Cluster-Based Analysis of Novice Coding Misconceptions in Block-Based Programming

Andrew Emerson[1], Andy Smith[1], Fernando J. Rodríguez[2], Eric N. Wiebe[1], Bradford W. Mott[1], Kristy Elizabeth Boyer[2], James C. Lester[1]
[1]North Carolina State University, Raleigh, North Carolina
[1]{ajemerso, pmsmith4, wiebe, bwmott, lester}@ncsu.edu
[2]University of Florida, Gainesville, Florida
[2]{fjrodriguez, keboyer}@ufl.edu

ABSTRACT

Recent years have seen an increasing interest in identifying common student misconceptions during introductory programming. In a parallel development, block-based programming environments for novice programmers have grown in popularity, especially in introductory courses. While these environments eliminate many syntax-related errors faced by novice programmers, there has been limited work that investigates the types of misconceptions students might exhibit in these environments. Developing a better understanding of these misconceptions will enable these programming environments and instructors to more effectively tailor feedback to students, such as prompts and hints, when they face challenges. In this paper, we present results from a cluster analysis of student programs from interactions with programming activities in a block-based programming environment for introductory computer science education. Using the interaction data from students' programming activities, we identify three families of student misconceptions and discuss their implications for refinement of the activities as well as design of future activities. We then examine the value of block counts, block sequence counts, and system interaction counts as programming features for clustering block-based programs. These clusters can help researchers identify which students would benefit from feedback or interventions and what kind of feedback provides the most benefit to that particular student.

CCS CONCEPTS

• **Social and professional topics → Computing education;**
• **Applied computing → Education**

SIGCSE '20, March 11–14, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6793-6/20/03...$15.00
https://doi.org/10.1145/3328778.3366924

KEYWORDS

Block-based programming; introductory programming education; cluster analysis

ACM Reference format:

Andrew Emerson, Andy Smith, Fernando J. Rodríguez, Eric N. Wiebe, Bradford W. Mott, Kristy Elizabeth Boyer, and James C. Lester. 2020. Cluster-Based Analysis of Novice Coding Misconceptions in Block-Based Programming. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE'20). ACM, New York, NY, USA, 7 pages.* https://doi.org/10.1145/3328778.3366924

1 Introduction

A long-standing issue in computer science education is the set of challenges many students face in introductory courses. These struggles can take many different forms, making it difficult to identify what intervention would best support learning [16]. While there is a broad range of research on student misconceptions in more traditional text-based programming environments in the computer science education [2, 18], educational data mining [22], and learning analytics communities [5], there has been limited work on novice coding misconceptions in block-based programming.

A growing trend in undergraduate introductory programming courses for both majors and non-majors is the use of block-based programming environments. These environments are designed to address the challenges many students face in introductory courses [23], particularly novice learners who lack substantial prior programming experience. Block-based programming languages offer several potential benefits, including reducing cognitive load [25], assisting with better understanding of program structure [24], and increasing the efficiency in completing coding tasks [13]. However, many learning environments built around block-based programming provide limited support, relying on teaching assistants and instructors who often have limited availability.

In this paper, we present a data-driven exploration of student programs created in PRIME, a block-based programming environment for undergraduate introductory computer science education. We clustered students' unsuccessful attempts at two particular programming activities using Bayesian Gaussian Mixture Models. Qualitative analysis of the resulting clusters

reveals interesting patterns of student misconceptions and provides insight into how different supports, such as focusing on remedial activities or focusing on specific concepts, could be used by instructors and adaptive learning environments to best address the needs of individual students.

2 Related Work

A large body of work has investigated student code analysis, often through the lens of automated assessment and plagiarism detection [21]. This work includes static analysis systems [19] that assess code without running it, dynamic analysis systems [17] that assess code based on the output of defined test cases, and hybrid systems combining both approaches [6]. Additionally, clustering has been used to visualize student programs and programming behaviors [6], to generate formative feedback for students [10, 12], and to provide feedback to instructors [8]. Building on these advances, we apply analogous techniques to block-based programs to investigate how this information can be used to improve instruction rather than automated assessment.

