain insight into student learning, e.g. by finding exercise parts where students have trouble, or identifying common mistakes and difficult concepts, and (ii) improve the tutoring system, e.g. by revising the pre-set feedback and testing cases in the system based on student behaviour. We demonstrate an application of this method to real data from a beginner programming course in Python, and also discuss possible extensions and broader applications of the method to other areas.

2 Related Work

A variety of automated marking and tutoring systems have been reported in computer science courses [1–7] and other areas [8–11] in recent years, and there has been much work focused on improving the sophistication of feedback provided by these systems [1, 5, 6, 8–11] Evaluations of these systems have often focused on functionality of the system, including its usability [2] and the availability of hints [5, 6, 8, 10], student opinions and engagement [4, 11, 12], before and after studies [11, 12] or comparisons with expert opinions. These evaluation methods provide valuable feedback to teachers, but some (e.g. comparative studies) do not scale well to large numbers of students due to the significant manual work and time required, and the information provided about how students interact with the system is often more general. Our work aims to address this by providing teachers with a way to understand and visualise how large numbers of students interact with an automated tutoring system, including how they respond to feedback, which can lead to clear and actionable ways to improve the system.

An important part of our work involves clustering similar programs. Previous work related to this includes clustering programs based on features or on their structure by using metrics such as the tree edit distance [13]. Gross et al. [14] investigated the usefulness of different similarity metrics in the context of programming and Paassen et al. [15] discussed a scheme for learning parameters of structure metrics, which could be used to determine program similarity. Another method is to focus on functionality or semantics of programs. In [16], similar programs were grouped by renaming all similar variables across programs (determined by running the programs), then combining

programs with the same sets of lines of code. Our clustering involves applying functionality-preserving transformations to programs, followed by further processing to account for differences that are irrelevant in the context of the exercise. Functionality-preserving transformations have been applied in other work for different purposes and languages [1, 5].

Our work falls into the broader area of mining log data from automated tutoring systems to provide teachers with information to improve educational outcomes. Other related work includes predicting students at risk of failing early in the semester [17], characterizing and predicting the performance of different groups of students [18] and analysing student behaviour and its evolution throughout the semester [19, 20].

3 Data

Our data come from Grok Learning [19], an online platform offering various programming courses of different difficulty levels to students, each consisting of a series of interactive lessons and exercises for students to complete. The exercises are marked automatically using a series of test cases written in advance by teachers, and students are provided with feedback based on the tests passed and failed. The students can then use this feedback to improve their programs, re-submit and re-test them, until all tests are passed.

The dataset used in this paper is extracted from a beginner Python programming challenge run through Grok Learning in 2017. It comprises of programming attempts, failed or successful, from two programming exercises ('What Rhymes with Grok' and 'Letter from the Queen'). Each exercise was attempted by over 5000 students, and the data includes all student programs that were run through the system or submitted for marking, any tests passed or failed by the programs, the feedback automatically generated by the tests, submission times and information on any errors from running the programs.

4 Method

The method we propose aims to give teachers an overview of student interactions with an automated tutor for computer programming for a given exercise. This includes: the types of programs students submit, how students respond to the given feedback, the student progression through the exercise and the exercise parts where students experience difficulty. This knowledge can assist teachers in identifying system strengths and areas for improvement, and also in gaining insights into student learning.

Importantly, as our method is data-driven, using real student programs and their subsequent revisions after receiving feedback, it can reveal underlying problems experienced by students that were not necessarily anticipated by the teacher when devising the test feedback. For example, in Table 1, a sample student submission for an exercise discussed in the next section is shown, along with the feedback received. The issue with the program is that it prints a hardcoded answer, instead of the user's input, and the feedback given addresses the issue by reminding the student to use a variable

instead. However, our data-driven approach reveals that 100/172 students in this situation simply changed their hardcoded message after receiving the feedback. This is not what a teacher would expect, and so highlights the value of data-driven analysis techniques. Our approach can also reveal bugs in test cases, opportunities to make these tests more targeted and whether students make similar mistakes in later exercises.

