Using Student Trace Logs To Determine Meaningful Progress and Struggle During Programming Problem Solving

Yihuan Dong
North Carolina State University
ydong2@ncsu.edu

Samiha Marwan
North Carolina State Universtiy
samarwan@ncsu.edu

Preya Shabrina
North Carolina State Universtiy
pshabri@ncsu.edu

Thomas Price
North Carolina State Universtiy
twprice@ncsu.edu

Tiffany Barnes
North Carolina State Universtiy
twprice@ncsu.edu

ABSTRACT
Over the years, researchers have studied novice programming behaviors when doing assignments and projects to identify struggling students. Much of these efforts focused on using student programming and interaction features to predict student success at a course level. While these methods are effective at early detection of struggling students in the long run, there is also a need to identify struggling students during an assignment so that we can provide proactive intervention to prevent unproductive struggle and frustration. This work proposes a data-driven method that uses student trace logs to identify struggling moments during a programming assignment and determine the appropriate time for an intervention. We define a struggling moment as not achieving significant progress within a certain amount of time, relative to the amount of progress made and time taken in a sample student dataset. The paper describes how we determine significant progress and a time threshold for struggling students. We validated our algorithm's classification of struggling and progressing moments with experts rating whether they believe an intervention is needed for a sample of 20% of the dataset. The result shows that our automatic struggle detection method can accurately detect struggling students with less than 2 minutes of work with over 77% estimated accuracy. Our work contributes significantly to building proactive immediate support features for intelligent programming environments.

Keywords
block-based programming, open-ended assignment, struggling, progressing, data-driven method, trace log

Yihuan Dong, Samiha Marwan, Preya Shabrina, Tiffany Barnes and Thomas Price "Using Student Trace Logs To Determine Meaningful Progress and Struggle During Programming Problem Solving". 2021. In: Proceedings of The 14th International Conference on Educational Data Mining (EDM21). International Educational Data Mining Society, 439-445. https://educationaldatamining.org/edm2021/
EDM '21 June 29 - July 02 2021, Paris, France

1. INTRODUCTION AND BACKGROUND
Computer programming is a challenging topic for novices. As a result, there has been an increasing interest in the early detection of struggling students for proactive intervention to reduce dropout rates and improve student learning in programming courses.

Researchers have collected and studied student problem-solving trace data during programming assignments from various perspectives. Most of this effort has focused on analyzing student programming actions to reveal their behavioral traits and programming patterns. This research typically uses manual inspection of the trace logs [4, 16, 9] or applies machine learning models [7, 3, 1] to categorize student behaviors and discuss the characteristics of each category and their potential impact on or relationship with student performance. The contribution of these studies usually lies in helping educators understand novice learning processes and promote positive behaviors. Another strand of research that analyzed student trace data focuses on student compilation behaviors [11, 19, 2] and syntax errors and bugs [21, 10] in student traces. This research used statistical inferences, machine learning methods, and visualization techniques to explore the relationship between specific patterns and student success and identify novice students' common mistakes. Some of these patterns are helpful for identifying students struggling with certain concepts.

However, we found that there has not been enough research that uses student trace log to model their progress and identify struggling moments during programming assignments. One major application of struggling detection is providing proactive hints in intelligent tutoring systems (ITS), as previous research has shown that novices, especially those with low prior knowledge or experience, may not request on-demand hints even when they need them [18]. Prior work identifying struggling students in traces generally focused on early detection of struggling students determined by the assignment outcome [12, 5, 6] and are not suitable for identifying struggle during programming assignments.

In this work, we propose a novel data-driven approach to identify struggling moments from student trace data. We describe how we adopted the SourceCheck algorithm to model *student progress* during open-ended programming assignments and how to identify *progressing moments* and *struggling moments* from student progress. We used human experts to evaluate the progressing moments and struggling moments classified by our method. Our initial result shows that human experts had a decent agreement with our algorithm and that it is possible to determine if a student is struggling during programming assignments within two minutes. To the best of our knowledge, this is the first attempt to use a data-driven progress measurement to identify struggling students. We also discuss how our method may generalize to other domains that meet certain requirements.

