An Item Response Theory Analysis of Algorithms and Programming Concepts in App Inventor Projects

Nathalia da Cruz Alves
Department of Informatics and Statistics
Federal University of Santa Catarina
Florianópolis/Santa Catarina/Brazil
nathalia.alves@posgrad.ufsc.br

Christiane Gresse von Wangenheim
Department of Informatics and Statistics
Federal University of Santa Catarina
Florianópolis/Santa Catarina/Brazil
c.wangenheim@ufsc.br

Jean Carlo Rossa Hauck
Department of Informatics and Statistics
Federal University of Santa Catarina
Florianópolis/Santa Catarina/Brazil
jean.hauck@ufsc.br

Adriano Ferreti Borgatto
Department of Informatics and Statistics
Federal University of Santa Catarina
Florianópolis/Santa Catarina/Brazil
adriano.borgatto@ufsc.br

ABSTRACT

Computing education is often introduced in K-12 focusing on algorithms and programming concepts using block-based programming environments, such as App Inventor. Yet, learning programming is a complex process and novices struggle with several difficulties. Thus, to be effective, instructional units need to be designed regarding not only the content but also its sequencing taking into consideration difficulties related to the concepts and the idiosyncrasies of programming environments. Such systematic sequencing can be based on large-scale project analyses by regarding the volition, incentive, and opportunity of students to apply the relevant program constructs as latent psychometric constructs using Item Response Theory to obtain quantitative 'difficulty' estimates for each concept. Therefore, this article presents the results of a large-scale data-driven analysis of the demonstrated use in practice of algorithms and programming concepts in App Inventor. Based on a dataset of more than 88,000 App Inventor projects assessed automatically with the CodeMaster rubric, we perform an analysis using Item Response Theory. The results demonstrate that the easiness of some concepts can be explained by their inherent characteristics, but also due to the characteristics of App Inventor as a programming environment. These results can help teachers, instructional and curriculum designers in the sequencing, scaffolding, and assessment design of programming education in K-12.

KEYWORDS

Algorithms and Programming, App Inventor, Item Response Theory, Sequencing.

EduComp'21, Abril 27–30, 2021, Jataí, Goiás, Brasil (On-line)

1 INTRODUCTION

The importance of computing nowadays for anyone regardless of the area of expertise is widely recognized. Consequently, computing education is making its way into K-12 worldwide, ranging from online MOOCs, extracurricular activities to courses fully integrated into the curriculum [1][2][3]. Several countries have developed guidelines and curricula for K-12 computing education [4]. Among, those, one of the most prominent models is the K-12 Computer Science Framework [5] defining a set of core computing concepts and practices to be covered in K-12. The core concepts represent major content areas in the field of computer science, including computing systems, networks, data and analysis, algorithms & programming as well as the impacts of computing. Core practices represent behaviors that computationally literate students should use to engage with the concepts of computing, such as recognizing and defining computational problems and creating computational artifacts. The standard also defines the sequencing of these concepts and practices describing how the students' conceptual understanding and practice of computing should become more sophisticated over time and across educational stages in K-12. Other guidelines and curricula, such as Computing at School [6] or the Brazilian Computer Society Guidelines for Computing Education in K-12 [7], cover similar basic concepts and practices.

There are several approaches to teach computing, yet, in practice, they typically focus on algorithms and programming concepts and related practices as being one of the main knowledge areas of computing [1][8][9]. This comprises the competency to develop algorithms to solve problems in a language that computers can understand including basic programming concepts such as control (e.g., loops and conditionals), modularity, variables, etc. (Figure 1).

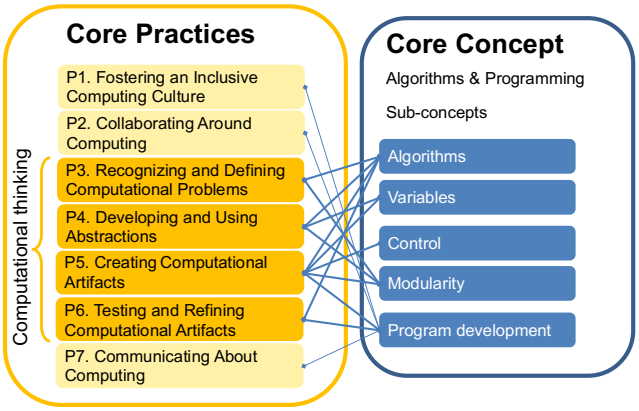

Figure 1: Core practices and sub-concepts related to the core concept algorithms & programming concepts [5]

Variables refer to storing and manipulating data from computer programs. Control concepts specify the order in which instructions are executed within an algorithm or program (e.g., using loops and/or conditionals). Modularity involves dividing complex tasks into simpler tasks and combining them to create something complex. Program development represents the software engineering process that is repeated until acceptance criteria are met. In addition, several core practices are related to algorithms & programming as presented in Figure 1.

In order to introduce programming in K-12, typically visual block-based environments are used. These environments allow to choose and drag-and-drop commands providing visual cues to the user as to how and where commands may be used reducing the cognitive load for novices [10][11]. A prominent example is App Inventor (appinventor.mit.edu), an online platform for the development of mobile applications for Android devices. It is used by a wide range of people of all ages and backgrounds with more than 1 million unique monthly active users from 195 countries who created almost 35 million mobile apps as of January 2021. App Inventor projects can be shared via the App Inventor Gallery [12] under the creative commons license. App Inventor is also widely used to teach computing through the development of mobile applications [13] adopting diverse instructional strategies, ranging from well-defined interactive tutorials to open-ended ill-structured activities in a constructivist context following a problem-based learning approach [14]. These typically aim at teaching students to create their mobile applications to solve real-world issues applying a computational action strategy to make computing education more inclusive, motivating, and empowering for young learners [15][16]. More and more also adaptive learning systems are being adopted [17] providing personalized instruction and feedback tailored to the needs of individual learners.

Yet, learning to program is a highly complex process and novices struggle with a wide range of difficulties [18][19]. It involves diverse cognitive activities and mental representations concerning the analysis of requirements, design, program understanding, modifying and debugging, as well as the construction of conceptual knowledge on basic operations (such as loops, conditional statements, etc.) [20]. Learning programming can be considered an exploratory process in which software artifacts are created through an incremental problem-solving process using multiple competencies, i.e., computational concepts, practices, and perspectives [21][22][23].

Thus, in order to be effective, instructional units aimed at teaching programming need to be systematically designed taking into consideration not only the content to be taught but also the sequencing of instruction and the idiosyncrasies of programming environments. As the order and organization of learning activities affect the way information is processed and retained [24], it is important to sequence the content in a way it can be most easily grasped by the student using a particular programming environment [25] to improve the learners' understanding and to help them to achieve the objectives [26]. If inadequately sequenced, a learner may be overloaded, which can negatively affect learning, performance, and motivation [27]. How content is sequenced is determined by the developmental level and current comprehension of the student, the instructional method, and the evolutionary structure of the knowledge on the given subject [28]. There are many different ways to sequence content elements [29], as, for example by adopting a simple to complex sequence strategy according to the main types of knowledge structure [30].

Thus, finding an optimal learning sequence is difficult, especially for different programming environments used to teach algorithms and programming concepts. Therefore, it is important to investigate the factors that lead to students learning difficulty in programming. Several studies already examine the learning of specific concepts when developing apps with App Inventor, including procedural abstraction concepts [31], events [9], programmatic sophistication [32], effectiveness [33], or appropriateness [10] of App Inventor as an educational environment. Others study the learning progression of students in computing courses in K-12, e.g., Xie and Abelson [34], who analyze the relationship between the progression of skill in using App Inventor functionality and in using computational thinking concepts as learners create more apps. Other research aiming at investigating the difficulty of content in computing education analyzing how students learn to program is mostly related to higher education [35], other block-based languages, such as Scratch [36][37][38][39][40][41], LaPlaya [42], and SNAP! based environments [43], object-oriented programming [44], etc.