Table 1. Sample student program and feedback received. The feedback seems useful, but many students do not respond as expected to it.

Student Program	Feedback
`input('What rhymes with Grok? ')` `print('rock')`	Testing the second example in the question (when the user enters sock). Your submission did not produce the correct output. Your program output: What rhymes with Grok? **sock** rock when it was meant to output: What rhymes with Grok? **sock** sock Remember to store the user input in a variable, and to use the variable name with print

Our approach consists of two main steps, summarised in Fig. 1. The first is to cluster student programs based on their functionality, and the second is to extract useful information from the clusters by building an interaction graph and using sequential pattern mining. We describe these steps in more detail below.

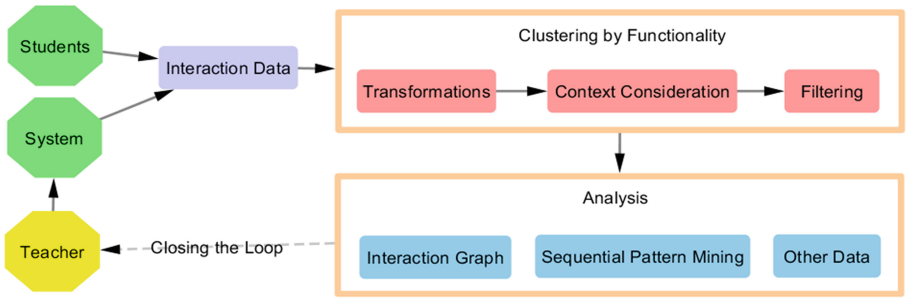

Fig. 1. Summary of our approach to closing the loop between teachers, systems and students

4.1 Clustering

To allow teachers to interpret a large number of student program submissions, we begin by clustering programs based on their functionality. Two programs are considered to have the same functionality if an observer with access to any input/output examples cannot distinguish between the two programs. This definition is consistent with how

the testing is done using the pre-specified test cases. The advantage of using this clustering approach is that a group of programs functioning similarly is easily exemplified by a single example, so the clusters are easy to interpret. Though differences in syntax and efficiency can reveal further interesting information about student understanding, we do not focus on these here.

The clustering in the form we describe is applicable to 'calculation-based' programs. That is, programs employing any of the following: literals, mathematical operators, variables, simple functions and console-based input and output. However, this could be extended to broader classes of programs in future.

The method consists of three steps: the application of functionality-preserving transformations, considering context and filtering, which we outline in turn.

Functionality-Preserving Transformations. We describe a number of transformations with Python in mind, but these could be generalised to other languages. Some examples are shown in Table 2.

Table 2. Examples of Python programs and their standardised forms

Original Program	Standardised	Explanation
`print(input('Quo kka '))`	`print('Quokka ', end='') i1 = input() print(i1)`	separate input into printing and input, new variable for changing value, standardised variable naming
`a = input() b = '…' c = a + b print(c)`	`i1 = input() print(i1 + '…\n', end='')`	substitute variables, standardise printing, standardised naming
`'''comment''' a = 2 print(7*12)`	`print('84', end = '')`	remove comments and unused variables, evaluate expressions, standardise printing

Variables. In general, transformations are applied so that variables are used only and always for values that can vary on different runs of the program. Such values include the output from random or input functions. If variables are not being used for these values, they are introduced. Variables used for all other values (such as literals) are removed and any occurrences of them are replaced by their value. The names of variables are then standardised.

Expressions. Expressions are simplified where possible. This includes the evaluation of consecutive literals or literals in commutative expressions and the elimination of trivial operations, such as empty string addition. If the value of an expression can be determined without complete knowledge of all operands, it is also evaluated (e.g. an expression with multiplication by 0).

Expressions are also expanded to standardise their form. This includes application of distributive laws and replacement of some operations involving literals. For

example, `a**2` would be replaced by `a*a`. For consistency, commutative expressions are ordered by operand types, e.g. integers are placed first.