The rise in prevalence of block-based programming environments has also led to a growing body of work focused on analyzing block-based programs. The iSnap environment uses a Contextual Tree Decomposition algorithm to compare student block-based programs to generate next-step hints [14, 15]. Wang et al. [22] used deep learning based approaches to generate representations of student block-based program trajectories to create clusters of learning styles. Swidan et al. [20] analyzed multiple-choice questionnaires created based on examples in the Scratch environment to better understand common misconceptions afforded by the block-based interface. Our work extends this area of research by presenting a novel approach to clustering block-based programs, as well as a qualitative analysis of how these clusters relate to patterns of misunderstanding.

Cluster analysis has been used extensively in disciplines outside of computer science. Min et al. [11] and Akram et al. [1] utilized clustering based on student problem-solving behaviors to inform stealth assessment models. Amershi & Conati [3] used clustering of student behaviors in an interactive lab environment to inform classification models of student behavior. Kardan et al. [9] clustered users based on log-data from a complex simulation environment to gain a better understanding of student behavior and misconceptions. Though none of these environments involved programming, the work reported here builds on the types of features and clustering techniques they introduced, as well as extending it to the post-hoc analysis of the clusters.

3 PRIME Environment

The studies in this paper were conducted within the context of PRIME, an adaptive block-based programming environment designed to support novices as they learn introductory programming concepts. Within the environment, there are twenty programming activities, where each successive activity builds on the skills learned from the previous one. The activities cover the following fundamental CS concepts: input/output, numeric data types, mathematical expressions, variables, iterations (both definite and indefinite), abstraction, functions, parameters, return values, Boolean data types, conditionals, and debugging. Within PRIME, students build their programs utilizing a customized version of Google's Blockly block-based programming plugin [7].

3.1 Study Design

We conducted a study using PRIME at a large university in the southeastern United States. The target courses for this study were two online sections of an introductory course for undergraduate engineering majors. A total of 248 participants logged on to the PRIME system, and we analyzed the programs of the 222 students who attempted at least one activity. These participants had an average age of 18, with 31.5% of them reporting their gender as female. Students who reported being from primarily Non-CS Engineering majors made up 90.3% of the sample; 6.9% reported their major as Computer Science, while the remaining students reported majors such as Math, Agricultural Science, or Undecided. Of these students, 75.8% reported their ethnicity as White, 12.9% as Asian, 3.2% as African American, and 1.2% as Hispanic or Latino.

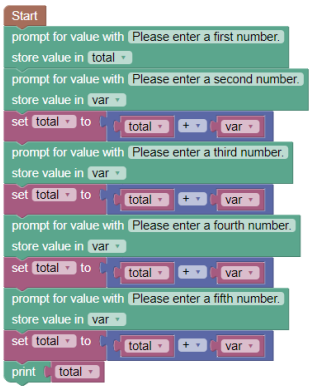

Figure 1: Sample correct student solution to the Accumulator Programming Activity.

In this work, we focused on two activities in the PRIME environment that were of a sufficient data size to conduct clustering and had a significant number of incorrect solution attempts in the activity itself. The first activity, Accumulator, had 35 correct attempts out of 102 total student attempts. The second activity, Repeat Loop, had 38 correct attempts out of 65.

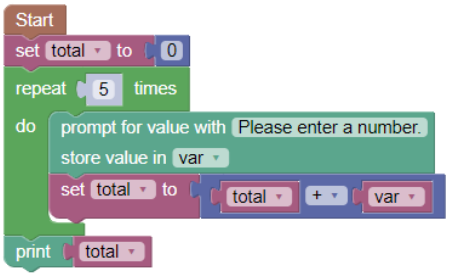

Figure 2: Sample correct student solution to the Repeat Loop Programming Activity.

In the Accumulator activity, students are asked to display the sum of five numbers entered by the user while using only two variables. This exercise builds on previous exercises where students were introduced to concepts such as user input, output, basic mathematical expressions, and variables. In PRIME, user input is accomplished through the *prompt* block. Figure 1 shows a sample solution to the exercise. The following sections identify and describe the clusters derived from the data for the Accumulator Activity.

In the Repeat Loop activity, students built upon the concepts from the Accumulator activity and were tasked with removing the repeated segments of code by using a repeat loop. Figure 2 shows an example of a successful solution to this activity.

