Detecting When a Learner Requires Assistance with Programming and Delivering a Useful Hint

Marcus Messer
King's College London
marcus.messer@kcl.ac.uk

ABSTRACT
Over the last few years, computer science class sizes have increased, resulting in tutors providing more support to struggling students, and instructors having less time per-student in larger classes. Universities typically assign multiple tutors to lab sessions, especially introductory programming courses, to maximise the help available to students during their sessions. However, using multiple tutors does not help struggling students outside of official sessions. The lack of support outside official settings is especially the case for online courses and remote learning. To help resolve student frustration from not being able to get support when they need it, we propose a tool that can detect when a student is struggling with their programming task and give them a hint that gets them closer to their goal.

Keywords
computer science education, computer programs, sequence mining, learning behaviours, feedback

1. INTRODUCTION
Over the last few years, the number of students enrolled in computer science courses has increased [12] and self-directed learning is becoming more prevalent. As class sizes grow, it becomes more difficult for course leaders and assistants to assist in programming labs. This difficulty is because it takes time to solve and explain various programming issues, and instructors have less time per-student in larger classes. Multiple tutors are frequently assigned to a single lab to maximise the availability of expert assistance. However, not all students will receive assistance during their scheduled lab sessions in some circumstances, particularly approaching deadlines, when multiple requests for help occur in a single lab session. Even if a tutor is available, many students do not seek assistance when struggling to solve a problem [15]. However, if a tutor offers a student support, they will usually accept it.

M. Messer. Detecting when a learner requires assistance with programming and delivering a useful hint. In A. Mitrovic and N. Bosch, editors, *Proceedings of the 15th International Conference on Educational Data Mining*, pages 778–781, Durham, United Kingdom, July 2022. International Educational Data Mining Society.

Students have limited access to expert support outside of regular classes and office hours. The limited access is especially true for online courses, distance learning and students undertaking self-directed learning. The limited expert help in and outside official settings may lead to prolonged frustration, one of the leading causes of students dropping out [20].

A student could seek help for numerous programming areas, including compiler errors, logical errors, code style and best practice. Detecting code style issues and best practices are solved using various static code analysis tools, such as FindBugs and CheckStyle. FindBugs detects bad practices, performance, correctness, and "dodgy code" using pre-defined bug patterns, including common issues for novice programmers. Examples of which includes the comparison of String objects using "== or !=" and issues that cause `IndexOutOfBoundsException` [2]. CheckStyle detects various code style issues, including incorrect naming conventions and code design [7].

Some of the most challenging errors to solve are compiler errors, examples of which include "missing return statement", "method call: parameter type mismatch", and "method call targeting the wrong type" [16]. Understanding compiler error messages and correcting them is difficult and time-consuming for programmers of any degree of experience, especially novices [22]. Poorly written error messages could lead to a student not solving a problem for hours or days without help. Even with skilled help, an error could take several minutes to correct [13]. While compiler errors can be challenging to solve, logical errors are considerably more challenging to correct because they usually necessitate understanding the problem's context and the student's written work.

The proposed research will focus on detecting when a student requires assistance with compiler, logic and style errors and providing a meaningful hint on how to solve the error.

2. BACKGROUND & RELATED WORK
Previous work on detecting a struggling student has focused on other aspects than the source code. Spacco et al. used measures based on time and number of submissions to detect *flailing* students [21]. Rodrigo and Baker investigated detecting the aggregate frustration over many labs, using compilation based measures [20].

The compilation measures that Rodrigo and Baker implemented were the average time between compilations, the total number of compilations, the total number of errors, and Jadud's Error Quotient (EQ) [20]. The EQ indicates how well a student can handle syntax errors while programming. It looks at successive pairs of compile events to see if both result in an error and if the error type is the same [9] (refined in [10]).