2. DETERMINE INTERVENTION TIMES

In this work, we consider a student to be *struggling* during problem-solving when they could not make *enough progress* within a *typical amount of time*. As such, to determine the proper time to provide proactive intervention, we need to investigate: 1) how to measure student progress *during* solving a programming assignment, 2) what constitutes significant progress, and 3) how much time it typically takes a student to make significant progress, and finally, 4) what is the appropriate *time threshold* beyond which we consider the student is struggling and need help. This section describes how our algorithm models student progress and identifies potential progressing and struggling moments through these four steps.

2.1 Dataset

We analyzed a dataset collected from an introductory programming course for non-computer science major students in Fall 2017. Students learned to program by doing a series of open-ended programming assignments adapted from the BJC curriculum [8] in a block-based programming environment called Snap*!*.

We extended the Snap*!* environment to record all students' programming actions into *traces*. Each trace, identified by a project id, contains all actions a student performed during an assignment (e.g., creating or deleting a block) with timestamps. There are two types of actions in the traces, *non-coding* actions, and *coding* actions. Non-coding actions do not change the program scripts, for example, searching for blocks and running the program. Coding actions change the abstract syntax tree of program scripts, such as creating variables, creating custom blocks, and reordering blocks. Alongside every coding action, our Snap*!* environment also saves a code *snapshot* after the action, allowing us to reconstruct the steps a student took to build their final code and analyze their coding progress.

In this work, we focused on analyzing two assignments, Squiral and Guessing Game. Squiral is a homework assignment that asks students to create a procedure that draws a square-like spiral with a certain number of rotations specified by an input parameter. A correct Squiral solution typically contains 7 to 14 lines of code. Guessing Game (GG) is an in-class assignment that requires students to create a game that greets the players by their names, asks the player to guess the secret number, and tells the player if their guesses were too high too low until they guessed correctly. A typical Guessing Game solution contains 14 to 18 lines of code. These two assignments allow us to explore how well our method identifies students' struggling moments in assignments with different time constraints. Table 1 shows the descriptive statistics of the traces analyzed in the two assignments. We preprocessed the traces to remove any idle time of more than five minutes, during which the student did not perform any action.

Table 1: Descriptive statistics of the trace logs and grades of the two assignments.

	Traces	Rows	Time on Task	Avg Grade
Squiral	45	25160	29.6m	9.8/12
GG	59	22744	30.5m	11.7/12

2.2 Define Progress

The first step to identify struggling is to measure student progress in the assignment. Previous work used code compilation results [11, 19, 2], students' programming behavior patterns [7], and features completion [15, 13] to monitor student progress in an assignment. While these criteria are reliable indicators of how many assignment requirements the students have met, they did not use the student traces' full potential to identify struggling students at an action-level granularity during the assignment.

We adopted the *SourceCheck* algorithm [17] to measure student progress during an assignment. *SourceCheck* was initially designed as a hint generation algorithm to provide on-demand, next-step hints to help students move towards the closest correct solution to the student's current code. When generating hints, emphSourceCheck first compares the student's current code snapshot with a list of correct solutions (usually collected from past student data) and generates *mapping costs* from the student's code snapshot to each of the correct solutions. These mapping cost values represent how similar the correct solutions are to the student code – the more similar a correct solution is to the student's code, the lower the mapping cost is. *SourceCheck* picks the correct solution with the lowest mapping cost as the *closest correct solution* and generates next-step hints to move the student to that solution.

We adapted the mapping cost into a **similarity score** by reversing the mapping cost such that when a student moves closer to the closest correct solution, the mapping cost decreases, and the similarity score increases. One novel aspect of this paper is how we use the similarity score to measure student progress in the two assignments [1] We calculate a similarity score for every snapshot in a student trace using the *SourceCheck* algorithm. We define a snapshot's *progress* in an assignment as the similarity score *difference* between the *current* snapshot and its *previous* snapshot. As such, at a particular snapshot, we say a student is making a **positive progress** if the similarity score difference is positive and a **negative progress** if the similarity score difference is negative.