The assumption in many of these studies is that student progress can be understood through difficulties with specific programming constructs. Thus, the analysis of code created can provide insights concerning the 'difficulty' of learning certain concepts. Depending on the activities (well-defined or ill-defined) the programming ability of a person can be influenced by the volition, incentive, and opportunity to apply computing concepts in a programming environment and those factors should be taken into account.

An alternative is to regard those constructs as latent psychometric constructs and use Item Response Theory (IRT) [45] to obtain quantitative 'difficulty' estimates for each content element [45]. IRT refers to a family of mathematical models that

attempt to explain the relationship between latent traits (unobservable characteristics or attributes such as volition, incentive, and opportunity to apply computing concepts, including loop, conditional concepts, etc. in a programming project) and their manifestations (i.e., observed outcomes, performance such as using loop and conditional blocks in App Inventor projects). Typically applied for testing, IRT establish a link between the properties of the items on an instrument, individuals responding to these items, and the underlying trait being measured. IRT assumes that the latent trait and items of a measure are organized in an unobservable continuum. Therefore, its main purpose focuses on establishing the individual's position on that continuum. IRT is widely used for large-scale assessments [46], such as PISA (https://www.oecd.org/pisa/) or TOEFL (https://www.ets.org/toefl).

Yet, it can also be used to obtain systematic information about the 'difficulty' of concepts and the distribution of the respective competencies among students. This can be done based on the code created by the students as an outcome of the learning process, regarding certain attributes of the code as manifestations of latent psychometric constructs according to the principles of IRT [47][48][62]. The occurrence of certain concepts like loops or conditional statements can be considered as satisfiability on certain items (e.g., "the existence of loops"). Consequently, the probability of such satisfiability depending on the item 'difficulty', the estimated person abilities, and the volition, incentive, and opportunity to apply computing concepts, can be described by certain psychometric models, e.g., the Rasch or Graded Response Model. For example, Berges and Hubwieser [47] used IRT for assessing coding abilities by analyzing the source code created as an outcome of the learning process in the context of a freshman course at university for text-based object-oriented programming. Similarly, Kramer et al. [48] used IRT for assessing students' abilities in text-based object-oriented programming in an introductory programming course. Both studies focused on the Java programming language.

Although several studies analyze some aspects of algorithms and programming using block-based programming environments, so far, no research focusing directly on the analysis of the difficulty of concepts, including those approached by the K-12 Computer Science Standard and specifically concerning the block-based programming environment App Inventor has been found. Therefore, the objective of this study is to analyze the 'demonstrated difficulty' in App Inventor projects. Adopting IRT, we analyze algorithms & programming items based on the CodeMaster rubric [49][50] by extracting them automatically from the code of App Inventor projects. The results provide information about the 'demonstrated difficulty' of the concepts application and their distribution among the App Inventor projects. These results of this study can be used by instructional and curriculum designers in order to guide the sequencing of programming education in K-12.

2 BACKGROUND

2.1 App Inventor

One of the most prominent block-based programming environments for computing education is App Inventor that allows creating mobile applications [12]. It was originally provided by Google and it is currently run by the Massachusetts Institute of Technology. The current version 2.0 of App Inventor runs on a web browser (Figure 2), replacing App Inventor Classic. App Inventor is used by a wide audience, from K-12 to higher education, including end-user developers who write programs to support their primary job or hobbies [51][13].

A mobile app can be created in two stages with App Inventor. First, using the Designer Editor, user interface components, such as buttons, labels, etc. are configured (Figure 2). The designer also allows to specify non-visual components such as sensors, social, and media components that access mobile device features. The app's behavior is programmed in a second stage by connecting visual programming blocks in the Blocks Editor. Each block corresponds to abstract syntax tree nodes in traditional programming languages.

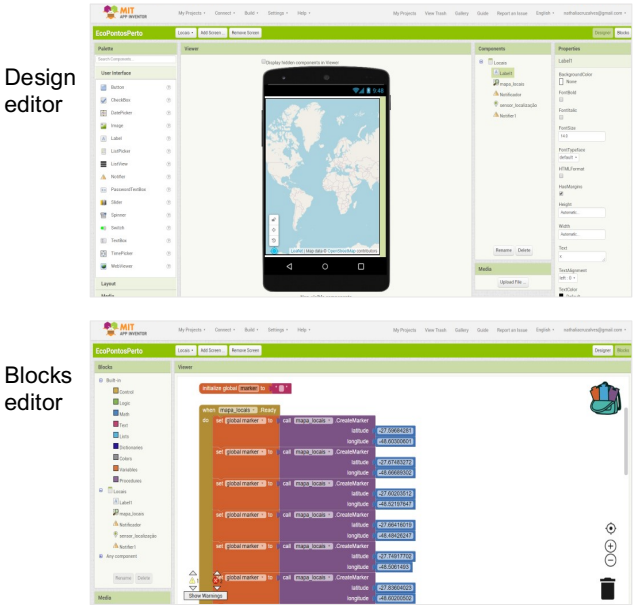

Figure 2: App Inventor Designer and Blocks Editor

Blocks can represent standard programming concepts like loops, procedures, conditionals, etc., or conditions, events, and actions for a particular component of the app or any component. App Inventor blocks are divided into two categories: built-in blocks and component blocks. Built-in blocks are available for use in any app and refer to overall programming concepts. Component blocks include events, set and get, call methods, and component object blocks that are available for specific design components added to the app (Table 1).

Table 1: Overview of App Inventor blocks

Type	Category	Description
Built-in blocks	**Control**	Blocks responsible for control commands including important blocks like loops, conditionals, and screen actions. Examples: controls_while, controls_if, controls_closeScreen.
	Logic	Blocks responsible for logic operations on variables including relational and Boolean. Examples: logic_compare, logic_operation.
	Math	Blocks responsible for creating numbers and perform basic and advanced math operations. Examples: math_add, math_cos.
	Text	Blocks responsible for creating and manipulating *original* strings. Examples: text, text_length.
	Lists	Blocks responsible for creating and manipulating *original* lists. Example: lists_create_with, lists_add_items.
	Colors	Blocks responsible for creating and manipulating colors. Examples: color_red, color_blue.
	Variables	Blocks responsible for creating and manipulating *original* variables. Examples: global_declaration, lexical_variable_set.
	Procedures	Blocks responsible for definition and call of *original* procedures. Examples: procedures_defnoreturn, procedures_callnoreturn.
Component blocks	**Events**	Blocks responsible for specifying how a component responds to certain events, such as a button has been pressed. Example: component_event
	Set and Get	Blocks responsible for change components' properties. Example: component_set_get
	Call Methods	Blocks responsible for call component methods to perform complex tasks. Example: component_method
	Component object	Blocks responsible for getting the instance component. Example: component_component_block

The source code files of the App Inventor project can be exported as AIA files. An AIA file is a compressed file collection that includes a project properties file, media files that the app uses, and two files are generated for each screen in the app: a BKY file and a SCM file. The BKY file wraps an XML structure including all the blocks of programming used to define the behavior of the app, and the SCM file wraps a JSON structure that contains all the used visual components in the app [52]. This AIA file can be automatically assessed with the algorithms & programming rubric (Table 2) by the CodeMaster tool.