4 Clustering Student Programs

In determining a data-driven grouping of student programs, we used a cluster-based algorithm to assign labels to each program. Specifically, we have adopted Bayesian Gaussian Mixture Models [4], which attempt to find the smallest number of clusters that separate the data, in this case, student programs. This approach utilizes an expectation-maximization (EM) approach to select the minimum number of clusters based on the minimal Bayesian Information Criterion (BIC). In choosing the number of clusters, the algorithm will set clusters that are not useful to have a weight of zero, which results in that cluster not being used. For this work, we clustered a static representation of a student's block-based program at the point when they left an activity. We derive this representation of the student's programming workspace such that it can be used in the clustering algorithm and for further analysis.

The features we used in representing the student programs include the following three families of features: *Basic Block Features*, *N-Gram Block Counts*, and *System Interaction Counts*.

- *Basic Block Features*: This family consists of the total number of blocks, the total number of variables used, the total number of blocks attached to the *Start* block, and the total number of blocks not attached to the *Start* block.
- *N-Gram Block Counts*: For each programming activity, we defined five unigram, three bigram, four trigram, and one 4-gram sequence of blocks that are specific to the programming activities and count how many times each sequence occurs in the program.
- *System Interaction Counts*: This family includes the total number of times the student executed their program and the total number of hints requested for the activity.

In analyzing student programs, we extracted *n*-grams consisting of key sub-sequences of blocks that likely exist in correct student programs. We construct a count of these sub-sequences and use this as an individual feature.

Using the described feature set and clustering algorithm, we clustered the student programs. In order to describe differences between the derived clusters, we conducted multiple one-way ANOVA tests between each feature for each cluster. To correct for the family-wise error rate when conducting several statistical tests, we used a Bonferroni correction. These tests were conducted to better describe how the clusters were being separated rather than validating the actual cluster assignments.

4.1 Accumulator Programming Activity

Through the clustering algorithm, we derived a total of three clusters for this activity. We will now describe the three clusters.

4.1.1 Exploration Cluster

Programs from the first cluster in this activity are defined as *Exploration* programs. In this group, sixteen students exhibited traits that we interpret as either a lack of effort or lack of conceptual knowledge to complete the problem. When observing the features that were crucial in separating programs into this cluster, we noticed several that stood out. For example, we observed *Exploration* programs as having fewer blocks on average compared to the other clusters (F=110.17, p<0.0026). Programs in this cluster seemed to also be grouped by a lesser number of *prompt* blocks (F=144.32, p<0.0026). Additionally, programs in this group had fewer sequences of arithmetic operations between the two variables (i.e., summing the inputs) compared to other clusters (F=125.89, p<0.0026).

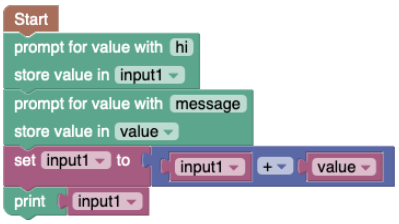

Figure 3: An example of testing smaller ideas without attempting the entire program.

None of the attempts in this group of student programs prompt the user for enough values (i.e., five inputs). The most *prompt* blocks that exist in any student program for this cluster is four. Several students with programs in this group actually did construct the correct pattern of blocks for a portion of the solution, but they did not complete the program. For example, one student (Figure 3) correctly summed the values of two inputs but then failed to continue to ask the user for input. This pattern of testing a smaller idea without completing the entire program was exhibited by six students. Some student programs seemed to even try clever tricks using composition of arithmetic blocks (Figure 4). However, these ideas did not ultimately work because the programs associated with these were left unfinished.

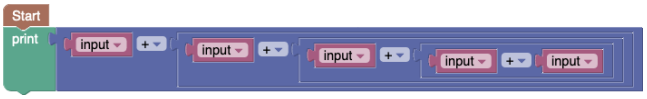

Figure 4: An example of attempting a composition of arithmetic blocks.

The types of issues present in the incomplete student programs of this cluster highlight several implications. First, the issue may be related to motivation or self-efficacy, which could be mitigated by providing real-time encouragement at key moments, inspiring students to persist. However, if the issue is a lack of understanding of key concepts for the activity, there is no common error that occurs in each program to identify a single hint or feedback. It may be practical to flag students whose programs fall into this cluster, allowing the instructor to support the student with finer-grained instruction.

