Rodrigo, M. M. T. et al. (Eds.) (2021). Proceedings of the 29th International Conference on Computers in Education. Asia-Pacific Society for Computers in Education

Analysing Reachable and Unreachable Codes in App Inventor Programs for Supporting the Assessment of Computational Thinking Concepts

Siu Cheung KONG[a*], Chun Wing POON[b] & Bowen LIU[b]
[a]*Department of Mathematics and Information Technology, The Education University of Hong Kong, Hong Kong*
[b]*Centre for Learning, Teaching and Technology, The Education University of Hong Kong, Hong Kong*
*sckong@eduhk.hk

Abstract: Examining students' programming artefacts is a practical and effective approach to assessing the development of students' computational thinking (CT). Some tools can automatically assess students' programming artefacts according to predefined rubrics. However, these tools may fail to properly assess students' artefacts when unreachable codes exist. In this study, we designed an automatic assessment tool that can analyse reachable and unreachable codes independently in students' App Inventor programs in static code analysis. Our tool counts reachable and unreachable codes separately to avoid potential bias. The designed tool can process students' programs in a batch and calculate the average block frequency of all programs in the batch. The average block frequency provides teachers with an initial impression of the achievement of the entire class. Previous assessment tools have used predefined rubrics to assess students' performance. We suggest that students' understanding of CT concepts could be assessed by block frequency comparison with a known programming solution. By comparing the frequency of the reachable and unreachable code in the programs with that of the starter code, students' progression in the task can be evaluated. By comparing the block frequency of these programs with a suggested solution, students' difficulties in programming can be identified.

Keywords: App Inventor, assessment, computational thinking, concepts, reachable codes, unreachable codes

1. Introduction

Assessment is an important issue in computational thinking (CT) education for K–12 students. Effective and efficient CT assessments can measure students' learning and highlight gaps in their understanding. In this study, we used an enhanced static code analyser with unreachable code detection to assess students' understanding of the CT concepts in well-structured, open-ended programming tasks. We developed a web-based tool to analyse students' App Inventor programs. This tool generates a block frequency report in which reachable and unreachable code blocks are counted separately. Different from previous assessment tools that have relied on predefined rubrics for assessment, it is possible to assess students' CT concepts by directly comparing block frequency in a well-structured, open-ended programming task, which usually provides a starter code and a suggested solution. In this study, we provide an example of the application of our assessment tool to a group of students' App Inventor programs to examine students' understanding of the list data structure. With the batch processing function of this tool, teachers can easily obtain an impression of the learning status of the entire class and identify students' learning gaps in the acquisition of certain CT concepts.

2. Research Background

Computational thinking, which is a set of analytical thinking skills that uses fundamental computing concepts and practices for problem-solving, has been regarded as a necessary ability in the twenty-first century (Wing, 2006). Therefore, much effort has been devoted to promoting CT education in K–12 education (Grover & Pea, 2013; Lye & Koh, 2014). Because the goal of CT goes beyond teaching students to be digital consumers and aims to encourage problem solving with computing methods, CT education usually involves activities that provide students opportunities to create computational artefacts. Teaching CT through a programming activity is a popular and practical strategy. Students can be exposed to CT concepts during the process of creating their programmes within block-based visual programming environments. Students' programming artefacts are natural and informative products for assessing CT development. Several automatic tools, such as Dr. Scratch (Moreno-León & Robles, 2015) and CodeMasters artefacts (Von Wangenheim et al., 2018), have been developed to assess students' artefacts. These tools conduct static code analyses of students' programming artefacts and score the programs based on predefined rubrics. The rubrics set criteria to map scores to students' CT-related abilities using features and traits of code. Although these tools can significantly alleviate teachers' workload during assessment, they are inadequate when unreachable codes exist in students' programs.

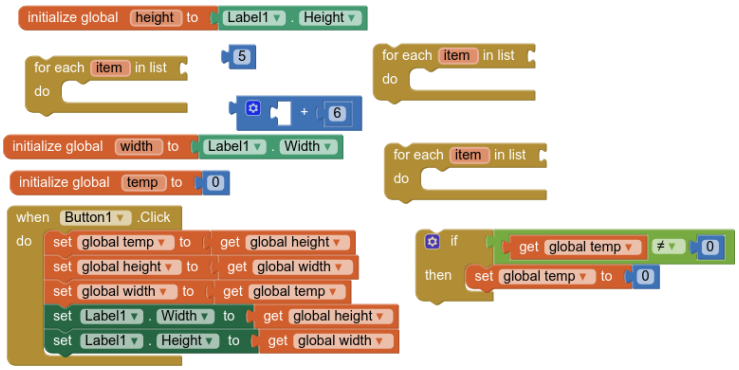

Figure 1. A Students' App Inventor Program that Contains Unreachable Codes.