2.2 CodeMaster rubric

CodeMaster [49][50] is an automated performance-based assessment rubric and grader. It enables an analysis of the code of App Inventor programs supported by a free web-based tool providing feedback to students and teachers in the form of a score with respect to algorithms & programming and the graphical user interface design of the apps created. The model has been developed based on a systematic mapping study [53] following an instructional design process [54] and the procedure for rubric definition proposed by Goodrich [55]. The rubric is based on the K-12 Computer Science Framework [5] as well as other rubrics and frameworks, including [21][8][56].

Table 2: CodeMaster rubric for assessing algorithms and programming based on the analysis of App Inventor projects

Criterion	Performance Level (categories)			
	0	**1**	**2**	**3**
1. Operators	No operator blocks are used.	Arithmetic operator blocks are used.	Relational operator blocks are used.	Boolean operator blocks are used.
2. Variables	No use of variables.	Modification or use of predefined variables.	Creation and operation with variables.	-
3. Strings	No use of strings.	Use of string block to change the text of design components.	Creation and operation with strings.	-
4. Naming	Few or no names are changed from their defaults.	10 to 25% of the names are changed from their defaults.	26 to 75% of the names are changed from their defaults.	More than 75% of the names are changed from their defaults.
5. Lists	No lists are used.	One single-dimensional list is used.	More than one single-dimensional list is used.	Lists of tuples are used.
6. Data persistence	Data are stored only in variables or UI component properties, and do not persist when app is closed.	Data is stored in files.	Local database is used.	Web database is used.
7. Events	No type of event handlers is used.	One type of event handlers is used.	Two or three types of event handlers are used.	More than three types of event handlers are used.
8. Loops	No use of loops.	Simple loops are used.	'For each' loops with simple variables are used.	'For each' loops with list items are used.
9. Conditional	No use of conditionals.	Uses 'if' structure.	Uses one 'if then else' structure.	Uses more than one 'if then else' structure.
10. Synchronization	No use of timer for synchronization.	Use of timer for synchronization.	-	-
11. Procedural Abstraction	No use of procedures.	One procedure is defined and called.	More than one procedure defined.	There are procedures for code organization and re-use.
12. Sensors	No use of sensors.	One type of sensor is used.	Two types of sensors are used.	More than two types of sensors are used.
13. Drawing and Animation	No use of drawing and animation components.	Uses canvas component.	Uses ball component.	Uses image sprite component.
14. Maps	No use of city maps.	Use of a city map block.	Use of city map markers blocks.	-
15. Screens	Single screen with visual components, whose state is not changed programmatically.	Single screen with visual components, whose state is changed programmatically.	Three screens with visual components of which at least one is programmed to change state.	Four screens with visual components of which at least two are programmed to change state.

The CodeMaster rubric for assessing algorithms and programming concepts is composed of 15 items. It includes general algorithms and programming concepts, including operators, conditionals, loops, etc., as well as, mobile algorithms and programming concepts, including specific aspects related to the development of mobile features such as sensors, screens, etc. For each item performance levels are defined on ordinal scales, ranging from "item is not (or minimally) present" to advanced usage of the item. Aiming at the automation of the assessment, the performance levels are defined for automatically measurable characteristics based on the code of App Inventor projects.

The CodeMaster rubric can be regarded as reliable (Cronbach's alpha α=0.84) [49]. Concerning construct validity, there also exists an indication of convergent validity based on the results of a correlation and factor analysis. These results indicate that the rubric can be used for a valid assessment of algorithm and programming concepts of App Inventor programs as part of a comprehensive assessment completed by other assessment methods, such as observations [49]. The assessment using the CodeMaster rubric is automated by performing a static code analysis. The analysis is done by counting the kind and the number of command blocks used in App Inventor projects with respect to algorithms and programming concepts as defined in the rubric.

2.3 Item Response Theory – Graded Response Model

Item Response Theory (IRT) is a powerful tool in the quantitative processes of educational assessment as it allows analyzing item properties using falsifiable models. There are many mathematical models and to choose the adequate model the number of item response categories must be taken into account. Typically, for polytomous items, such as the CodeMaster rubric with three or more performance levels, the Graded Response Model (GRM) proposed by Samejima [57] is used. The GRM assumes that an item's response categories (denoted by k) are ordered among themselves and are arranged in order from smallest (1) to largest (m_i+ 1), where m_i+ 1 is the number of categories of the i-th item. Thus, the probability (P) of an individual j with the latent trait θ to satisfy the k-th category from item i is given by the expression:

$$P_{i,k}(\theta_j) = P^{+}_{i,k}(\theta_j) - P^{+}_{i,k+1}(\theta_j)$$

In order to get the probability $P^{+}_{i,k}(\theta_j)$ an expression from the 2-parameter logistic model can be used:

$$P^{+}_{i,k}(\theta_j) = \frac{1}{1 + e^{-Da_i(\theta_j - b_{i,k})}}$$

Where:

- i (item) = 1, 2, …, I
- k (category) = 0, 1, …, m_i
- j (individual) = 1, 2, …, n
- θ_j represents the latent trait of an individual j
- $P^{+}_{i,k}(\theta_j)$ is the probability of an individual j with the latent trait θ to satisfy the k-th category or higher from item i
- a_i represents the slope parameter of item i
- $b_{i,k}$ is the position parameter of the k-th category from item i, measured on the same scale as the latent trait (θ)
- D is a scale factor, constant and equal to 1

From the definition as categories are arranged in order from smallest to largest, the b's values representing the position parameter should be:

$$b_{i,1} \leq b_{i,2} \leq \cdots \leq b_{i,m_i}$$

Samejima [57] also defined that $P^{+}_{i,0}(\theta_j)$ — the threshold parameter for the lowest category, equals 1, and $P^{+}_{i,m+1}(\theta_j)$ — the probability of answering above the highest category, is zero:

$$P^{+}_{i,0}(\theta_j) = 1$$

$$P^{+}_{i,m+1}(\theta_j) = 0$$

As a result, the b parameters representing position can be interpreted as the threshold of passing from a lower to a higher performance level (Figure 3).

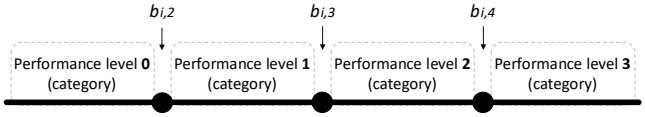

Figure 3: Position parameters (b's) for items with 4 adjacent difficulty performance levels (as in the CodeMaster rubric).

The position of items and their categories can be analyzed using the estimated values of b parameters on the same scale. Therefore, items that present b parameter values far below the average are considered "easy" as they result in a high probability of an average individual to satisfy the item's category. Similarly, items that present high b parameters far above average are considered "difficult", because of the low probability of an average individual to satisfy the item's category.

3 RESEARCH METHODOLOGY

Adopting the Goal Question Metric approach [58], the objective of this study is defined as to analyze the 'demonstrated difficulty' of algorithms & programming concepts of App Inventor projects based on the CodeMaster rubric [49]. Here the term 'demonstrated difficulty' is defined as the volition, incentive, and opportunity to apply programming concepts in an App Inventor project shared via App Inventor Gallery, on which no further background information on the authors is provided.

Initially, we use data collected in the form of publicly available and accessible projects from the App Inventor Gallery in June 2018. As a result, we use a dataset containing the source-code from 88,864 App Inventor projects. We automatically assessed these projects using the CodeMaster tool with respect to algorithms & programming concepts by extracting them from the source code through static code analysis. Out of the 88,864 projects, 88,812 were successfully assessed with the CodeMaster tool. 52 projects failed to be analyzed due to technical difficulties. The collected data were pooled in a single sample to analyze the difficulty of the items. The dataset was analyzed using the mirt package from the R programming language [59].