Recent research has investigated how EQ could be improved. Watson and Godwin created the Watwin algorithm, using consecutive compiler events to calculate a score; however, they also included checks for whether the error is on the same line and penalties for the time taken to solve each compilation pairing [23]. In addition, Becker developed the repeated error density (RED), which calculates a score based on repeated error strings, accounting for the lengths of repeated error strings in a sequence and assigning higher penalties for higher repeated error density [4].

While programming, students are given different types of feedback, the most common of which is next-step hints. After a student is stuck in a state and requires help to progress, a next-step hint will guide them to a more complete and correct version of the code [1].

There have been many implementations of next-step hints in different programming environments. Obermüller et al. have recently implemented next-step hints into Scratch. They first select a set of suitable candidate solutions using an automated test suite, then find the best matching candidate solution as the target solution. After they have the students and the target solution, they determine differences in the Abstract Syntax Tree (AST) to synthesise the hints [17]. However, this solution requires abundant candidate solutions.

There have been multiple attempts to provide hints in vast and sparsely populated spaces. Paaßen et al. produced the Continous Hint Factory (CHF), which uses a supervised machine learning approach to provide a hint to the student. CHF uses the edit distance of the student's current state and traces data of past students who have visited the same state in similar states to choose the edit with the lowest error as the hint [18].

While Paaßsen et al.'s approach still require some trace data, Rivers and Koedinger use at least one reference solution and a test method to produce a solution space to find the next-step hint. They use a path construction algorithm to find the edit path from the current state to the solution, then finds the closest correct state within the space to find the edit that will be the basis of the next-step hint [19].

The previous approaches to next-step hint generation require trace data or a reference solution, but what if the student is the first to attempt such a task? Efremov et al. have used a reinforcement learning-based neural network hint policy to enable next-step hints for students attempting a programing task for the very first time [8].

3. PROBLEM STATEMENT

As Computer Science courses grow, the need for expert help increases; this is especially the case for online courses, remote learning and approaching coursework deadlines. Although having more tutors in regular sessions allows for more help, increasing the support offered in and out of the classroom would be beneficial.

Our research will investigate the following research questions:

RQ1 How can traces of student code changes be used to detect struggling students?

RQ2 After detecting a struggling student, when is the most effective time to give them a hint?

RQ3 Can we generate next-step hints that focus on syntactic and logical errors comparable to the support offered by human tutors in these areas?

4. PROPOSED METHOD

The first phase of this project will determine whether or not a student is struggling. We will use Blackbox, a large dataset of novice programming data collected over the last eight years [6], to look for trends that could signal a struggling student. Blackbox is a large scale repository of novice programmers' activity, including the length of programming sessions, compilation event history, and editing behaviour [6]. The editing behaviour includes a history of source code changes, allowing researchers to step through each change of a student's code iteratively. We will explore patterns that could identify struggling students using the source code and compiler histories.

We will use existing research to detect at-risk students using compiled histories and invocations, such as the "Watwin" algorithm [23] and Repeated Error Density [5]. In conjunction with the compiler history, we will investigate the source code patterns that indicate that a student is struggling. We will use data mining approaches to derive developers' behaviour patterns, similar to Kinnewbrew's research into learners' behaviour. Kinnebrew et al. used interaction trace data from Betty's Brain, which introduced students to a science topic (climate change). They abstracted and labelled different activities into different categories to analyse the data using differential sequence mining, identifying differentially frequent patterns across two groups of sequences [11].

We will investigate using interaction traces of source code to detect struggling students. Source code traces will allow us to examine how the student develops and any patterns in their programming and debugging practices that could indicate that they are struggling. Using source code traces could detect more struggling students than just compiler histories, as source code traces enable us to analyse more than just compiler related issues. An example pattern of a struggling student could be commenting out large chunks of code, potentially followed by uncommenting small portions to figure out which part of the code is causing their problem, which could signify that the student has trouble debugging and locating the problem. Another example may be the number of revisions made to a single code block in a given timeframe, suggesting that they are having trouble solving a logical or syntactic mistake, depending on the compilation result.

We will use the patterns we found to detect when a student is struggling and give them a hint.