Function-Specific Transformations. Transformations specific to allowed functions are applied to standardise programs further. For example, an `input` call can be separated into a `print` call followed by an `input` call with no arguments. Consecutive print statements are collapsed into a single statement with no varying keywords. Type casts are removed where possible.

Other. If two variable declarations can be swapped without altering the function of the program, a standardised ordering is applied, for example, based on the variable used first. Whitespace, comments and any lines that do not affect the program's functionality are removed.

Considering Context. After transformations are applied and functionally equivalent programs are clustered together, the number of clusters is then reduced by ignoring differences that do not matter. An example of such a difference is whether a program prints '`hi`' or '`hello`' in an exercise where the goal is to print '`How are you?`': in both cases, the programs are incorrect for the same reason.

This reduction is achieved by combining programs with a similar standardised form together, so long as they fail the same test cases. For simple programs, this can just be differences in strings or numerical values. For more complex programs, some mathematical expressions could be treated as equivalent, or some statements with the same general purpose. In our case study, we combined programs with exactly the same abstract syntax tree structure and node types so, for example, differences in strings would be ignored, so long as the same tests were passed and failed.

Filtering. Finally, to ensure teachers are left with a manageable number of clusters to consider, clusters containing less than a specified number of programs are filtered out. In the case study we describe next, this threshold was 10.

4.2 Analysis

After clustering, student interactions with the system are analysed using a combination of different methods: building an interaction graph and finding frequent patterns.

In the interaction graph, nodes and edges represent student program clusters and their transitions between these clusters respectively. By providing a general overview of how students progress through the exercise, this graph can reveal places where students struggle and therefore can be used to improve feedback. For example, cycles in the graph could be indicative of student confusion or misleading feedback. Student starting positions in the graph could also indicate how prepared students were for the exercise, and ending positions in the graph could highlight areas of difficulty.

Next, sequential pattern mining is used to find common consecutive transitions between clusters. This is done by iterating through the transitions of each student and adding all unique paths of a given length to a tally. For example, a tally of 370 for the path *abc* would indicate that 370 unique students submitted three consecutive programs

in clusters *a*, *b* then *c* at some point when completing the exercise. Paths can then be assessed for their efficiency and how reflective they are of the teacher's expectations.

Any potentially interesting areas can be analysed in conjunction with other data from the tutoring system, including submission times, how often students run their code before marking it, which tests are passed and failed and samples of feedback for different program clusters. This can allow teachers to assess whether feedback was appropriate and students responded to it as expected, and how it could be improved in future. It can also highlight opportunities to make testing more targeted if different types of programs are failing the same test, and reveal issues with testing procedures if some programs are passing tests they shouldn't pass.

Finally, the performance of students across exercises can be observed to assess their learning progress. For example, teachers can see if students continue to make similar mistakes or continue to require similar feedback to earlier exercises.

5 Case Study

We demonstrate the application of this method using two beginner programming questions from the Grok Learning programming challenge. These questions are particularly interesting because their completion rate is very high (over 97%), so a traditional analysis could risk skipping over them. However, the small percentage of students who do not complete the tasks still represent a large enough group to be interested in helping, and any improvements could help all students to interact more usefully with the system.

In the first example, we will show how the method can be used to assess whether student responses in different contexts reflect what an instructor might expect, and how this can be used to improve the system test cases and the feedback provided to students. In the second example, we focus on how the method can be used to understand student learning and help to develop more targeted testing and feedback.

5.1 Example 1 – What Rhymes with Grok?

In this exercise, students write a program to print user input. Details of the exercise from a student's perspective are shown in Fig. 2, with instructions on the left, sample code in the top right and details of tests run on the code in the bottom right.