In order to analyze the item properties, we use the IRT Gradual Response Model proposed by Samejima [57]. This analysis is done by estimating the correspondence between an unobserved latent trait (the volition, incentive, and opportunity to apply computing concepts), and observable evidence (the assessed App Inventor projects).

Verifying unidimensionality. In order to use the unidimensional GRM, it is necessary to assure that the instrument can be analyzed by a single predominant dimension. Therefore, we performed a parallel analysis with scree plot and full information factor analysis beforehand (Figure 4) [49].

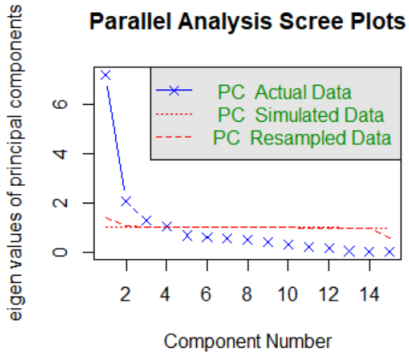

Figure 4: Parallel analysis [49]

The parallel analysis assumes that every dimension above the red line can be considered a relevant dimension. Thus, the results suggest that the instrument may contain 3 dimensions (Figure 4). However, there is a predominant dimension, indicating that the instrument can be analyzed by a single predominant dimension [49]. When performing the full information factor analysis [49], we also observed that when considering a single dimension, all factor loadings were greater than 0.3, which indicates that the items are related to this predominant dimension, except by the item "Maps" which presented a 0.262 factor loading (please see [49] for a detailed analysis). Despite the factor loading of this item being slightly less than 0.3 we decided to keep it in the analysis as this item may be underrepresented in our dataset [49]. In addition, we calculated the test variance. For acceptable calibration, the first dimension should account for at least 20% of the test variance [60]. We obtained a variance explained by the first-dimension of 53% characterizing the strong unidimensionality of the instrument as required by the IRT model used in this study.

4 ANALYSIS

In order to analyze the properties of the items in the CodeMaster rubric, we use the Gradual Response Model (GRM) [57] to estimate the slope (a) parameter and position (b's) parameters for each item. The metric is established by setting population parameters to average = 0 and standard deviation = 1. Since the CodeMaster rubric contains polytomous items, several b parameters are estimated (b_2, b_3, and b_4) to differentiate the passage from one score to another. In this regard, b_2 represents the difficulty of achieving score 1 on any item, b_3 represents the difficulty of achieving score 2 on any item, and b_4 represents the difficulty of achieving score 3 on any item. Consequently, items on a 2-point ordinal scale (without a description for category 3) also do not present a parameter b_4 (e.g., item variables). In IRT, *a* and *b* parameters can theoretically assume any real value between $-\infty$ and $+\infty$. However, a negative value for *a* parameter is not expected. Typically values above 1.0 are considered good, as they indicate that the item discriminates well learners with different abilities. In this study, *b* parameters are the main indicators to be analyzed, as they indicate the position of the item. For *b* parameters, values close to or within the range [-5, 5] are expected, with negative values indicating that an item is positioned below average and positive values indicating above average.

Table 3: Parameters estimated with standard errors (SE)

Item (i)	a (SE)	b_2 (SE)	b_3 (SE)	b_4 (SE)
1. Operators	3.08 (0.02)	-0.06 (0.01)	0.21 (0.01)	0.47 (0.01)
2. Variables	2.97 (0.02)	-0.83 (0.01)	-0.01 (0.01)	n.a.
3. Strings	1.66 (0.01)	-0.57 (0.01)	0.94 (0.01)	n.a.
4. Naming	1.68 (0.01)	-0.31 (0.01)	0.07 (0.01)	1.89 (0.01)
5. Lists	1.24 (0.01)	1.49 (0.01)	2.00 (0.02)	5.20 (0.07)
6. Data persist.	1.57 (0.02)	1.82 (0.02)	1.90 (0.02)	3.36 (0.04)
7. Events	2.88 (0.02)	-1.65 (0.01)	-0.90 (0.01)	-0.47 (0.01)
8. Loops	1.77 (0.03)	2.14 (0.02)	2.29 (0.02)	2.57 (0.03)
9. Conditional	2.32 (0.02)	0.34 (0.01)	0.80 (0.01)	1.57 (0.01)
10. Synch.	2.81 (0.03)	0.89 (0.01)	n.a.	n.a.
11. Proced. Abstraction	3.18 (0.03)	0.99 (0.01)	1.08 (0.01)	1.19 (0.01)
12. Sensors	1.53 (0.01)	0.64 (0.01)	2.77 (0.02)	4.39 (0.05)
13. Drawing and Anim.	1.32 (0.01)	0.82 (0.01)	1.25 (0.01)	1.45 (0.01)
14. Maps	0.65 (0.14)	11.36 (2.41)	n.a.	12.46 (2.66)
15. Screens	1.19 (0.01)	-2.53 (0.02)	0.89 (0.01)	1.10 (0.01)

In general, most items were well estimated, with slope (*a*) parameter values above 1 (Table 3). In addition, the values of the position parameters (b_2, b_3, and b_4) are within the range [-5, 5]. Only the item *lists* and *maps* presented parameter b_4 values above 5. Standard errors (SE) of each *b* parameter present similar results and are in low magnitude, therefore, presenting no estimation problem, with exception of the item *maps*, which presents SEs in an order of magnitude higher than the SEs of all items parameters. The reason may be that in our dataset map blocks are very rarely used (about 0,1% of the projects) as they had been added more recently to the App Inventor environment. Thus, the parameters of item *maps* cannot be used for the interpretation of positioning.

Analyzing the results, it can be inferred that the easiest category to satisfy is the first category of item 15 (*screens*), as it presents the smallest *b* parameter (b_2 = -2.53). On the other hand, obtaining three points for the item *lists* is more difficult than any other item as it presents the highest value for a *b* parameter (b_4 = 5.20). And, although item 14 (*maps*) presents high *b* parameters, this item is not considered here as it presents an estimation problem with SE in an order of magnitude higher than all other items' SE.

Based on the estimated *b* parameters (b_2, b_3, and b_4) presented in Table 3 the items are placed on a wright map with a (0.1) scale, i.e., with average = 0 and standard deviation = 1 (Figure 5). The

scale is an "arbitrary" scale, where the relations of order and positions between its points are most important and not necessarily its magnitude. The wright map provides a general picture by placing the positioning of demonstrated difficulty of the items on the same measurement scale as the abilities with respect to algorithms and programming concepts based on assessed App Inventor projects used as observable evidence. The left side shows the distribution of the measured ability in App Inventor projects from the most able ones at the top to the least able ones at the bottom. The right side shows items distributed from the most difficult ones at the top to the least difficult ones at the bottom (Figure 5).

From the placement of items on the scale, we can infer that an item with a *b* parameter estimated at 1.5 is 1.5 standard deviations above the average ability. Thus, such item is more difficult than all items that are placed below point 1.5 on the scale. In the context of programming with App Inventor, the easiest items include item 7 (*events*) and item 15 (*screens*) (Figure 5), as these items have negative *b* parameters far below zero. These results are semantically consistent, as App Inventor encourages unlimited use of events and creating screens that change programmatically as an essential functionality of useful mobile apps [9].