4.1.2 Near Miss Cluster

This cluster of student programs consisted of forty-two total programs, which is the largest group. The programs in this group are best described as *Near Misses*. In general, these students seemed to have assembled the largest number of blocks and put in a significant amount of effort towards completing the activity. Programs in this cluster have a higher frequency of arithmetic sequences (i.e., summing the inputs) when compared to the other clusters (F=125.89, p<0.0026). In addition, the programs in this cluster were observed to have a higher frequency of the sequence *prompt*-to-*set variable* (F=175.42, p<0.0026), which is a crucial component of the program.

The majority of errors in this cluster can be characterized as small logic errors, shown in Figure 5 (Left). Several students correctly set an existing variable to the sum of the previous two inputs, but they would often overwrite a variable that stores one of the inputs before using it, thus making the sum incorrect. Several other programs in this cluster solve most of the activity but leave out one key component, such as the last *print* block. Another set of students seem to have a misunderstanding of how variables operate and would constantly overwrite variables or set a variable equal to itself (e.g., x = x). While there were only five students who exhibited this trait, they often used a large number of blocks to attempt to overcome the incorrect code.

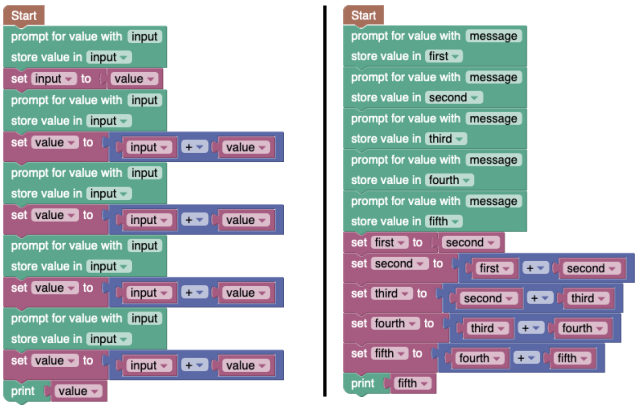

Figure 5: (Left) An example of a logic error in the first *set variable* block; (Right) An example of not following the instructions of only using two variables.

A slightly different category of error in this cluster of student programs was not following instructions carefully as seen in Figure 5 (Right). For example, this activity requires students to use only two variables. However, at least eight students used additional variables in their program. A different version of this phenomenon was exhibited by three student programs that did not prompt the user for all five inputs but would otherwise complete the activity.

The family of errors present in student programs in this cluster could likely be handled better within the system, by providing hints related to the specific error states identified above. Predictive models could be trained on specific error states to identify which type of error a student's program exhibits, which could then generate a specific type of hint or feedback. Having students test smaller components of their code would help fix the overall functionality of the entire program. Another way to address issues with programs in this cluster would be to remind students to carefully follow instructions (e.g., "Remember to only use two variables"), or provide feedback after the system has detected excessive variable usage.

4.1.3 Disorganized Cluster

The final cluster within this activity consisted of only 9 student programs. Programs in this cluster generally consisted of many more cluttered workspaces than the other clusters, but the programs were generally *Disorganized*. While these programs may resemble those in the *Near Misses* cluster, they consist primarily of unfinished ideas and more exploration-based coding. This group exhibited more effort than the *Exploration* group, but still had many errors. Specifically, programs in the *Disorganized* group seemed to attempt to overcome early strategic error by adding more blocks, making their programs comparably large.

Students in this cluster tended to approach the problem from a slightly different perspective. Instead of using the *set variable* block, they would often choose the *change variable* by block instead. This resulted in a less intuitive and more complex program. Students in this cluster often struggled with the logic of setting variables. Additionally, several students in this group would use too many blocks, often disconnected from the *Start* block and presumably left as unexplored or discarded, but interesting, ideas to the students. With the surplus of blocks, the programs seem less organized, and the workspaces would appear cluttered. One student used too many variables in their solution, while another student attempted to complete this problem with only one variable.

To address the student programs within this cluster, it would be useful to indicate to students that they can delete blocks and test smaller portions at a time. A possible remedy would be to break the problem up into smaller segments for the students to construct from the ground up.

4.2 Repeat Loop Programming Activity

Through the clustering algorithm, we again derived a total of three clusters for this activity. We will now describe the three clusters.