Figure 3 shows the interaction graph produced by clustering programs of 5874 students who attempted this question. Circular nodes represent student program clusters, and edges show the number of times a submission in the source node cluster was followed by a submission in the target node cluster. Node and edge size correspond to the number of submissions, so larger nodes and edges are more important. Edges from the start and end nodes show the number of students who started and ended in each cluster. Table 3 shows the most common cluster paths of length 3 and 4 among students from the sequential pattern mining, ordered by the number of students taking the paths.

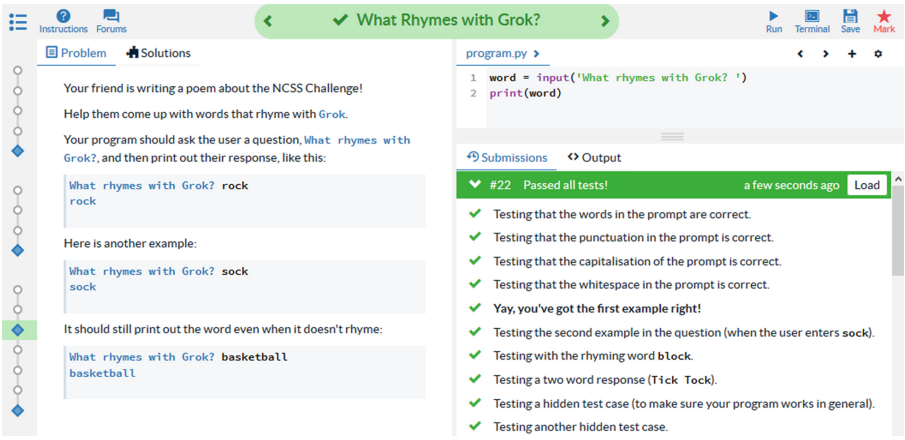

Fig. 2. A screenshot from a student's perspective of the exercise "What Rhymes with Grok?"

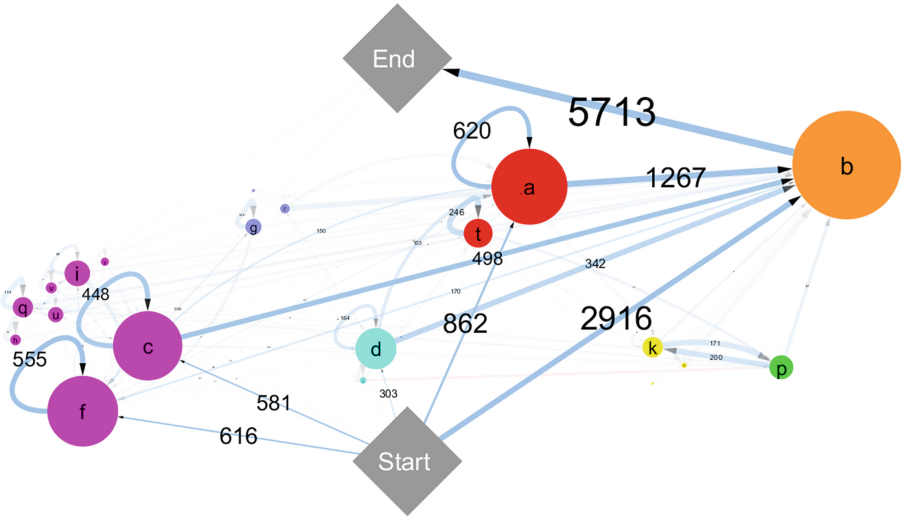

Fig. 3. Interaction graph for the exercise "What Rhymes with Grok"

Identifying Areas of Interest. From the graph and common paths, we can make some general observations about student interactions with the system. Firstly, cluster *b* is the most common starting point, followed by *a*, *f*, *c* and *d*. In the graph, there are self-loops on *a*, *f*, *c* and *d* and a cycle between *p* and *k*, and repetitions of these also appear in many common paths. The paths *cab* and *dab*, with three different nodes, could also be interesting. Table 4 shows a summary of the potentially interesting clusters just identified, including a sample program from each cluster.

Table 3. Path patterns for the exercise 'What Rhymes with Grok?"