The most difficult items include *lists*, *data persistence*, *loops*, and *sensors* (items 5, 6, 8, and 12 respectively). For example, the 3rd category on *lists* has the highest demonstrated difficulty parameter (b_4), being the most difficult to achieve among all items. Item *maps* parameters are not presented because the values are out of range of the wright map [-2.5, 5.5] and are not considered here. Although the *loops* item is also considered difficult, it is noteworthy that loop blocks in App Inventor programs are rarely used, because many iterative processes that would be expressed with loops in other programming languages are expressed as an event that performs a single step of the iteration every time it is triggered [9]. Thus, the demonstrated difficulty of *loops* may be poorly represented in the App Inventor dataset, as more than 94% of apps are assessed with 0 points regarding *loops* (see Figure 6). In other visual programming environments, such as Scratch, the usage of this concept and consequently the demonstrated difficulty may be different.

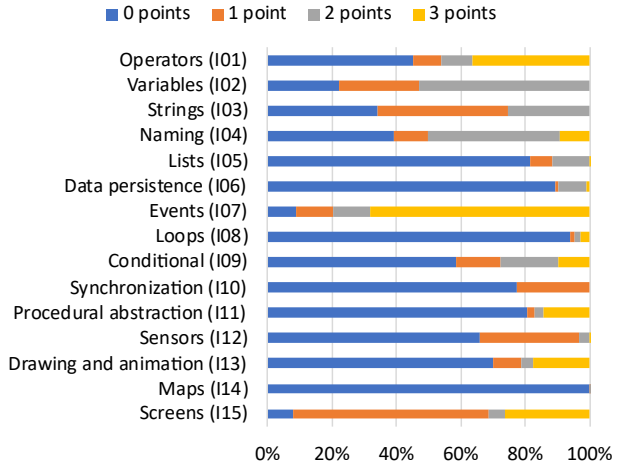

Figure 6: Frequency of the performance level score for each item

According to the estimated *b* parameters, the Item Characteristic Curves (ICC) for each item are plotted (Figure 7). While the theoretical range of a latent trait is from negative infinity to positive infinitive, for practical considerations the range

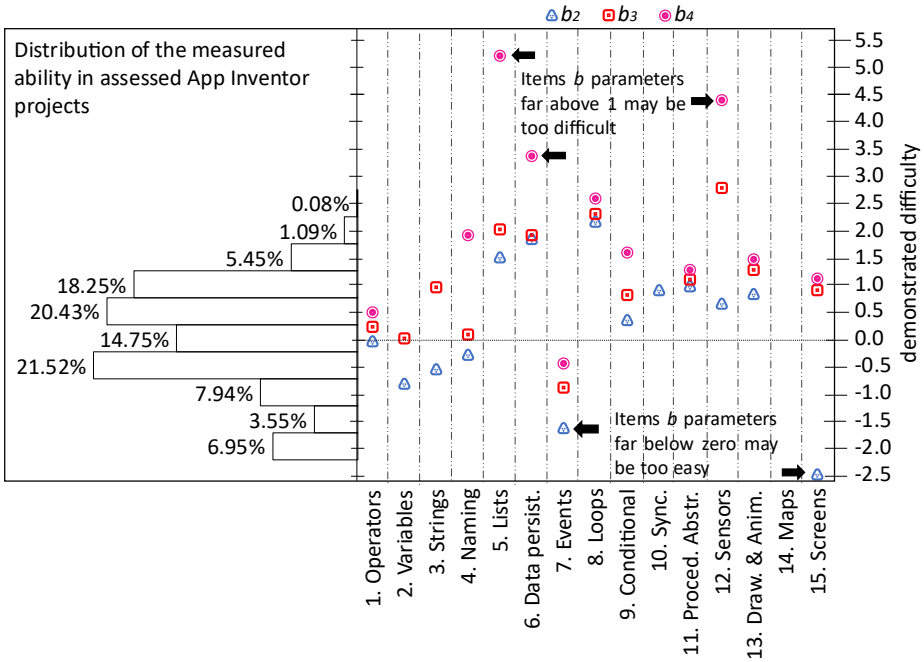

Figure 5: Wright map of the algorithms and programming items in App Inventor projects

of values can be limited from -4 (low) to +4 (high) on the x-axis. Thus, items with low demonstrated difficulty are placed closer to low latent trait values and high demonstrated difficulty items are placed closer to the high latent trait. Therefore, items that have high *b* parameters, which indicate high demonstrated difficulty, such as items 5 (*lists*), 6 (*data persistence*), 8 (*loops*), and 12 (*sensors*) present the curves dislocated to the right (Figure 7). In the same way, items with low demonstrated difficulty, such as items 7 (*events*) and 15 (*screens*) present curves dislocated to the left. Although item 14 (*maps*) presented the highest difficulty parameters (Table 3), and the "curve" is hidden above latent trait 4.0, these parameters presented standards errors in a high order of magnitude (Table 3). Consequently, the ICC for *maps* cannot be used for difficulty interpretation purposes.

Items with only three performance levels, such as item 2 (*variables*), 3 (*strings*), and 14 (*maps*) have fewer curves than the other items (Figure 7). This is because of the impossibility of satisfying a fourth category as no such performance level has been defined for these items (see Table 2). This also applies to items with two performance levels, such as item 10 (*synchronization*).

The P0 curve refers to the probability of satisfying category zero (or achieving score 0) for any item given the latent trait in the x-axis (Figure 7). Similarly, the P1, P2, and P3 curves refer to the probability of achieving scores 1, 2, and 3 respectively given the latent trait in the x-axis (Figure 7). Thus, the P0 is close to 1.0 for low latent trait values, as projects assessed with a "low" latent trait (the volition, incentive, and opportunity to apply computing concepts) have a probability close to 100% of achieving score 0. For example, presenting a latent trait less than -1.0 results in a bigger probability of achieving score 0 in item 3 (*strings*) than score 1. On the other hand, P0 is close to 0 for high latent trait values, as projects assessed with a "high" latent trait have a probability close to 0% of achieving score 0. For example, presenting a latent trait greater than the average (0.0) results in having a bigger probability in P1 for item 3 (*string*s), which is related to achieving score 1, than in P0, which is related to achieving score 0 for the same item (Figure 7).

Some items' curves are more attached than others, for example, the curves of item 2 (*variables*) are more attached than the curves of item 3 (*strings*) (Figure 7). This occurs because *b* parameters of *variables* are less distant than *b* parameters of *strings*, as the distance between b_2 and b_3, i.e., b_3 - b_2, of the item *variables* is 0.82, while their distance for the item *strings* is 1.51 (Table 3). This means that is easier to progress from "modifying or using predefined variables" to "creating and operating with variables", than progressing from "using string block to change the text of design component" to "creating and operating with strings", as defined in the CodeMaster rubric (Table 2). This is expected as operating with variables is easier than operating with strings, as strings can be broken apart to make new strings, or put together