[1]Note that we assume a student is moving towards the closest correct solution at any given snapshot. Students do not know the prior student solutions used by *SourceCheck*, and we have no ground truth to identify what strategy a student may be using for their assignments.

We can visualize a student's progress in an assignment by plotting the similarity scores for each snapshot against the cumulative active time, shown in blue in Figure 1. The red line in Figure 1 represents what we call "absolute progress", which we define in the next section. The blue dots in Figure 1 represent the similarity score of every snapshot in a *Squiral* trace, and the blue line represents the progress (similarity change between consecutive snapshots) over time. Figure 1 demonstrates that the student made steady *positive progress* in the first eight minutes. Then, the student had a reduced similarity score for about four minutes, tearing the code apart trying to complete an objective of the assignment. Afterward, the student continued to make rapid positive progress until 14 minutes, stopped progressing for around three minutes, and finally reached their final submission at around 17 minutes.

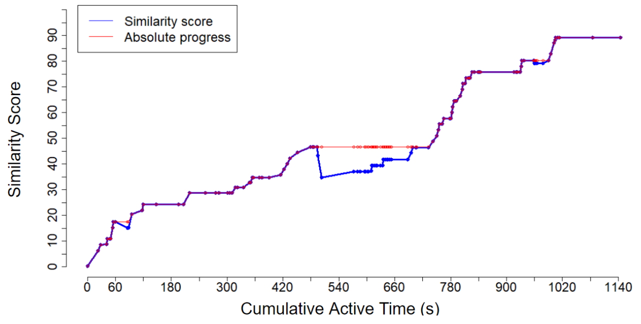

Figure 1: Similarity score (blue) and absolute progress (red) change over time in one student trace of Squiral.

2.3 Determine Significant Progress

Through inspecting multiple students' traces and comparing them with their corresponding progress plot (e.g., Figure 1), we found that not all positive progress represents a significant change to the program. Some minor similarity score increases were due to reordering code that does not change the code semantics but slightly reduces the mapping cost. This observation means that using *any* amount of similarity score increase for making progress *may not* be sufficient. Thus, it is important to determine how much similarity score increase can be considered *significant progress*.

To determine significant progress, we first define *absolute progress* for a student s_j at time t_i. We define $S_{max}(s_j, t_i)$ to be the maximum similarity score achieved by student s_j in an assignment between time t_0 and t_i, $S_{max}(s_j, t_i) = max^i_{k=0} S(s_j, t_k)$. We then define **absolute progress** as the difference in the maximum similarity scores between t_i and the previous snapshot time t_{i-1}, $P_{absolute}(s_j, t_i) = max(S_{max}(s_j, t_i) - S_{max}(s_j, t_{i-1}), 0)$. To visualize absolute progress, we plot the highest similarity scores achieved since the beginning of the trace (S_{max}), as shown in red in Figure 1. The absolute progress is *positive* whenever S_{max} increases between two consecutive snapshots.

We then calculated and sorted the absolute progress values from all the student traces for each assignment in increasing order and plotted all *positive absolute progress values* by percentile (using the quantile function in R), as shown in Figure 2. We used the 25th percentile of absolute progress values as the threshold for making **significant progress**. This choice was also used in another work identifying struggling students in a MOOC programming assignments [20]. The intuition is that if a student's absolute progress is no more than three-quarters of all the absolute progress, we consider the student is not making enough progress. Figure 2 shows that the significant progress threshold is 1.25 for Squiral and 1.5 for Guessing Game.