In order to give students a next-step hint, we will integrate the ideas of the Continuous Hint Factory (CHF) [18] and Evremov et al.'s research on using reinforcement learning to generate next-step hints [8]. We will use CHF as our primary method of producing hints, using Blackbox as the basis of potential candidate solutions. Blackbox contains data from opted-in users of BlueJ [14]. BlueJ is an educational IDE used worldwide by many different institutions at many different education levels. Accompanying BlueJ is a widely used textbook [3] that provides example projects and tasks, of which there are multiple solutions within Blackbox. However, there will be cases where a student is programming a task for the first time, such as new coursework. These cases are where Evremov et al.'s approach comes into effect, the ability to still provide next-step hints for the very first attempts of new programming tasks without the requirement of producing candidate or model solutions.

We will use Blackbox to learn patterns of struggling students and for elements of hint generation, as Blackbox contains both compile history and source code changes. To implement the ideas discussed in this section, we will create a plugin for BlueJ, which will run continuously in the background analysing the students' compile history and source code traces, comparing them to any patterns we have learnt from Blackbox. The plugin will include a "human-in-the-loop" process, enabling the user to give feedback to the system if the detection of their struggle was correct or if the hint helped solve their issue.

The following are the proposed steps that the system will follow:

1. While the student develops and compiles, the system will analyse the source code and compile histories for defined patterns.
2. If a pattern is detected, ask the user if they would like a hint.
   (a) If yes, give the student a hint and ask them to rate the quality and usefulness of the hint.
   (b) If no, log that they did not want a hint. If enough students give feedback that they do not want a hint for this specific pattern, flag it for investigation.
3. If the same pattern on the same part of the code is detected, repeat steps 2a and 2b, but with a more specific hint.
4. Repeat the steps above while the student develops.

We will conduct a series of trials to evaluate our proposed tool to determine if our system correctly detects struggling students and if the hints delivered are meaningful. Our trials could include asking students of various skills levels to complete a set of programming tasks of increasing difficulty, with and without our proposed tool. After they complete the programming tasks, we could interview the students to determine if they thought they were offered a hint at the correct time and if the hint given was helpful. In addition to interviewing students, we may ask tutors for their feedback on whether the suggested method provided timely assistance and whether the hints assisted students in learning.

5. RISKS OF NEXT-STEP HINTS

This project aims to generate hints that will trigger a learning effect. We will have to hypothesise what area of the programming task the student is struggling with in order to generate a hint that will increase the students' understanding in that area. Using patterns of student development behaviour with next-step hint techniques could indicate which area the student is struggling with and give them a hint that increases their understanding and help them continue with their programming task.

While employing next-step hints might help students learn, there is a risk that using incrementally more specific hints can harm learning by giving students answers without teaching them how to improve. Students who abuse the hint system in this way exacerbate the risk of them not learning how to improve. We will investigate different methods to decrease the possibility of abuse in traditional hint systems, including providing a hint only when the system recognises that the student is having difficulty. However, this may not eliminate the risk, as some students will learn how to manipulate erroneous detections in order to obtain consecutive suggestions that will lead to the correct answer.

These risks lead to the pedagogical question: *Should the system give the student the answer if they have fundamental misunderstandings?* While undertaking this project, we will aim to minimise the risks discussed in this section and maximise the learning effect of the generated hints.

6. PROPOSED CONTRIBUTIONS

The proposed contributions of the above research will include:

- A method that can detect a struggling student using compiler error history and source code changes.
- Investigate when is the optimal time to hint after detecting a struggling student.
- A tool for detecting struggling students and giving them an appropriate next-step hint at the right time.

These contributions will benefit the Computer Science Education community by offering a tool to support students inside and outside formal education settings and alleviate some growing pressure on tutors of large classes.

7. CONCLUSION

We proposed the development of a tool that can detect when a programmer, mainly a novice programmer, is struggling and provide them with a next-step hint. We will look into how various pattern recognition techniques and past work on metrics can be used to identify a struggling student based on compiler and code change histories.