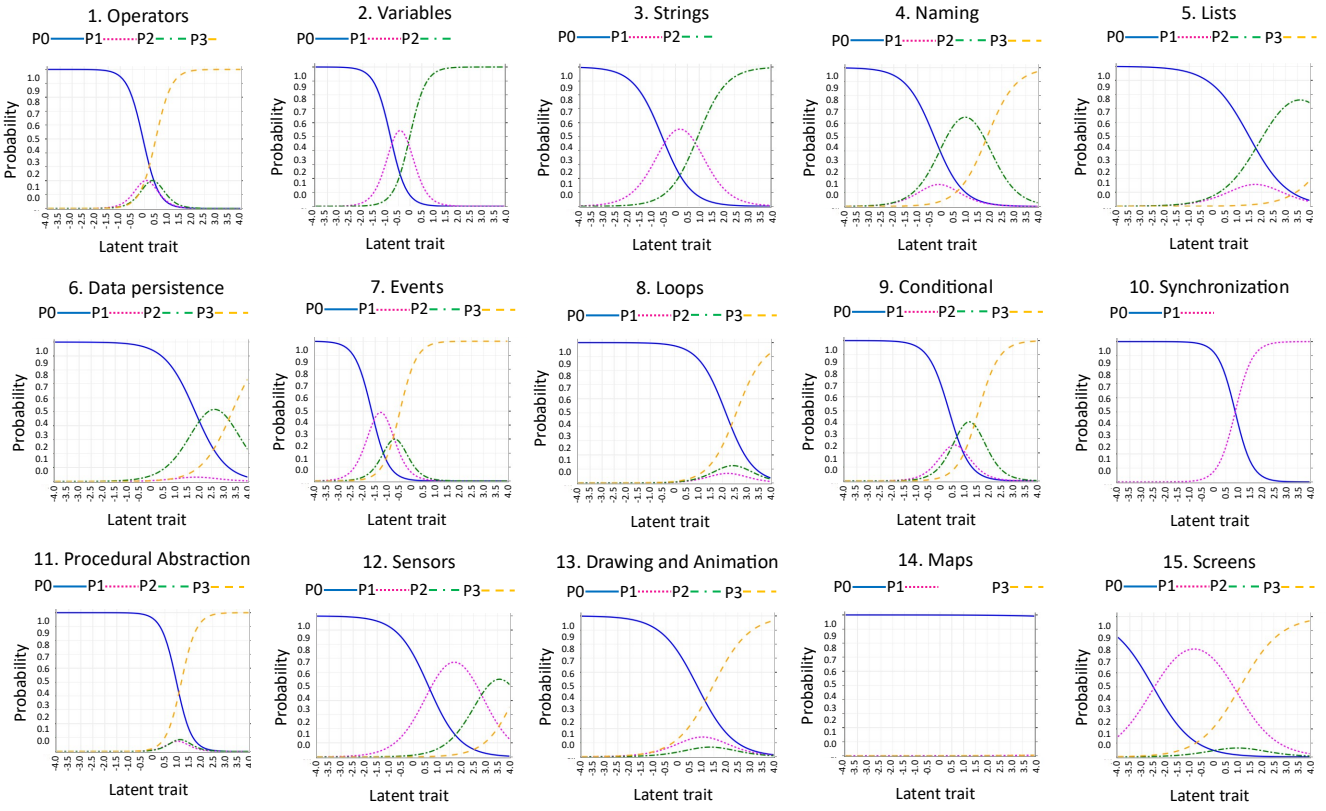

Figure 7: Item Characteristic Curves for each item

and make longer strings [61]. Similarly, this can also be observed regarding item 1 (*operators*) and 11 (*sensors*).

5 DISCUSSION

The results of the analysis provide an insight into the degree of demonstrated difficulty concerning algorithms and programming in the context of the development of apps with App Inventor (Table 4).

Table 4: Summarized results

	Demonstrated difficulty level		
Item (i)	**Low**	**Medium**	**High**
1. Operators		X	
2. Variables	X		
3. Strings	X		
4. Naming	X		
5. Lists			X
6. Data persistence			X
7. Events	X		
8. Loops			X
9. Conditional		X	
10. Synchronization		X	
11. Procedural Abstraction		X	
12. Sensors			X
13. Drawing and Animation		X	
14. Maps (excluded)			
15. Screens	X		

Variables, *strings*, *naming*, *events*, and *screens* (items 2, 3, 4, 7, and 15 respectively) are the easiest concepts when programming with App Inventor, as all probabilities curves are dislocated to the left (Figure 7). Items with medium demonstrated difficulty include *operators*, *conditional*, *synchronization*, *procedural abstraction*, and *drawing and animation* (item 1, 9, 10, 11, and 13 respectively) as the probability curves are close to the average (0.0) latent trait. The most difficult items are *lists*, *data persistence*, *loops*, and *sensors* (item 5, 6, 8, and 12 respectively) as the probability curves are dislocated to the right (Figure 7). This also confirms results presented by Xie and Abelson [34] indicating, for example, that apps that require data persistence (e.g., databases) represent more advanced artifacts. Some of the items with estimated high difficulty may be influenced by its infrequent use in App Inventor projects, e.g., *loops* rather than indicating the difficulty of understanding loops in general, and may be different when using other visual programming environments.

These results can be used as a systematic basis supported by data for the sequencing of computing instruction in K-12 by teaching the development of apps with App Inventor. For instance, a manual organization by computer science teachers may achieve a similar result. However, this work does not support the findings based on opinion but data. Based on the results of the scale placement (Figure 5) and the detailed demonstrated difficulty ICC (Figure 7), teaching algorithms and programming concepts with App Inventor should thus start with the creation of screens and events as well as the usage of strings, and variables and naming. Then on the next stage, the instructional design could cover operators and conditionals as well as synchronization and procedural abstraction, while only more advanced students should be presented with problems requiring lists, data persistence, and sensors, allowing them to follow a smooth pathway as they progress toward mastery of the skills with scaffolding support.

5.1 Threats to validity

Our study is subject to several threats to validity which have been handled in order to be minimized. One risk is related to grouping data as App Inventor projects come from various contexts in the worldwide App Inventor community, and no additional information about the creator history is available in the App Inventor Gallery. Another factor that may influence the usage of commands may be the tutorials and instructional units typically used as well as a considerable number of very simple App Inventor projects at the App Inventor Gallery. However, as typically App Inventor is used by novices and/or in the context of computing education in K-12, we consider this acceptable considering the large-scale sample. Another threat regarding the possibility of generalizing the results is related to the sample size and projects from only one repository. Yet, this risk is minimized by using a significant number of apps (over 88,000) from the repository for App Inventor project used by contributors from around the world.

Another risk is that the analysis based on the created code does not only assess whether a learner is able to achieve a certain item of the rubric, but also whether the learner is willing to do so. Therefore, we also restrict the interpretation of the "difficulty" of items in this study to the "demonstrated difficulty" defined as the volition, incentive, and opportunity to apply programming concepts in an App Inventor project shared via App Inventor Gallery. For measurement we used the CodeMaster rubric that was systematically defined and validated using Classical Test Theory with results reported in Alves et al. [49], indicating the reliability and validity of the rubric for the assessment of algorithm and programming concepts of App Inventor projects. Automating the assessment of the App Inventor projects with the CodeMaster tool further reduced the risks of reliability issues which may have caused through manual assessment. In order to mitigate threats concerning the research methodology, we adopted the GQM approach for measurement [58] and selected appropriate statistical techniques for the analysis [45], performing also necessary tests with respect to the characteristics of the dataset to assure their adequacy.

6 CONCLUSION

In this article we presented the results of an analysis of the demonstrated difficulty of general mobile algorithms and programming concepts based on App Inventor projects. Considering the difficulty of items, we identified that events and variables are the easiest items when programming with App Inventor, while the most difficult items are loops, data persistence, and lists. Comparing these results to analyses based on other block-based languages, we can observe based on data that the

difficulty of achieving performance levels of certain items may depend on the specific programming language, and, thus the programming environment to be adopted has to be explicitly considered in the instructional design of computing education. The results of this analysis can be used to systematically discuss and improve the sequencing of instructional units for teaching algorithms and programming with App Inventor by adopting scaffolding techniques.

ACKNOWLEDGMENTS

We would like to thank all researchers from the MIT App Inventor team, who provided support for the access to the App Inventor Gallery. The authors would also like to thank the anonymous referees for their valuable comments and helpful suggestions. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 and by the Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil (CNPq) - Grant No.: 302149/2016-3.