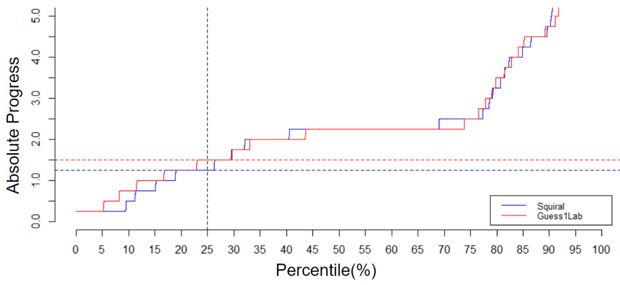

Figure 2: Positive absolute progress in all traces by percentile

2.4 Determine Typical Time

Now that we have determined the significant progress for each assignment, the next step to identify struggling moments is to extract the *typical time* for students to make significant progress. To do this, we first split all the traces into *code chunks* whenever the student makes significant progress–each code chunk contains multiple snapshots. This gave us 648 code chunks from Squiral and 1207 code chunks from Guessing Game. Then, we calculate the elapsed time between the first and the last snapshot in each code chunk and organize the elapsed times in ascending order. We plot the ordered elapsed times in Figure 3 where the y-axis is the elapsed time, and the x-axis is the percentile of the elapsed time distribution. The green line in Figure 3 marks the third quartile of time used to make significant progress. Note that the elapsed time to make significant progress grows almost exponentially after the third quartile. Therefore, we chose to use the third quartile as the cutoff for the *typical time* to make significant progress. The third-quartile time (dashed green line) in Figure 3 intersects with the Squiral progress (solid blue line) at 105 seconds and intersects with the Guessing Game significant progress (solid red line) at 85 seconds. We use these times as the typical time for the students to make significant progress in Squiral and Guessing Game. There are several dashed lines on Figure 3, which we explain in section 3.

2.5 Determine Progressing and Struggling Moments

The last step of this process is to use the typical time to make significant progress in identifying progressing and struggling moments for each assignment. To do this, we took all the code chunks generated in the third step and divided them into two groups, *struggling moments* and *progressing moments*. Recall that we define a student as struggling if the student does not make enough progress within a typical amount of time. Therefore, struggling moments are defined to be code chunks that have elapsed time greater than our struggling time threshold (75th percentile of time for significant progress), meaning that in this code chunk, even though students spent a long time, they did not make

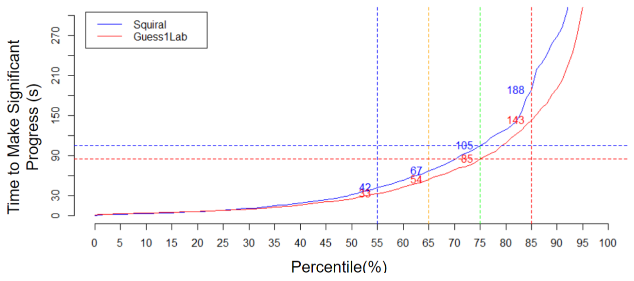

Figure 3: The elapsed time for all code chunks to make significant progress by percentile. The green, yellow, blue and red dash lines mark the proposed, earlier, even earlier, and later intervention times, respectively.

significant progress. Conversely, progressing moments are defined to be code chunks that took equal or less time than the 75th percentile of the typical time to make significant progress. Table 2 summarizes the significant progress, typical time for significant progress, and the number of code chunks generated for each assignment.

Table 2: A summary of products generated by our algorithm for the two assignments.

	Squiral	GG
Sig. Progress	1.25	1.5
Typical Time for Sig. Prog.	105.3s	84.5s
# Code Chunks	648	1207
# (%) Struggling Moments	131(20.2%)	269(22.3%)
# (%) Progressing Moments	517 (79.8%)	938(77.7%)

3. EVALUATION

Our evaluation is driven by the following research questions:

RQ1: To what extent the human experts agree with the progressing moments and struggling moments identified by our method?
RQ2: What are the common causes that the human experts do not agree with the progressing and struggling moments identified by our method?