Unreachable codes, also called dead codes, are codes that are never executed in the complete run-time of a program (Amanullah & Bell, 2020). Although unreachable codes seem to have no harmful effect on programs' functionality, they increase the complexity of the programs, reduce the readability of codes, and introduce potential risks in the iterative development process of programs. In using traditional static-code analysis for CT assessment, unreachable codes can distort the assessment results and provide inaccurate feedback to teachers and students. Figure 1 depicts a student's App Inventor program that attempts to switch the width and height of a label. Notably, several loop blocks are dangling without connecting to an event handler. However, this program may mislead the assessment tool to conclude that the student has properly understood the loop concept. Although having an unreachable code seems to be a bad programming habit, some novice programmers can benefit from it (Amanullah & Bell, 2018). The unreachable codes sometimes work as a temporary workspace to store unused blocks and provide a start point for novices. The proportion of unreachable codes in students' final programming products can also indicate their achievement in learning tasks. In some well-structured, open-ended programming tasks, starter codes are provided at the beginning and are not fully executable. The unreachable codes that remain in the final products may indicate students' barriers in the programming process.

3. A Web Tool for Automatic Analysis of App Inventor Programs

To identify unreachable codes, we developed a tool for the automatic detection of unreachable codes in students' App Inventor programs. The tool is available on the web for teachers to use, and it supports batch processing of students' App Inventor programs.

3.1 Unreachable Codes in App Inventor Programs

App Inventor is a block-based programming environment for creating mobile applications. In most situations, codes are designed in event-driven patterns in an app, and the reachable codes are components of either an event handler, a procedure, or a global variable definition. If a block of codes has a top-level block type that is not one of the above three types, it is regarded as an unreachable code block because there are no ways to activate its execution. Figure 2a shows an example of a dangling code block that is unreachable in an App Inventor program. Additionally, empty event handlers, procedures without connected blocks, and incomplete global variable definitions (Figure 2b) were also regarded as unreachable codes in this study.

There are other scenarios that make the codes unreachable, such as an event that can never be triggered due to a missing event-driven item in the design interface in App Inventor. We did not define these codes as unreachable in this study because they can be reached and rectified after a logical structured walk-through of the program design (Lemos, 1979).

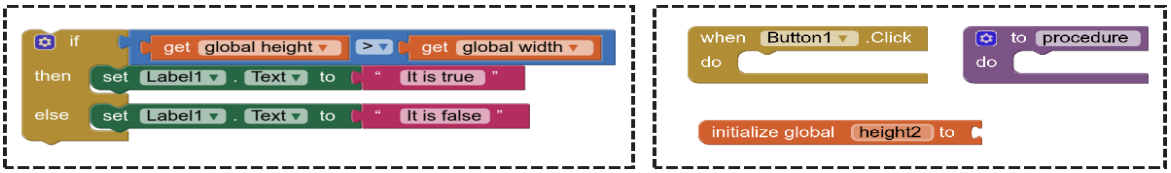

Figure 2. Examples of Unreachable Codes with 2(a) shows a dangling code block that is unreachable and 2(b) with empty event handlers, procedures without connected blocks and incomplete global variable definition

3.2 Unreachable Code Detection via XML Parsing

Our tool is an App Inventor analyser developed in Java and deployed as a web service. The web tool can analyse App Inventor programs in a batch and summarize the block frequency of all the programs. Our tool first extract the XML tree from the .aia file, then identify the blocks in the program via a lexical analysis. After identifying the blocks, our tool counts the number of blocks by traversing the XML tree and generates a block frequency report. Table 2 shows the results of analysis of a students' App Inventor program in Figure 1. The categories used in the table classify blocks according to their related CT concepts. The following two steps are used to identify whether a code block is reachable or not: 1) check whether the top-level block is either an event handler, a procedure, or a global variable definition; 2) check whether the top-level block is connected to any other blocks. Unreachable codes are not repeatedly counted in other categories.

Table 2. *The Block Frequencies of the Codes in Figure 1*

	Block Frequency
CT concepts:	
Events & Sequences	2
Conditionals	0
Operators	0
Repetition	0
Naming & Variables	14
Data Structures	0
Procedure	0
Unreachable Codes:	
Empty Event Handler	0
Empty Procedure	0
Incomplete Variable Initialization	0
Dangling Blocks	12

4. Example Analysis of a Batch of App Inventor Programs

To show the effectiveness of the assessment tool, we analysed a batch of students' App Inventor programs to illustrate how the detection of unreachable codes could indicate students' understanding of CT concepts and how teachers could assess the programs.