4.2.1 Exploration Cluster

The first cluster of student programs for this group was labeled *Exploration* due to its similarity with the cluster from the previous activity. There were only eight students in this cluster, with programs consisting mostly of incomplete ideas and low effort, *while* block functionality was explored broadly through small, experimental block configurations. When observing key differences in feature patterns for this cluster, we noticed that programs in this cluster have a lower frequency of the *set variable* block (F=48.59, p<0.0026). Additionally, programs in this cluster were clustered as such by having fewer usages of the sequence *prompt*-to-*set variable*, especially when inserted within a loop (F=33.78, p<0.0026).

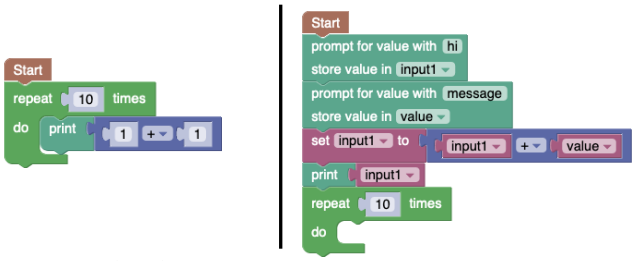

Figure 6: (Left) Student testing the functionality of the *repeat* block; (Right) Student not inserting blocks into the *repeat* block.

A key problem in four out of the eight programs in this cluster was that students did not insert any blocks within the *repeat* block as seen in Figure 6 (Right). Thus, they failed to complete a critical portion of the program. One student did actually insert blocks into the *repeat* block, but did not include other key functionality, such as printing the result of the program. Some students seemed to be purely testing what the *repeat* block performed, as was the case with one other student as seen in Figure 6 (Left).

In addressing the issues with this cluster of student programs, it is critical to encourage students to continue persisting in their programming, even after exploring block functionality. Additionally, providing worked examples of properly implemented loops may help scaffold that difficult portion of the exercise.

4.2.2 Near Miss Cluster

Of the students who did not complete this programming activity, the *Near Misses* cluster was the most common category for these programs. Eleven student programs fell into this cluster. These students generally followed instructions, clearly showed signs of significant effort, and had one or two minor mistakes in their program.

The mistakes in these programs primarily consisted of slightly incorrect logic. For instance, several students overwrote a variable exactly one time in their loop, which caused the variable to not update correctly during each iteration. One student added the two variables together one too many times, thus resulting in a sum that was too large (Figure 7). This error is common in programming, analogous to "off by one" errors in looping. This student essentially performed the loop for one additional unnecessary iteration. Another student included all of the logical steps that would go inside of their loop block but did not actually use the loop block.

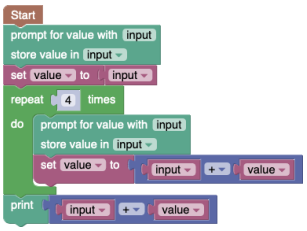

Figure 7: Program that added the variables one too many times.

Similar to the *Near Misses* cluster in the Accumulator Programming Activity, the students in this cluster could benefit from hints based on addressing slight logical errors. Feedback triggered on these common error states, such as checking to see if a variable is getting overwritten or if the loop is iterating the incorrect number of times, could be critical to producing a correct program.

4.2.3 Disorganized Cluster

This cluster of student programs also consisted of eight student programs. These programs were exclusively very large programs that either did not follow instructions or did not adjust code from the previous problem. Programs in this cluster seemed to be clustered as such by having a higher frequency of the sequence *prompt*-to-*set variable* (F=76.35, p<0.0026). Additionally, this cluster of programs was distinguished by its higher frequency of blocks in general (F=27.72, p<0.0026). Programs in this cluster were also distinguished as having higher counts of blocks that were not attached to the *Start* block, on average (F=16.61, p<0.0026).

Some aspects of this cluster, such as the large number of blocks used and a large number of blocks not attached to the *Start* block, mirror the *Disorganized Cluster* of the Accumulator activity. Six of the eight students used too many *prompt* or *set variable* blocks. These students also borrowed the same structure as the previous problem in their solution, which is only a small component specified by the instructions. These students generally had portions of correct logic in their programs but did not appear to follow the provided instructions, specifically in regard to utilizing the *repeat* block. Several students also had disconnected blocks in the program, making the workspace appear cluttered. Several other students seemed to have an overall acceptable program structure, but had errors in their looping logic (e.g., the *prompt* block was outside of the loop).