We invited three expert raters, all included as authors, to participate in the rating of whether an intervention is needed for given code chunk samples. All three experts are computer science graduate students, two with extensive experience analyzing Guessing Game traces, and all three with extensive experience analyzing Squiral traces for research.

We first created the *struggling rating sample dataset* by picking a random struggling moment from each trace until the rating dataset contains 20% of all struggling moments for each assignment. Then, we created the *progressing rating sample dataset* by selecting the progressing moments immediately before each struggling moment in the struggling rating sample dataset. Finally, three redundant progressing moments were excluded from the progressing rating dataset because they were shared by consecutive struggling moments in the same trace. As a result, our rating dataset ended up with 29 struggling moments and 29 progressing moments for Squiral (out of 131), and 57 struggling moments and 54 progressing moments for Guessing Game (out of 269).

The experts were told to imagine themselves as TAs when rating. They used a customized interface that allowed them to visually step through the students' code changes in the rating moments to decide whether an intervention is needed. The experts used the time elapsed between actions and the type of the actions to inform their rating decision. They were asked to avoid using *hindsight*, which means that they should not justify their decision for intervening at an earlier time using student's later actions.

When rating the struggling moments, to make it easy to compare expert ratings, the experts were given five intervention timing options. The five options corresponds to potential intervention times marked by the colored vertical lines shown in Figure 3, which correspond to suggesting an intervention at time: blue or before (quantile(0.55)), yellow (quantile(0.65)), green (quantile(0.75), the typical time to make significant progress), red (quantile(0.85)), or "not now" (after red, or never). We chose these candidate percentiles to gain insights into expert preferences of intervention times for future analysis. For struggling moments, experts were shown the code changes from the start until the last action before the 85th percentile time for significant progress (red) to decide when an intervention would be most appropriate, or "not now." For progressing moments, experts were shown the entire progressing moment (all code changes within the time period where significant progress was achieved) and asked the expert to rate whether the moment "needs intervention" or "not now." Aside from rating the struggling and progressing moments, experts were also encouraged to take notes on why they believed an intervention is needed whenever they rate a sample as "needs intervention" for both the struggling and the progressing rating sample datasets. These notes will help us understand the experts' point of view when inspecting the disagreements between the experts and our algorithm.

Before formal rating, the experts practiced rating on a training dataset by immediately discuss their ratings after rating each sample until they were comfortable with the rating process. Then, the experts rated the struggling moments independently in three rounds, each round rating a third of the samples in the dataset. After each round, the experts gathered and discussed the differences in their ratings to share perspectives and resolve disagreements caused by oversight. We did not require the experts to reach a complete consensus on the rating because experts sometimes have different opinions on handling specific situations. Finally, after rating the struggling sample dataset, the experts independently rated the progressing dataset and were asked to check disagreements to correct rating errors caused by oversight.

4. ANALYSIS AND RESULT

To evaluate our RQ1 considering to what extent the human experts agree with the struggling moments and progressing moments identified by our algorithm, in this analysis, we merged the five rating options the experts used in the rating of the struggling chunks dataset into two options, "need intervention" and "not now." Specifically, we merged "at blue or before," "at yellow," and "at green" options into "need intervention" and merged the "at red" and "not now" options into "not now." These two option labels are identical to those used when rating the progressing moments to directly compare how much the experts agreed with the generated struggling and progressing moments. Splitting and merging the options at green where the proposed time is at allows us to determine if our proposed typical time to make significant progress is appropriate and enough for the experts to decide whether they believe an intervention is needed.

Our analysis of expert ratings focuses on their ratings after discussion because those ratings are after the effort to remove personal error or bias. However, we report the inter-rater reliability (IRR), calculated with Fleiss Kappa, both before and after discussions to demonstrate the impact of the discussions. To determine how well the experts agree with the algorithm, we turned each expert's ratings into binary scores of 0s and 1s depending on whether the expert rating agrees with the rating dataset. Specifically, for struggling moments, the score is 1 if the expert rated "need intervention" and the score is 0 otherwise. For progressing moments, on the other hand, the score is 0 if the expert rated "need intervention" and the score is 1 otherwise. We then take the average of all expert ratings to calculate a *combined rating score* for each sample such that a combined rating of 0 or 1 represents that the experts reached an agreement and anything in between 0 and 1 means the two or three experts had different opinions, even after discussion, on whether the student was making progress or struggling. Finally, we calculate the *agreement rate* for each rating dataset by summing up all the combined rating scores in that rating dataset and divide the sum by the total number of samples in the rating dataset for that assignment.