REFERENCES

[1] S. Grover and R. Pea. 2013. Computational thinking in K–1: A review of the state of the field. Educational Researcher, 42, 1, 38–43. DOI:https://doi.org/10.3102/0013189X12463051

[2] P. Hubwieser, M. N. Giannakos, M. Berges, T. Brinda, I. Diethelm, J. Magenheim, Y. Pal, J. Jackova, and E. Jasute. 2015. A Global Snapshot of Computer Science Education in K-12 Schools. In Proceedings of the ITiCSE on Working Group Reports. Association for Computing Machinery, New York, NY, USA, 65–83. DOI:https://doi.org/10.1145/2858796.2858799

[3] S. Y. Lye and J. H. L. Koh. 2014. Review on teaching and learning of computational thinking through programming: What is next for K-12? Computers in Human Behavior, 41, C, 51–61. DOI:https://doi.org/10.1016/j.chb.2014.09.012

[4] M. Webb, N. Davis, and T. Bell. 2017. Computer science in K-12 school curricula of the 2lst century: Why, what and when?. Education and Information Technolologies, 22, 445–468. DOI:https://doi.org/10.1007/s10639-016-9493-x

[5] CSTA. 2016. K-12 Computer Science Framework. Retrieved September 2, 2020 from https://k12cs.org/

[6] CAS. 2015. Computing at School. Retrieved September 1, 2020, from https://www.computingatschool.org.uk/

[7] SBC. 2018. Brazilian Computer Society Guidelines for Computing Education in K-12. Retrieved September 3, 2020, from https://www.sbc.org.br/educacao/diretoria-de-educacao-basica

[8] S. Grover, S. Basu, and P. Schank. 2018. What We Can Learn About Student Learning From Open-Ended Programming Projects in Middle School Computer Science. In Proceedings of the 49th ACM Technical Symposium on Computer Science Education. Association for Computing Machinery, NY, USA, 999-1004. DOI:https://doi.org/10.1145/3159450.3159522

[9] F. Turbak, M. Sherman, F. Martin, D. Wolber, and S. C. Pokress. 2014. Events First Programming in App Inventor. Journal of Computing Sciences in Colleges, 29, 6, 81-89.

[10] S. Papadakis, M. Kalogiannakis, V. Orfanakis, and N. Zaranis. 2017. The appropriateness of Scratch and App Inventor as educational environments for teaching introductory programming in primary and secondary education. International Journal of Web-Based Learning and Teaching Technologies, 12, 4, 58-77. DOI:https://doi.org/10.4018/IJWLTT.2017100106

[11] D. Weintrop. 2019. Block-based Programming in Computer Science Education. Communications of the ACM, 62, 8, 22-25. DOI: http://doi.org/10.1145/3341221

[12] MIT. 2020. MIT App Inventor. About us. Retrieved September 1, 2020 from http://appinventor.mit.edu/explore/about-us.html

[13] D. Wolber, H. Abelson, and M. Friedman. 2014. Democratizing Computing with App Inventor. GetMobile: Mobile Computing and Communications. 18, 4, 53–58. DOI:https://doi.org/10.1145/2721914.2721935

[14] E.W. Patton, M. Tissenbaum, and F. Harunani. 2019. MIT App Inventor: Objectives, Design, and Development. In Kong SC., Abelson H. (eds), Computational Thinking Education, Springer. DOI:https://doi.org/10.1007/978-981-13-6528-7_3

[15] S. B. Fee and A. M. Holland-Minkley. 2010. Teaching computer science through problems, not solutions. Computer Science Education, 20, 2, 129-144. DOI:https://doi.org/10.1080/08993408.2010.486271

[16] M. Tissenbaum, J. Sheldon, and H. Abelson. 2019. From Computational Thinking to Computational Action. Communications of the ACM, 62, 3, 34-36. DOI:https://doi.org/10.1145/3265747

[17] H. Khosravi, S. Sadiq, and D. Gasevic. 2020. Development and Adoption of an Adaptive Learning System. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education, Association for Computing Machinery, New York, NY, USA, 58–64. DOI:https://doi.org/10.1145/3328778.3366900

[18] J. Bennedsen and M. E. Caspersen. 2007. Failure Rates in Introductory Programming, ACM SIGCSE Bulletin, 39, 2, 32-36. DOI:https://doi.org/10.1145/1272848.1272879

[19] J. Bennedsen and M. E. Caspersen. 2019. Failure rates in introductory programming: 12 years later. ACM Inroads 10, 2, 30–36. DOI:https://doi.org/10.1145/3324888

[20] J. Rogalski and R. Samurçay. 1990. Acquisition of programming knowledge and skills. In J.M.Hoc, T.R.G. Green, R. Samurçay, & D.J. Gillmore (eds.), Psychology of programming, Academic Press. DOI:https://doi.org/10.1016/B978-0-12-350772-3.50015-X

[21] K. Brennan and M. Resnick. 2012. New frameworks for studying and assessing the development of computational thinking. In Proceedings of the Annual Meeting of the American Educational Research Association, Vancouver, Canada.

[22] R. D. Pea and D. M. Kurland. 1984. On the cognitive effects of learning computer programming. New Ideas in Psychology, 2, 2, 137–168. DOI:https://doi.org/10.1016/0732-118X(84)90018-7

[23] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. 2009. Scratch: programming for all. Communications of the ACM 52, 11, 60–67. DOI:https://doi.org/10.1145/1592761.1592779

[24] J. van Patten, C. I. Chao, and C. M. Reigeluth. 1986. A review of strategies for sequencing and synthesizing instruction. Review of Educational Research, 56, 4, 437-471. DOI:https://doi.org/10.3102/00346543056004437

[25] J. S. Bruner. 1966. Toward a theory of instruction. Harvard University Press.

[26] G. R. Morrison, S. M. Ross, and J. E. Kemp. 2010. Designing Effective Instruction, 6th ed. John Wiley & Sons.

[27] J. Sweller, J. J. G. van Merrienboer, and F. G. W. C. Paas, 1998. Cognitive Architecture and Instructional Design. Educational Psychology Review, 10, 251–296. DOI:https://doi.org/10.1023/A:1022193728205

[28] C. Dede. 1986. A review and synthesis of recent research in intelligent computer-assisted instruction. International Journal on Man-Machine Studies, 24, 329-353. DOI:https://doi.org/10.1016/S0020-7373(86)80050-5

[29] O. Vainas, Y. Ben-David, R. Gilad-Bachrach, M. Ronen, O. Bar-Ilan, R. Shillo, G. Lukin, D. Sitton, D. 2019. Staying in The Zone: Sequencing Content in Classrooms Based on The Zone of Proximal Development. In Proceedings of the 12th International Conference on Educational Data Mining, Montreal, Canada, 659 – 662.

[30] C. M. Reigeluth. 1999. The Elaboration theory: Guidance for scope and sequence decision. In C. Reigeluth (ed.) Instructional-Design Theories and Models (vol.II), Erlbaum Associates.