In addressing these student errors, reminding students to change the structure from their previous program could help encourage the correct solution. The logical flow of the two

activities is the same, but there is a fundamental difference in structure when using a *repeat* block.

5 Discussion and Limitations

In clustering student programs, common patterns emerged across both activities. Within the Accumulator and Repeat Loop activities, we were able to distinguish *Exploration*, *Disorganized*, and *Near Miss* programs. While the characteristics of the clusters for each activity were unique to the content of that activity, the overall patterns were similar. *Exploration* programs tended to be small, modular programs that either seemed to be testing an individual component or functionality, or they were a low-effort attempt at assembling any sort of blocks. Feedback designed to continue to motivate the student, or designed to encourage low self-confidence students, could be used to address these programs. Additionally, if the system flagged the student for this category of programs, the instructor could provide additional support. *Disorganized* programs tended to be large, cluttered programs that frequently had blocks that were not attached to the *Start* block. Students in this cluster seemed to not follow instructions. Feedback designed to focus on the details of the program requirements, then break the problem down into smaller focused components could help students overcome challenges exhibited by these programs. Students with *Near Miss* programs were more intentional with the blocks they used, and often only lacked a small component to achieve a correct submission. These programs frequently suffered from minor logic errors, which can be addressed through feedback. Identifying the specific logic error state would allow for targeted feedback to address these programs.

While this work is specific to two activities within the PRIME environment, results from this endeavor reveal that by using system-logged features, a clustering algorithm can identify students who are facing common difficulties. These clusters can be visually interpreted, which allows educators and system designers to better understand the types of feedback students need that is informed by the context. Since introductory CS courses often have a high student-to-instructor ratio, clustering provides a strategic, scalable tool that can be leveraged by both machine and human to formatively assess and support students.

Overall, this work further highlights the need for instructors to engage in post-hoc investigation of student errors at a more granular level to identify what types of interventions can be deployed to best address these struggles in future iterations of the course. For example, while interventions such as presenting well-chosen example programs or code-snippets may be effective support for *Near Misses*, additional instruction focusing on testing and debugging strategies may be more beneficial for students creating *Disorganized* programs. While these specific clusters may not exist for all exercises, combining data-driven approaches with qualitative analysis can potentially move the community towards shared taxonomies of misconceptions and a better understanding of the effectiveness of different types of interventions.

While this work was able to identify patterns of incomplete or incorrect student programs, further validation is needed to make a definitive connection between these programs and specific student misconceptions demonstrated across multiple activities or identified through survey instruments. Another limitation of this work is that the clusters analyzed are specific to these PRIME activities. However, the clustering techniques described above can readily be applied to other block-based programming artifacts and potentially text-based programming artifacts when n-gram/segments of code can be defined and automatically detected.

6 Conclusion

With the goal of supporting student learning in introductory undergraduate CS courses, we explored the use of clustering with programs of students who did not complete two activities in the PRIME block-based programming environment. Using system-logged features and patterns of key block sequences (i.e., n-grams), we identified three distinct clusters of these programs using Bayesian Gaussian Mixture Models. The three distinct clusters of *Exploration*, *Disorganized*, and *Near Miss* programs indicated that there may be more general identifiable patterns across block-based programs that can be automatically detected. Identifying these in real-time could lead to a better support system for novice programmers who may otherwise disengage from the assignments.

In future work, it will be important to explore how to most effectively use these clusters as a mechanism to provide adaptive feedback for students. Additionally, it will be important to further define key features for the analysis of student code. For example, including features that indicate behavioral components of the code, as well as test-case type analysis, will support more effective code categorization. By creating and testing different feature sets across different activities and block-based programming environments, the community can move toward a more detailed taxonomy of student errors and misconceptions, while at the same time enabling researchers to investigate which types and combinations of formative feedback, classroom instruction, and learning activities can most effectively address the different issues. This will also enable a more detailed comparison between block-based and text-based programming environments, as well as provide further insight into what learners can benefit most from each modality and when in the learning process each modality fits best.

It will also be informative to expand the analysis to examine the trajectories of student programs. In addition to using the final snapshot of code, it would also be useful to use the sequences of student code and individual actions taken by the student as they create their program, especially in larger, more open-ended tasks. Beyond improving this particular learning environment, the results call for a systematic study of student behavior in block-based programming environments, as well as the investigation of how best to support learners by providing guidance to address misconceptions early in the learning process.

ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under Grants DUE-1626235 and DUE-1625908. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Akram, B., Min, W., Wiebe, E.N., Mott, B.W., Boyer, K.E. and Lester, J.C. 2018. Improving stealth assessment in game-based learning with LSTM-based analytics. In *Proceedings of the 11th International Conference on Educational Data Mining*, 208–218.

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

[3] Amershi, S. and Conati, C.C. 2009. Combining Unsupervised and Supervised Classification to Build User Models for Exploratory. *JEDM-Journal of Educational Data Mining*. 1, 1, 18–71.

[4] Blei, D.M. and Jordan, M.I. 2006. Variational inference for Dirichlet process mixtures. *Bayesian Analysis*. 1, 1 A, 121–144.

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

[6] Glassman, E.L., Scott, J., Singh, R., Guo, P. and Miller, R.C. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction*. 22, 2, 7:1-35.

[7] Google blockly-a visual programming editor: 2013.

[8] Joyner, D.A., Salguero, E., Arrison, R., Wang, Z., Yin, K., Ruksana, M. and Wellington, B. 2019. From clusters to content: Using code clustering for course improvement. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 780–786.

[9] Kardan, S., Roll, I. and Conati, C. 2014. The usefulness of log based clustering in a complex simulation environment. In *International Conference on Intelligent Tutoring Systems*, 168–177.

[10] Marin, V.J., Pereira, T., Sridharan, S. and Rivero, C.R. 2017. Automated personalized feedback in introductory Java programming MOOCs. In *Proceedings of the International Conference on Data Engineering*, 1259–1270.

[11] Min, W., Frankosky, M.H., Mott, B.W., Rowe, J.P., Wiebe, E., Boyer, K.E. and Lester, J.C. 2015. DeepStealth: Leveraging deep learning models for stealth assessment in game-based learning environments. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence in Education*, 277–286.

[12] Parihar, S., Das, R., Dadachanji, Z., Karkare, A., Singh, P.K. and Bhattacharya, A. 2017. Automatic grading and feedback using program repair for introductory programming courses. In *Annual Conference on Innovation and Technology in Computer Science Education*, 92–97.

[13] Price, T.W. and Barnes, T. 2015. Comparing textual and block interfaces in a novice programming environment. In *Proceedings of the 11th International Conference on International Computing Education Research*, 91–99.

[14] Price, T.W., Dong, Y. and Barnes, T. 2016. Generating data-driven hints for open-ended programming. In *Proceedings of the Ninth International Conference on Educational Data Mining*, 191–198.

[15] Price, T.W., Dong, Y. and Lipovac, D. 2017. iSnap: Towards intelligent tutoring in novice programming environments. In *Proceedings of the Forty-Eighth ACM Symposium on Computer Science Education*, 483–488.

[16] Qian, Y. and Lehman, J. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education*. 18, 1.

[17] Singh, R., Gulwani, S. and Solar-Lezama, A. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 15–26.

[18] Sorva, J. 2013. Notional machines and introductory programming education. *ACM Transactions on Computing Education*. 13, 2.

[19] Striewe, M. and Goedicke, M. 2014. A Review of static analysis approaches for programming exercises. In *International Computer Assisted Assessment Conference*, 100–113.

[20] Swidan, A., Hermans, F. and Smit, M. 2018. Programming misconceptions for school students. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. August, 151–159.

[21] Ullah, Z., Lajis, A., Jamjoom, M., Altalhi, A., Al-Ghamdi, A. and Saleem, F. 2018. The effect of automatic assessment on novice programming: Strengths and limitations of existing systems. *Computer Applications in Engineering Education*.

[22] Wang, L., Sy, A., Liu, L. and Piech, C. 2017. Learning to represent student knowledge on programming exercises using deep learning. In *Proceedings of the 10th International Conference on Educational Data Mining*, 324–329.

[23] Watson, C. and Li, F.W. 2014. Failure rates in introductory programming revisited. In *Proceedings of the 19th Conference on Innovation & Technology in Computer Science*, 39–44.

[24] Weintrop, D. and Wilensky, U. 2017. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education*. 18, 1, 1–25.

[25] Xie, B. and Abelson, H. 2016. Skill progression in MIT app inventor. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*, 213–217.