4.1 RQ1: Expert Agreement

Table 3 shows the percentage of progressing moments and struggling moments that the expert ratings agreed with the algorithm after discussion, as well as the corresponding inter-rater reliability before and after discussions. Looking at the agreement rate between the expert and our algorithm, we see that for both assignments, over 77% of the time, the human experts agreed that an intervention was needed when the algorithm determined the student was struggling. Over 85% of the time, the human experts agreed that an intervention was not needed when the algorithm determined the student was making progress. This suggests that our method was able to identify struggling moments and progressing moments from the trace data with decent accuracy.

Table 3: Human expert agreement with the algorithm identified struggling moments and progressing moments

	Struggling Rating			Progressing Rating		
	N	Need Interv.	IRR (B/A)	N	Not Now	IRR
Squiral (3 raters)	29	77.0%	0.805^{**} / 0.847^{**}	29	85.2%	0.853^{**}
GG (2 raters)	57	83.3%	0.539^{**} / 0.646^{**}	54	85.1%	0.819^{**}

Looking at the expert agreement with each other, we found that the experts had excellent inter-rater reliability for progressing moments ratings on both assignments and for struggling moments ratings on Squiral. However, the experts were only able to reach a moderate agreement, even after discussion, for struggling moments on Guessing Game. We explored reasons that might cause experts to disagree with each other by manually inspecting the traces and their notes. We found that in four out of five struggling moments that the experts disagreed, students did not have errors in their snapshots but only performed less than six actions, which is way below the average number of 12 actions of all rated struggling moment samples. In such cases, one expert prefers to hold off on any intervention until seeing more student actions, whereas the other expert believes that the student needed a nudge telling them what they should do next. We did not see such a case in Squiral because all the rated struggling moments with few student actions had relatively apparent flaws in the student codes that warrant intervention. We will talk more about this in the discussion section.

4.2 RQ2: Common Causes of Disagreement

We manually investigated the ratings on which the human tutors and our algorithm disagreed. We present some common causes of disagreement for struggling moments and progressing moments, respectively.

Disagreement in Struggling Moments

Solution Matching: A decreased similarity score does not always mean the student is making negative progress. Due to characteristics of the SourceCheck algorithm, in some cases, a reduced similarity score can also be caused by the SourceCheck algorithm mapping the student's previous snapshot and the current snapshot to different correct solutions because the student added a particular code block. In such cases, if the student similarity score does not surpass the maximum similarity score since the beginning within an expected amount of time, our algorithm will determine the student is struggling, even though the student is making progress (having an increasing similarity score). However, the student may just be using a different approach to solve the problem from the expert's perspective.

Few Coding Actions: Since our algorithm focused on students' progress over time, sometimes students might not have taken enough actions for experts to determine if the student is struggling or not. There are several possible reasons why students have few actions, including trying to reason with their code, running their code, evaluating the result, or being off task. Disagreement in this situation did not only occur between experts and our algorithm but also happened between expert raters, causing a relatively lower inter-rater reliability of struggling moments in the Guessing Game assignment.

Disagreement in Progressing Moments

Logic Errors: The experts are good at catching critical logic errors in the student code and tend to intervene if there is a critical logic error in student code that might prevent them from completing the feature they are working on. For example, in Guessing Game, both experts decided to intervene when a student set the secret number to a boolean value and was trying to use the secret number to give player feedback on their guesses because the student would not be able to test the feedback feature properly without correctly setting the secret number first. In contrast, our algorithm considered that the student was making progress because the students added blocks seen in the correct solutions.