Path Length	Cluster Path and Number of Students										
3	*aab* 306	*ccb* 170	*aaa* 134	*fff* 129	*ccc* 102	*cab* 94	*dab* 85	*ddb* 82	*ffb* 59	*pkp* 53	*ttt* 52
4	*aaab* 119	*cccb* 72	*aaaa* 64	*ffff* 47	*cccc* 47	*tttt* 34					

Table 4. Cluster summaries

Cluster	Size	Failed test	Sample Program
b	5713	None	`x = input("What rhymes with Grok? ")` `print(x)`
a	2044	prompt ws	`x = input("What rhymes with Grok?")` `print(x)`
f	1455	prompt words	`print('Hello!')`
c	1281	prompt words	`name = input('what is your name ')` `print(name)`
d	691	prompt case	`name = input('what rhymes with Grok? ')` `print(name)`
p	397	eg2	`input('What rhymes with Grok? ')` `print('rock')`
k	338	eg1	`name = input('What rhymes with Grok? ')` `print('clock')`

Investigating Areas of Interest. From where students began, it is clear many were generally well prepared for the exercise: cluster *b* indicates a correct solution, and clusters *a*, *c* and *d* are close to a correct solution, with just some issues with the prompt string. The students who began at *f*, however, were missing some key concepts as they were only using the print function.

In relation to the cycle between *p* and *k*, both clusters are similar in that students used the input function, but printed out a hardcoded message instead of the user's response. When this hardcoded message was not 'rock', the first example test was failed. Otherwise, the second test was failed. A cycle between these clusters suggests students were changing the hardcoded string instead of correcting the issue. The feedback given to these students, shown in Table 5, consisted of a notification that their output was incorrect and, in the case of *p*, a suggestion on how to correct this was given. However, according to the data, 50% of submissions (200/397) in cluster *p* were followed by a submission in cluster *k*, and this involved 58% of unique students (100/172) who had a program in cluster *p*. For *k*, the figures were similar: 171/338 submissions (51%) and 84/139 students (60%). Though a teacher might expect students to change the hardcoded string based on the feedback at *k*, the fact that the same is true

Table 5. Sample feedback for programs in clusters *k* and *p*

k Sample Feedback	*p* Sample Feedback
Testing the first example in the question. Your submission did not produce the correct output. Your program output: What rhymes with Grok? **rock** clock when it was meant to output: What rhymes with Grok? **rock** rock	Testing the second example in the question (when the user enters sock). Your submission did not produce the correct output. Your program output: What rhymes with Grok? **sock** rock when it was meant to output: What rhymes with Grok? **sock** sock Remember to store the user input in a variable, and to use the variable name with print

of *p* is surprising. One possibility is that students try to modify their code as soon as they see the output is wrong, without reading all of the feedback. This could perhaps be addressed by including the suggestion at the beginning of the feedback.

The common paths, *dab* and *cab*, are examples of students responding to feedback as expected, being guided efficiently to the solution. At *d*, *c* and *a*, the feedback directs students to correct the words, capitalisation and whitespace respectively in their input prompts. The median time taken for each of the transitions (*ca*, *da* and *ab*) was 54, 32 and 44 s respectively, and the median number of times students ran their code after correcting their code was one, the required number. This suggests students on these paths quickly understood and responded to the given feedback.

Of the clusters with self-loops, students in *f* seemed to experience the most difficulty: it has the largest proportion of repeated submissions (38%) by the largest proportion of unique students (42%), and the largest median time between repeats (42 s). In fact, 24% of the 161 students who did not complete the exercise stopped at *f*. Though this number was still small relative to the total number of students, it could still be an opportunity to improve feedback. The feedback given for the sample program in cluster *f* focused on the input prompt not containing the correct words, but perhaps it could be changed to target the presence of certain features in a student program instead, such as an input function.