4.1 Data Description

Our testing data contained 27 App Inventor programs developed by primary school students. These programs were the students' assignments in an App Inventor course. In this course, the students learned how to use the list, which is an abstract data type used in programming tasks. The assignment required the students to develop a mobile app that could randomly print the name of a country from a given list. This task required students to have basic knowledge of the list data structure, including how to create a list and how to access the elements using the index of the list. The task was presented to students together with the starter code shown in Figure 3a. The starter code contained unreachable codes and acted as a start point for students to complete the programming task. A suggested solution is shown in Figure 3b.

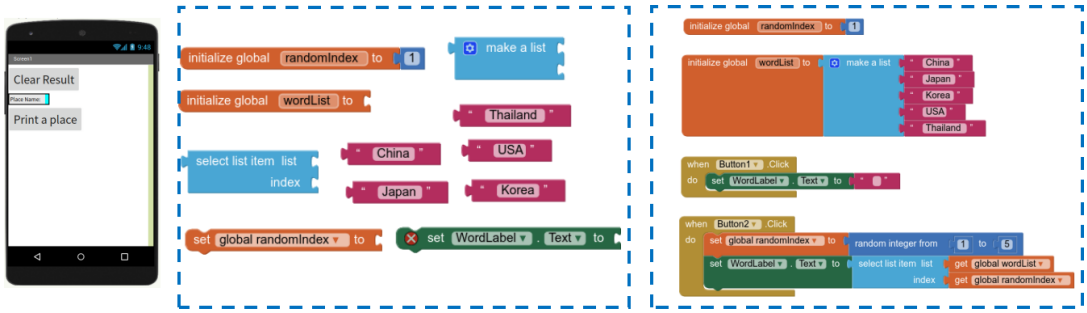

Figure 3. A Programming Task using List Data Structure. 3a: the starter code of the task; 3b: a suggested solution of the programming task.

4.2 Analysis of the Students' Programs

The batch processing function of our tool allowed us to analyse all of the programs at the same time. We uploaded all 27 programs to the analyser, and part of the generated result is shown in Table 3. The generated result contained the block frequency of the 27 programs and an average block frequency of all of the programs. The block frequencies of only five programs are presented in Table 3 for further discussion. We appended the block frequency of the starter code and the suggested solution for comparison.

From the block frequency of the starter code (Table 3), we can easily notice that most of the codes are counted into the categories under unreachable codes. A decrease in unreachable codes could be used to indicate the students' progress in this task. Eleven programs (40.7%) were free of unreachable codes. The average frequencies in the unreachable code categories were significantly lower than those measured initially, which suggests that the students correctly understood the goal of the task and demonstrated a certain degree of understanding of the list data structure. By comparing the average block frequency and suggested solution, we noticed that the average frequency of blocks in the '*Events and Sequences*', '*Naming & Sequences*', and '*Data Structures*' were lower than that in the suggested solution. We calculated the standard deviation (SD) and coefficient of variation (CV) in each category and the categories of the unreachable codes (Table 4). The '*Data Structures*' category had the largest variation among the three CT concept categories, suggesting that students had a different understanding of the data structure, which was the list structure in our study. The SD and CV of the categories under unreachable codes were relatively higher than those of the CT concepts categories. This suggests that unreachable codes can be a useful indicator of students' achievement in this task and more information can be explored in these unreachable codes.

We investigated the functionality of each program by comparing it with the block frequency of the suggested solution. If a program's block frequency was identical to that of the suggested solution, the student's program likely fulfilled the required function, although the connection between the blocks would still require a manual inspection. Such an inference would be reasonable if an open-ended question were well structured. From our data, eight programs (29.6%) were found to be identical to our suggested solution. The difference in block frequency can indicate the barrier in students' problem-solving process. We compared the block frequency of program 4 to our suggested solution and found that it had zero entries in the '*Data Structure*' category and non-zero entries in the '*Incomplete Variable Initialization*' and '*Dangling Blocks*' categories. Through a further study of the source code (Figure 4), we noticed that the student included the list-related operation blocks into the app but failed to connect those blocks correctly. This means that the student was not fully familiar with the usage of the list.