8. REFERENCES

[1] V. Aleven, I. Roll, B. M. Mclaren, and K. R. Koedinger. Help helps, but only so much: Research on help seeking with intelligent tutoring systems. *Int J Artif Intell Educ*, 26:205–223, 2016.

[2] N. Ayewah, D. Hovemeyer, D. J. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25:22–29, 2008.

[3] D. J. Barnes and M. Kölling. *Objects First With Java - A Practical Introduction Using BlueJ*. Pearson, sixth edition, 2016.

[4] B. A. Becker. A new metric to quantify repeated compiler errors for novice programmers. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, 11-13-July-2016:296–301, 7 2016.

[5] B. A. Becker. A new metric to quantify repeated compiler errors for novice programmers. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, 11-13-July-2016:296–301, 7 2016.

[6] N. C. C. Brown, M. Kölling, D. Mccall, and I. Utting. Blackbox: A large scale repository of novice programmers' activity. *Proceedings of the 45th ACM technical symposium on Computer science education*, 2014.

[7] O. Burn. Checkstyle. *http://checkstyle. sourceforge. net/*, 2003.

[8] A. Efremov, A. Ghosh, and A. Singla. Zero-shot learning of hint policy via reinforcement learning and program synthesis. *International Conference on Educational Data Mining (EDM)*, 2020.

[9] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. *ICER 2006 - Proceedings of the 2nd International Computing Education Research Workshop*, 2006:73–84, 2006.

[10] M. C. Jadud, M. Mercedes, T. Rodrigo, E. Tabanao, M. Beatriz, and E. Lahoz. Analyzing online protocols to characterize novice java programmers. *Philippine Journal of Science*, 138:177–190, 2009.

[11] J. S. Kinnebrew, K. M. Loretz, and G. Biswas. A contextualized, differential sequence mining method to derive students' learning behavior patterns. *Journal of Educational Data Mining*, 5:190–219, 2013.

[12] S. Krusche, L. M. Reimer, B. Bruegge, and N. von Frankenberg. An interactive learning method to engage students in modeling. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training*, 20, 2020.

[13] M. Kölling. The design of an object-oriented environment and language for teaching. pages 145–146, 1999.

[14] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The bluej system and its pedagogy. *http://dx.doi.org/10.1076/csed.13.4.249.17496*, 21:249–268, 2010.

[15] F. Lee. When the going gets tough, do the tough ask for help? help seeking and power motivation in organizations. *Organizational Behavior and Human Decision Processes*, 72:336–363, 12 1997.

[16] D. McCall and M. Kölling. A new look at novice programmer errors. *ACM Transactions on Computing Education (TOCE)*, 19, 7 2019.

[17] F. Obermüller, U. Heuer, and G. Fraser. Guiding next-step hint generation using automated tests. *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, 2021.

[18] B. Paaßen, B. Hammer, T. W. Price, T. Barnes, S. Gross, and N. Pinkwart. The continuous hint factory - providing hints in vast and sparsely populated edit distance spaces. 8 2017.

[19] K. Rivers and K. R. Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education 2015 27:1*, 27:37–64, 10 2015.

[20] M. M. T. Rodrigo and R. S. J. Baker. Coarse-grained detection of student frustration in an introductory programming course. *ICER'09 - Proceedings of the 2009 ACM Workshop on International Computing Education Research*, pages 75–79, 2009.

[21] J. Spacco, P. Denny, B. Richards, D. Babcock, D. Hovemeyer, J. Moscola, and R. Duvall. Analyzing student work patterns using programming exercise data. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education.*

[22] V. J. Traver. On compiler error messages: What they say and what they mean. *Advances in Human-Computer Interaction*, 2010, 2010.

[23] C. Watson, F. W. Li, and J. L. Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. *Proceedings - 2013 IEEE 13th International Conference on Advanced Learning Technologies, ICALT 2013*, pages 319–323, 2013.