Unexpected Clusters. This exercise provides an example of how unexpected clusters of programs can reveal potential issues with test cases. One small cluster of programs (21 instances) that failed the first example test had an interesting structure: the programs asked for input from the user and printed it correctly, but the input prompt was incorrect. This was interesting, because they passed the four tests checking the input prompt. By analysing the programs in this cluster, teachers could correct the tests, which could improve not only this exercise, but any others using similar tests.

5.2 Example 2 - Letter from the Queen

We briefly discuss a later exercise to show how this method can be used to measure student improvement over time. This exercise requires students to prompt the user for their age, convert the input to an integer, subtract it from 100 and print out the result. The interaction graph for this exercise is shown in Fig. 4.

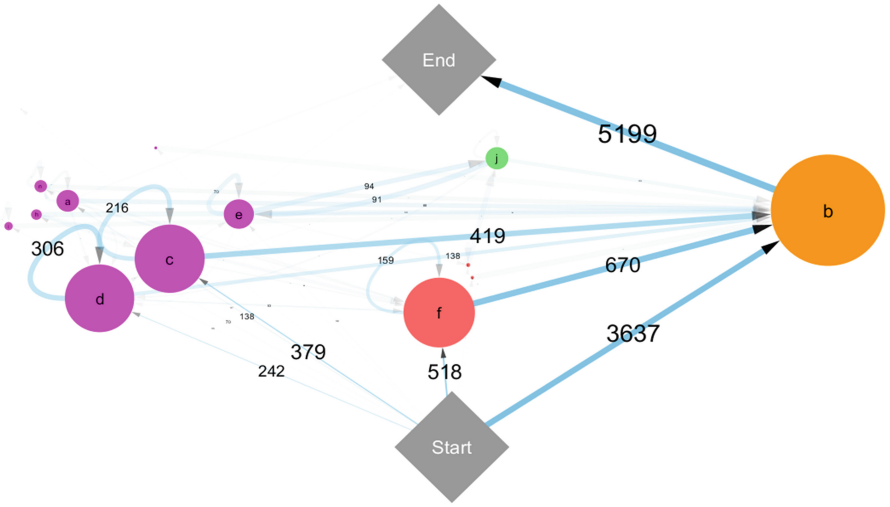

Fig. 4. Interaction graph for the exercise "Letter from the Queen"

Measuring Improvement. Despite the exercise being more difficult, a greater proportion of students (69% compared to 50%) submitted a correct program on their first try, and 99% completed the exercise, indicating students improved in this later problem. A similar hardcoded cycle as in the last problem can be seen (between *e* and *j*), suggesting some students were repeating the same mistakes, but this occurred for a smaller number of students. Similarly, the cycles on *f* and *c* are similar to the cycles on *d*, *c* and *a* in the earlier exercise in that they involve mostly correct student programs with incorrect prompt strings. These too involve fewer students than previously, suggesting improvement. Cluster *d* represents programs attempting to read too much input, which was much less frequent in the previous exercise. This new issue could be a result of students needing to print more output, which they mistakenly did by overusing the input function. This overall suggests students were generally improving, with less making the same mistakes as before and more making new mistakes as the question increased in difficulty.

Targeting Tests. In this exercise, 9 tests were run on student programs, but almost all failing programs were caught by the first three tests. This suggests there is an opportunity to make the tests more targeted to student code. From the clustering, teachers can see different kinds of programs failing the same test, such as is shown in Table 6. This

can then be used to create tests targeted at the different approaches. For example, one could be targeted towards hardcoded answers (*t*), another could check the subtraction is done correctly (*g*) and another could check for correct strings (*f*).

Table 6. Sample programs from clusters that fail the second test, which checks grammar

Cluster	Cluster Size	Sample Program

f	847	`print("Years until your letter...")`
		`time = 100 - age`
		`print(time)`

t	46	`input('How old are you?')`
		`print('Years until your letter...')`
		`print('85')`

g	41	`age=input ('How old are you?')`
		`print ('Years until your letter...')`
		`print ( int(age) - int( 100))`

6 Discussion and Conclusions

In this paper we have explored a new data-driven method for the assessment of online automated tutors in the context of computer programming. This method involves clustering student programs based on their functionality, building an interaction graph and applying sequential pattern mining, while also using other data from the system, to analyse student interactions and progress.