Human Factors: Sometimes, when deciding on intervention, the experts assess the natural language in students' code to infer information about student intention in a way that is not possible for our algorithm. For example, in the Guessing Game assignment, experts pointed out that an intervention is needed when several students used an if/else block instead of combining the say block and the join words block to greet the players. However, since the if/else block was used for giving player feedback in many correct solutions, our algorithm determined the student was making progress, despite the student using the if/else block incorrectly for a different purpose.

5. DISCUSSION

RQ1: *To what degree do the experts agree with the algorithm on struggling and progressing moments?* For Squiral, the experts agreed that an intervention was needed for over 77% of the struggling moment samples and agreed that no intervention was needed for over 83% of the progressing moment samples. However, the experts have a relatively lower disagreement with each other in Guessing Game ratings, caused by different opinions on whether an intervention is needed when a student has few actions within the time frame. Since the expert raters were not pedagogical experts and had experience level on par with experienced teaching assistants (TAs), we do not know how an experienced instructor would react to this or other scenarios. Nevertheless, our results show that our method of identifying struggling moments by measuring whether a student could make significant progress within a typical amount of time aligns well with opinions from our experts who have at least as much experience as highly qualified TAs. We believe this shows great potential to determine when to give proactive interventions to students in intelligent tutoring systems for programming.

RQ2: *What are the common cases that our algorithm does not handle well?* We listed four common causes that led to disagreements between the experts and our algorithm. The first cause, "solution matching", rests in the adoption of SourceCheck as a progress measure. Since the SourceCheck compares student solutions to multiple model solutions, sometimes, adding a code block to a snapshot could cause SourceCheck to pick a different solution as the closest solution with a lower similarity score. This calls for investigation on how we may wish to smooth out the abrupt progress change when mapped to different correct solutions between consecutive snapshots. "Logic errors" and "human factors" are problems that are difficult to solve by using distance-based similarity measures alone, since similarity measures merely compare the mapping cost between two pieces of code and do not assess the semantics in natural language or programming logic. Thus, these two causes may require obtaining knowledge from other types of analysis to identify. Previous research has used compilation error [11, 19, 2], and feature detection [14] to detect if there are specific errors and determine if the student is struggling. Incorporating these methods could provide richer information in the decision-making for struggling moments. Lastly, some expert disagreements with the algorithm in struggling moments were caused by dealing with "few actions" within the time frame of the code chunks. Our inter-rater reliability for Guessing Game shows that even experts had a hard time agreeing on if an intervention is needed in this case. There seem to be multiple factors that may affect the expert decision, including if there are errors in the snapshot, the type of actions performed, and the assignment requirements that are completed and incompleted. Potential solutions to this problem may include consulting experienced teachers and incorporating sequence pattern analysis [7].

It's worth pointing out one important contribution of this work is that our method of using a data-driven approach to identify struggling moments has the potential to be generalized into any other domain that meets the following criteria.

1.Student Performance: Good student performance on the task, meaning that the majority of the students achieve correct or mostly-correct final solutions.
2.Trace log data: having time-stamped trace logs that documents students' snapshots during an assignment.
3.Progress measure: A score, or a combination of correct solutions and a distance metric between snapshots and correct solutions, as we devised from SourceCheck.

There are two major ways our method can benefit future research. First, our method of identifying progressing and struggling moments provides a new way for researchers to study novice problem-solving behaviors and identify common misconceptions. Second, the significant progress value and typical time to make significant progress can be incorporated into intelligent tutoring systems for providing proactive feedback to struggling students.

This work has some clear limitations. First, we only used three expert raters to evaluate the sample result. The expert raters were not pedagogical experts and had experience levels on par with experienced TAs. Hence the evaluation result may be different if rated by experienced instructors. In addition, our work is limited by a small sample size with only two programming assignments. We need further investigation to determine if our result holds for assignments with even more varied complexity or in other problem-solving contexts.