Table 3. *The Analysed Results Obtained using Our Developed Tool*

	Average Block Frequency	Starter code	App Inventor Program 1	App Inventor Program 2	App Inventor Program 3	App Inventor Program 4	App Inventor Program 5	Suggested Solution
CT Concepts:								
Events & Sequences	2.44	0	3	3	3	2	2	3
Conditionals	0	0	0	0	0	0	0	0
Operators	0	0	0	0	0	0	0	0
Repetition	0	0	0	0	0	0	0	0
Naming & Variables	11.85	2	15	14	14	5	13	14
Data Structures	1.74	0	3	2	2	0	0	2
Procedure	0	0	0	0	0	0	0	0
Unreachable Codes:								
Empty Event Handler	0.14	0	0	1	0	0	3	0
Empty Procedure	0	0	0	0	0	0	0	0
Incomplete Variable Initialization	0.22	1	2	0	0	1	0	0
Dangling Blocks	2.6	9	6	1	0	12	3	0

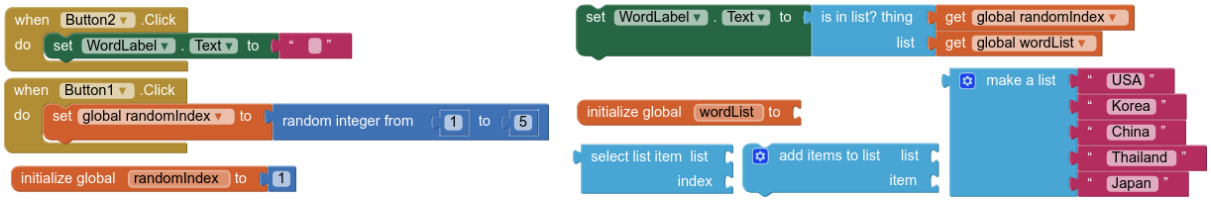

Figure 4. The Block Code of the App Inventor Program 4 in Table 3.

Table 4. *The Standard Deviations and Coefficients of Variation of the Block Frequencies*

	Average Block Frequency	Standard Deviation	Coefficient of Variation

CT Concepts:			
Events & Sequences	2.44	0.974	0.398
Conditionals	0	0	-
Operators	0	0	-
Repetition	0	0	-
Naming & Variables	11.85	3.697	0.311
Data Structures	1.74	0.712	0.409
Procedure	0	0	-
Unreachable codes:			
Empty Event Handler	0.14	0.60	4.06
Empty Procedure	0	0	-
Incomplete Variable Initialization	0.22	0.50	2.27
Dangling Blocks	2.67	3.63	1.36

5. Summary

In this study, we assessed students' CT concepts by analysing reachable and unreachable code separately in a static code analysis. We defined several patterns of unreachable codes and developed an automatic analyser that could detect these unreachable codes, count the block frequency, and process students' App Inventor programs in a batch. We tested our tool on a group of students' App Inventor programs developed in a well-structured, open-ended task that required knowledge of the list data structure. By comparing the block frequency of the starter code and the average block frequency of students' programs, teachers can quickly gain a high-level impression of the learning status of an entire class. The differences in unreachable code frequencies between the starter code and students' programs indicate the students' achievement. By comparing the individual block frequency with that of the suggested solution, teachers can rapidly locate gaps in students' programming processes. The assessment tool and method used in this study require the programming tasks to be open-ended and well-structured with at least one correct solution known in advance. Our method assesses students' programs by comparing block frequencies instead of using predefined rubrics. It enables efficient assessment and customised analysis. Currently the comparison of block frequencies is performed using numeric observations. In future studies, we will consider visualising the comparison of block frequencies to enable teachers to obtain visual hints for the assessment. We will conduct experiments in primary schools to further verify the efficacy and reliability of the system. In addition to assessment, we will explore the potential of applying the system to provide self-serve real-time feedback in students' programming learning.

References

Amanullah, K., & Bell, T. (2018). Analysing students' scratch programs and addressing issues using elementary patterns. *2018 IEEE Frontiers in Education Conference (FIE)*, 1–5.

Amanullah, K., & Bell, T. (2020). Revisiting code smells in block-based languages. *Proceedings of the 15th Workshop on Primary and Secondary Computing Education*, 1–2.

Grover, S., & Pea, R. (2013). Computational thinking in K-12: A review of the state of the field. *Educational Researcher*, *42*(1), 38–43.

Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior*, *41*, 51–61.

Lemos, R. S. (1979). An implementation of structured walk-throughs in teaching COBOL programming. *Communications of the ACM*, 22(6), 335-340.

Moreno-León, J., & Robles, G. (2015). Dr. Scratch: A web tool to automatically evaluate Scratch projects. *Proceedings of the 10th Workshop in Primary and Secondary Computing Education*, 132–133.

Von Wangenheim, C. G., Hauck, J. C. R., Demetrio, M. F., Pelle, R., da Cruz Alves, N., Barbosa, H., & Azevedo, L. F. (2018). CodeMaster - Automatic assessment and grading of app inventor and snap! Programs. *Informatics in Education*, *17*(1), 117–150. https://doi.org/10.15388/infedu.2018.08

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33–35.