Our method can be used to gain insights into student learning, including how students begin and end an exercise (e.g. indicating if they had the required prior knowledge), common paths taken and cycles that may be indicative of difficulty. By investigating these areas, teachers can identify difficult concepts and common mistakes. The differences in student behaviour across exercises can allow teachers to measure student learning by observing similarities and differences in the types of progressions and mistakes made by students in each exercise.

This information can also facilitate the improvement of tutoring systems. For example, the identification of areas where the feedback provided to students works well, and areas where it can be improved, including in places where student responses may be unexpected, can help teachers to revise automated feedback. In addition, it can assist in the improvement of test cases: unexpected student program clusters can be used to find problems with the sequence of tests, and different program clusters that fail the same test can be used to create more targeted tests. This information can also be used to provide feedback to students within the classroom, e.g. by explaining common issues and how to address them before attempting an exercise.

In future, this method could be improved by evaluating it with other teachers and extending it to a broader class of computer programs and languages. It could also be combined with other methods for clustering and analysis to provide further information to teachers on the quality of the student programs, such as style, time and space complexity, and could possibly be applied to all exercises in the course at once rather than individual exercises.

Another direction for future work is extending the method to generate automated feedback for students. For example, students could be encouraged to try new styles of coding: by identifying parts of their code that functioned similarly to parts of other students' code, a tutoring system could suggest they try the other style, which might be shorter or more efficient. Our method could also potentially be used in the generation of feedback by identifying functionally similar past examples and adapting the given feedback to new problems.

Ultimately, in a society where automated tutoring systems are becoming increasingly complex and serving larger and larger cohorts of students, data-driven methods such as the proposed method are essential to support teachers in their assessment with empirical evidence. In this way, we can close the loop between teachers, students and tutoring systems, ensuring teachers can diagnose and correct unexpected issues and, in doing so, allow these systems to meet their full potential as effective learning tools. Thus we believe the method we have presented is useful, not only to aid teachers now, but as a stepping stone to the continued improvement of tutoring systems in the future.

References

1. Gerdes, A., Heeren, B., Jeuring, J., van Binsbergen, L.T.: Ask-Elle: an adaptable programming tutor for haskell giving automated feedback. Int. J. Artif. Intell. Educ. **27**(1), 65–100 (2017). https://doi.org/10.1007/s40593-015-0080-x
2. Venables, A., Haywood, L.: Programming students NEED instant feedback! In: Proceedings of the Fifth Australasian Conference on Computing (ACE), pp. 267−272. ACM, Adelaide (2003)
3. Edwards, S.H., Perez-Quinones, M.A.: Web-CAT: automatically grading programming assignments. In: Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE), p. 328. ACM, Madrid (2008). https://doi.org/10.1145/1597849.1384371
4. Enstrom, E., Kreitz, G., Niemela, F., Soderman, P., Kann, V.: Five years with Kattis using an automated assessment system in teaching. In: Proceedings - Frontiers in Education Conference (FIE), pp. T3 J-1−T3 J-6, IEEE, Rapid City (2011). https://doi.org/10.1109/fie.2011.6142931
5. Rivers, K., Koedinger, K.: Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. Int. J. Artif. Intell. Educ. **27**(1), 37–64 (2017). https://doi.org/10.1007/s40593-015-0070-z
6. Chow, S., Yacef, K., Koprinska, I., Curran, J.: Automated data-driven hints for computer programming students. In: Adjunct Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization (UMAP), pp. 5−10, ACM, Bratislava, Slovakia (2017). https://doi.org/10.1145/3099023.3099065
7. Gramoli, V., Charleston, M., Jeffries, B., Koprinksa, I., McGrane, M., Radu, A., Viglas, A., Yacef, K.: Mining autograding data in computer science education. In: Proceedings of the Australasian Computer Science Week Multiconference. ACM, Canberra (2016). https://doi.org/10.1145/2843043.2843070
8. Stamper, J., Barnes, T., Lehmann, L., Croy, M.: The hint factory: automatic generation of contextualized help for existing computer aided instruction. In: Proceedings of the 9th International Conference on Intelligent Tutoring Systems (ITS), pp. 71−78. Springer, Montreal, Canada (2008)