6. CONCLUSION

This work presented a novel, data-driven approach to use a similarity measure to model student progress in programming assignments and identify progressing and struggling moments from trace log data. To evaluate the performance of our algorithm, we asked human experts to evaluate a sample of 20% of the algorithm-identified progressing and struggling moments from trace logs from students solving two programming assignments and rated if the experts agree with the algorithm. Our result shows that the expert agreed with over 77% of the struggling moments and over 83% of the progressing moments, which shows great potential. Our algorithm can be generalized to different domains if they have good student performance, trace log data, and a progress measure for in-progress student solution attempts.

7. REFERENCES

[1] A. Allevato and S. H. Edwards. Discovering patterns in student activity on programming assignments. In *ASEE Southeastern Section Annual Conference and Meeting*, 2010.

[2] A. Altadmri and N. C. Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 522–527, 2015.

[3] P. Blikstein, M. Worsley, C. Piech, M. Sahami, S. Cooper, and D. Koller. Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences*, 23(4):561–599, 2014.

[4] Y. Dong, S. Marwan, V. Catete, T. Price, and T. Barnes. Defining tinkering behavior in open-ended block-based programming assignments. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 1204–1210, 2019.

[5] G. Dyke. Which aspects of novice programmers' usage of an ide predict learning outcomes. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 505–510, 2011.

[6] A. Estey, H. Keuning, and Y. Coady. Automatically classifying students in need of support by detecting changes in programming behaviour. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 189–194, 2017.

[7] G. Gao, S. Marwan, and T. W. Price. Early performance prediction using interpretable patterns in programming process data. *arXiv preprint arXiv:2102.05765*, 2021.

[8] D. Garcia, B. Harvey, and T. Barnes. The beauty and joy of computing. *ACM Inroads*, 6(4):71–79, 2015.

[9] R. Hosseini, A. Vihavainen, and P. Brusilovsky. Exploring problem solving paths in a java programming course. 2014.

[10] M. C. Jadud. *An exploration of novice compilation behaviour in BlueJ*. PhD thesis, University of Kent, 2006.

[11] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*, pages 73–84, 2006.

[12] Y. Mao. One minute is enough: Early prediction of student success and event-level difficulty during novice programming tasks. In *In: Proceedings of the 12th International Conference on Educational Data Mining (EDM 2019)*, 2019.

[13] S. Marwan, G. Gao, S. Fisk, T. W. Price, and T. Barnes. Adaptive immediate feedback can improve novice programming engagement and intention to persist in computer science. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, pages 194–203, 2020.

[14] S. Marwan, G. Gao, S. Fisk, T. W. Price, and T. Barnes. Adaptive immediate feedback can improve novice programming engagement and intention to persist in computer science. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, pages 194–203, 2020.

[15] S. Marwan, T. W. Price, M. Chi, and T. Barnes. Immediate data-driven positive feedback increases engagement on programming homework for novices. 2020.

[16] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari. Habits of programming in scratch. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, pages 168–172, 2011.

[17] T. Price, R. Zhi, and T. Barnes. Evaluation of a data-driven feedback algorithm for open-ended programming. *International Educational Data Mining Society*, 2017.

[18] T. W. Price, R. Zhi, and T. Barnes. Hint generation under uncertainty: The effect of hint quality on help-seeking behavior. In *International conference on artificial intelligence in education*, pages 311–322. Springer, 2017.

[19] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud. Predicting at-risk novice java programmers through the analysis of online protocols. In *Proceedings of the seventh international workshop on Computing education research*, pages 85–92, 2011.

[20] R. Teusner, T. Hille, and T. Staubitz. Effects of automated interventions in programming assignments: evidence from a field experiment. In *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*, pages 1–10, 2018.

[21] M. N. C. Vee, B. Meyer, and K. L. Mannock. Empirical study of novice errors and error paths in objectoriented programming. In *Proceedings of the 7th Annual HEA-ICS Conference*, 2006.