[31] I. Li, F. Turbak, and E. Mustafaraj. 2017. Calls of the Wild:Exploring Procedural Abstraction in App Inventor. In Proceedings of the IEEE Blocks and Beyond Workshop. Raleigh, NC, USA, 79-86. DOI:http://doi.org/10.1109/BLOCKS.2017.8120417

[32] B. Xie, I. Shabir, and H. Abelson. 2015. Measuring the programmatic sophistication of app inventor projects grouped by functionality. Retrieved September 2, 2020 from http://web.mit.edu/bxie/www/thesis.pdf

[33] Y. Park and Y. Shin. 2019. Comparing the Effectiveness of Scratch and App Inventor with Regard to Learning Computational Thinking Concepts. Electronics, 8, 1269. DOI:http://doi.org/10.3390/electronics8111269

[34] B. Xie and H. Abelson. 2016. Skill progression in MIT app inventor. In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, Cambridge, GB, 213-217. DOI:http://doi.org/10.1109/VLHCC.2016.7739687.

[35] C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein. 2012. Modeling how students learn to program. In Proceedings of the 43rd ACM Technical Symposium on Computer Science Education. Association for Computing Machinery, New York, NY, USA, 153–160. DOI:https://doi.org/10.1145/2157136.2157182

[36] S. Grover and S. Basu. 2017. Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and Boolean Logic. In Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education, Association for Computing Machinery. New York, NY, USA, 267–272. DOI:https://doi.org/10.1145/3017680.3017723

[37] J. Moreno-León, G. Robles, and M. Román-González. 2020. Towards data-driven learning paths to develop computational thinking with Scratch. IEEE

Transactions on Emerging Topics in Computing, 8, 1, 193-205. DOI:https://doi.org/10.1109/TETC.2017.2734818.

[38] K. M. Rich, C. T. Strickland, T. A. Binkowski, T. A. Moran and D. Franklin. 2017. K-8 learning trajectories derived from research literature: Sequence, repetition, conditionals. In Proceedings of the ACM Conference on International Computing Education Research, Association for Computing Machinery, New York, NY, USA, 182-190. DOI:https://doi.org/10.1145/3105726.3106166

[39] K. M. Rich, C. T. Strickland, T. A. Binkowski and D. Franklin. 2018. Decomposition: A K-8 computational thinking learning trajectory. In Proceedings of the 2018 ACM Conference on International Computing Education Research, Association for Computing Machinery, New York, NY, USA, 124-132. DOI:https://doi.org/10.1145/3230977.3230979

[40] K. M. Rich, C. T. Strickland, T. A. Binkowski and D. Franklin. 2019. A K-8 debugging learning trajectory derived from research literature. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education, Association for Computing Machinery, New York, NY, USA, 745-751. DOI:https://doi.org/10.1145/3287324.3287396

[41] K. Seiter and B. Foreman. 2013. Modeling the learning progressions of computational thinking of primary grade students. In Proceedings of the 9th Annual International ACM Conference on International Computing Education Research, Association for Computing Machinery, New York, NY, USA, 59–66. DOI: https://doi.org/10.1145/2493394.2493403

[42] D. Franklin, G. Skifstad, R. Rolock, I. Mehrotra, V. Ding, A. Hansen, D. Weintrop, and D. Harlow. 2017. Using Upper-Elementary Student Performance to Understand Conceptual Sequencing in a Blocks-based Curriculum. In Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education. Association for Computing Machinery, New York, NY, USA, 231–236. DOI:https://doi.org/10.1145/3017680.3017760

[43] N. Lytle, V. Cateté, D. Boulden, Y. Dong, J. Houchins, A. Milliken, A. Isvik, D. Bounajim, E. Wiebe, and T. Barnes. 2019. Use, Modify, Create: Comparing Computational Thinking Lesson Progressions for STEM Classes. In Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education. Association for Computing Machinery, New York, NY, USA, 395–401. DOI:https://doi.org/10.1145/3304221.3319786

[44] J. Krugel et al. 2020. Automated Measurement of Competencies and Generation of Feedback in Object-Oriented Programming Courses. In Proceedings of the IEEE Global Engineering Education Conference (EDUCON), Porto, Portugal, 329-338. DOI:https://doi.org/10.1109/EDUCON45650.2020.9125323.

[45] R. J. De Ayala. 2009. The theory and practice of item response theory. Guilford Press.

[46] J. E. Carlson and M. van Davier. 2017. Item Response Theory. In Advancing Human Assessment, eds. Bennet & van Davier, Springer.

[47] M. Berges and P. Hubwieser. 2015. Evaluation of Source Code with Item Response Theory. In Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education, Association for Computing Machinery, New York, NY, USA, 51–56. DOI:https://doi.org/10.1145/2729094.2742619

[48] M. Kramer, D. A. Tobinski, and T. Brinda. 2016. On the way to a test instrument for object-oriented programming competencies. In Proceedings of the 16th Koli Calling International Conference on Computing Education Research. Association for Computing Machinery, New York, NY, USA, 145–149. DOI:https://doi.org/10.1145/2999541.2999544

[49] N. da C. Alves, C. Gresse von Wangenheim, J. C. R. Hauck and A. F. Borgatto. 2020. A large-scale evaluation of a rubric for the automatic assessment of algorithms and programming concepts. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education. Association for Computing Machinery, New York, NY, USA, 556–562. DOI:https://doi.org/10.1145/3328778.3366840

[50] C. Gresse von Wangenheim, J. C. R. Hauck, M. F. Demetrio, R. Pelle, N. da C. Alves, H. Barbosa, L. F. Azevedo. 2018. CodeMaster – Automatic Assessment and Grading of App Inventor and Snap! Programs. Informatics in Education, 17, 1, 117-150. DOI:https://doi.org/ 10.15388/infedu.2018.08

[51] J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck. 2011. The state of the art in end-user software engineering. ACM Computing Surveys, 43, 3, Article 21. DOI:https://doi.org/10.1145/1922649.1922658

[52] E. Mustafaraj, F. Turbak, and M. Svanberg. 2017. Identifying Original Projects in App Inventor. In Proceedings of the 30th International Florida Artificial Intelligence Research Society Conference, Marco Island, FL, USA. 567-572.

[53] N. da C. Alves, C. Gresse von Wangenheim, and J. C. R. Hauck. 2019. Approaches to assess computational thinking competences based on code analysis in K-12 education: A systematic mapping study. Informatics in Education, 18, 1, 17-39. DOI: https://doi.org/ 10.15388/infedu.2019.02

[54] R. M. Branch. 2010. Instructional Design: The ADDIE Approach. Springer.

[55] H. Goodrich. 1996. Understanding Rubrics. Educational Leadership, 54, 4, 14–18.

[56] M. Sherman and F. Martin. 2015. The assessment of mobile computational thinking. Journal of Computing Sciences in Colleges, 30, 6, 53–59.

[57] F. A. Samejima. 1969. Estimation of latent ability using a response pattern of graded scores. Psychometric Monograph, 34, 4, 2-17.

[58] V. R. Basili, G. Caldiera, and H. D. Rombach. 1994. The goal question metric approach. In Encyclopedia of Software Engineering, John Wiley & Sons.

[59] R. P. Chalmers. 2012. Mirt: A multidimensional item response theory package for the R Environment. Journal of Statistical Software, 48, 6, 1–29.

[60] M. D. Reckase. 1979. Unifactor latent trait models applied to multifactor tests: Results and implications. Journal of educational statistics, Sage Publications CA: Thousand Oaks, 4, 3, 207–230.

[61] LEGO. 2018. Lego Education Documentation. Retrieved September 2, 2020 from: https://makecode.mindstorms.com/types/string

[62] J. S. Santos, W. L. Andrade, J. Brunet, and M. R. Araujo Melo. 2020. A Systematic Literature Review of Methodology of Learning Evaluation Based on Item Response Theory in the Context of Programming Teaching. In Proceedings of the IEEE Frontiers in Education Conference (FIE), Uppsala, Sweden, 1-9. DOI:http://doi.org/10.1109/FIE44824.2020.9274068.