9. Johnson, S., Zaiane, O.: Deciding on feedback polarity and timing. In: Proceedings of the International Conference on Educational Data Mining (EDM), IEDMS, Chania, Greece, pp. 220−222 (2012)
10. Barnes, T., Stampler, J.: Automatic hint generation for logic proof tutoring using historical data. J. Educ. Technol. Soc. **13**(1), 3–12 (2010). https://doi.org/10.1007/978-3-540-69132-7_41
11. Perikos, I., Grivokostopoulou, F., Hatzilygeroudis, I.: Assistance and feedback mechanism in an intelligent tutoring system for teaching conversion of natural language into logic. Int. J. Artif. Intell. Educ. **27**(3), 475–514 (2017). https://doi.org/10.1007/s40593-017-0139-y
12. Dominguez, A., Yacef, K., Curran, J.: Data mining for individualized hints in eLearning. In: Proceedings of the International Conference on Educational Data Mining (EDM), IEDMS, Pittsburgh, PA, United States, pp. 91−100 (2010)
13. Yin, H., Moghadam, J., Fox, A.: Clustering student programming assignments to multiply instructor leverage. In: Proceedings of the Learning at Scale Conference (L@S), pp. 367−372. ACM, Vancouver (2015). https://doi.org/10.1145/2724660.2728695
14. Gross, S., Mokbel, B., Paassen, B., Hammer, B., Pinkwart, N.: Example-based feedback provision using structured solution spaces. Int. J. Learn. Technol. **9**(3), 248–280 (2014). https://doi.org/10.1504/ijlt.2014.065752
15. Paassen, B., Mokbel, B., Hammer, B.: Adaptive structure metrics for automated feedback provision in intelligent tutoring systems. Neurocomputing **192**, 3–13 (2016). https://doi.org/10.1016/j.neucom.2015.12.108
16. Glassman, E., Scott, J., Singh, R., Guo, P.J., Miller, R.C.: Overcode: visualizing variation in student solutions to programming problems at scale. ACM Trans. Comput. Hum. Interact. **22**(2), 1–35 (2015). https://doi.org/10.1145/2699751
17. Koprinska, I., Stretton, J., Yacef, K.: Students at risk: detection and remediation. In: Proceedings of the International Conference on Educational Data Mining (EDM), IEDMS, Madrid, Spain, pp. 512−515 (2015)
18. Koprinska, I., Stretton, J., Yacef, K.: Predicting student performance from multiple data sources. In: Conati, C., Heffernan, N., Mitrovic, A., Verdejo, M.F. (eds.) AIED 2015. LNCS (LNAI), vol. 9112, pp. 678–681. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19773-9_90
19. McBroom, J., Jeffries, B., Koprinska, I., Yacef, K.: Mining behaviours of students in autograding submission. In: Proceedings of the International Conference on Educational Data Mining (EDM), IEDMS, Raleigh, NC, United States, pp. 159−166 (2016)
20. McBroom, J., Jeffries, B., Koprinska, I., Yacef, K.: Exploring and following students' strategies when completing their weekly tasks. In: Proceedings of the International Conference on Educational Data Mining (EDM), IEDMS, Raleigh, NC, United States, pp. 609−610 (2016)
21. Grok Learning. https://groklearning.com. Accessed 8 Feb 2018
22. Figures 1, 3 and 4 produced using Cytoscape graphing software Cytoscape. www.cytoscape.org. Accessed 8 Feb 2018