A systematic literature review on knowledge tracing in learning programming

Philip I.S. Lei*† and António José Mendes*
*Univ Coimbra, CISUC, Dep. of Informatics Engineering, Coimbra, Portugal
†Computing Program, Macao Polytechnic Institute, Macao SAR, China
Email: philiplei@ipm.edu.mo, toze@dei.uc.pt

***Abstract*—This Research Full Paper presents a systematic review on knowledge tracing of learning programming based on student performance data in exercises.**

Programming has become an essential skill to solve real-world problems in modern engineering disciplines. However, when students start to learn how to program, they face a lot of challenges in acquiring various programming knowledge and skills. While it is beneficial to customise learning material to fit individual learning progress, the widely different learning pace of students in an introductory programming course has made it impractical for teachers to track the knowledge acquisition of individual. Hence, many recent works take a data-driven approach to model students' learning progress based on the performance data in programming-related exercises, which include the submitted program codes and answers to closed-ended programming exercises. By analyzing these performance data, a system can evaluate the students' knowledge level of various concepts and skills in programming.

This paper performs a systematic review and reports key information about recent works on programming knowledge tracing based on student performance data. An overview of the different choices of knowledge representation, domain knowledge model, performance measure and knowledge tracing algorithms is provided. The nature and granularity of knowledge components and the relationships between them are compared across the reviewed works. The different choice of programming knowledge representation leads to varied methods to assess knowledge levels from empirical performance data in programming-related exercises. Two broad categories of works are identified. The first is to overlay a student model on the domain knowledge model, and the student knowledge levels are updated in distinct time steps. The second trains temporal knowledge tracing models to predict students' future performance based on their performance in previous exercises. In addition, this review discusses the distinct challenges in knowledge tracing in programming education. It also points out limitations in current works and opportunities to improve knowledge tracing in learning programming.

***Index Terms*—systematic review, computer science, programming education, student modelling, knowledge tracing**

I. Introduction

With the increasing amount of data and higher model complexity in engineering problems, programming has become an essential skill to solve real-world problems in modern engineering disciplines. However, novice programmers face a lot of challenges in acquiring various programming knowledge and skills [1]. The difficulty in overcoming these challenges often results in low self-motivation [2] and a high drop-out rate in introductory programming courses [3].

While there is evidence for the pedagogical benefit of customising learning material to fit individual learning progress, the widely different learning pace of students in an introductory programming course has made it impractical for teachers to track the knowledge acquisition status of individual. Hence, there have been many recent works that take a data-driven approach to model student learning progress based on performance in programming-related exercises, which include the submitted program codes and answers to closed-ended programming exercises.

An essential step in designing a system to trace programming knowledge is to first break down the domain knowledge of programming into smaller units and trace the acquisition of these units while students attempt a sequence of exercises. In this review, we refer to these units as *knowledge components* (KCs), which are defined as "an acquired unit of cognitive function or structure that can be inferred from performance on a set of related tasks" in the Knowledge-Learning-Instruction framework [4]. We will sometimes also use the term 'skill' interchangeably.

Knowledge tracing works typically build a *student model* of a student's mastery of some pre-determined knowledge components, e.g. programming concepts and language constructs, according to whether the student produces correct responses in a sequence of exercises. Common models employed in the literature include probabilistic, statistical and artificial neural network models. The student model can be used to effectively customise the learning process in intelligent tutoring systems (ITS), to automatically generate hints to help students struggling in solving programming exercises, and to identify students in need of human tutor intervention.

This paper performs a systematic literature review and reports key information about recent works on student modelling and knowledge tracing based on student performance in closed-ended and open-ended programming exercises. It explores the numerous design decisions made in previous works and provides an overview of the different choices of knowledge components, domain knowledge model, performance measure and knowledge tracing algorithms. In addition, this review discusses the distinct challenges in knowledge tracing in programming education and the major approaches in framing the problem in the perspective of established methods of student modelling and knowledge tracing. This paper also points out limitations in current works and opportunities to

improve knowledge tracing in learning programming.

II. Related works

Several related literature reviews can be found in recent literature. Crow et al. [5] performed a systematic literature review of ITS for programming education. This work focuses on the various supported features for learning programming. On the other hand, Chrysafiadi et al. [6], and more recently Abyaa et al. [7], surveyed student modelling, describing student characteristics that are captured and the modelling techniques applied in different disciplines. They found that researchers preferred to use the overlay model, which traces students' mastery of knowledge components organized in a domain knowledge model.

The current study differs from the above reviews in its emphasis on *programming* knowledge representation. This allows us to discuss in more depth the issues in adapting common student modelling techniques to handle the particular nature of programming knowledge. Moreover, we can review the existing knowledge assessment methods for a specialized type of student submission – program codes.

III. Methodology

In this section, we explain the research questions we aim to answer in this review. The search strategy and the inclusion and exclusion criteria are designed to fulfill this aim.

A. Research questions

The research questions of this review are formulated as follows.

- What representation of programming knowledge is used?
- What is the role of a domain knowledge model in estimating knowledge level in different programming skills?
- What algorithms are used to trace knowledge acquisition?

Research question 1 addresses the foundation of knowledge tracing. It defines the nature and granularity of knowledge components. Choice of knowledge representation also leads to different detection means (e.g. automatic vs. expert input) and other nuances in the student model.

Some works in the literature build a student model as an overlay on top of the *domain knowledge model*, which embodies different relationships between knowledge components. Research question 2 examines how knowledge components are inter-related in the domain knowledge model, and how this affects the knowledge level update methods later applied.

Research question 3 explores diverse student modelling algorithms based on empirical student performances in exercises. We restrict ourselves to aspects of student modelling that are related to knowledge acquisition. In particular, we ignore affective student models and paid little attention to behavior data, demographic data and academic performance [8] [9].

B. Search strategy

We searched the ACM Digital Library and the IEEE Xplore databases because they cover most relevant venues for computer science education. The search string contained two main concepts, namely "knowledge tracing" and "learning programming". After a preliminary search, we realized that programming knowledge is traced in different applications, including modelling student knowledge to adapt learning material in ITS, educational data mining (e.g. predicting student performance), and studies of knowledge tracing algorithms per se. Moreover, the two main concepts were described with different wording. Therefore, we bootstrapped the search process from four seed articles [10]–[13], and collected synonyms for each concept from them. The search string used is as follows.

```
( "knowledge tracing" OR "student model" OR
"learning curve analysis" ) AND ( "learning
programming" OR "programming learning" OR
"introductory programming" OR "programming
tutor" OR "programming exercise" OR "how to
program" OR "CS1" OR "novice programmer" )
```

We performed search on both the abstract and full text of papers in the databases on March 2021. ACM Digital Library returned 80 results while IEEE Xplore database returned 98 results.

C. Inclusion and exclusion criteria

The above search string was deliberately built to cover the wide context in which programming knowledge is traced in the literature. As a consequence, it was expected that a large number of retrieved papers would be irrelevant to our theme of knowledge tracing in learning programming. Thus, we delineated the following inclusion and exclusion criteria to determine the relevant works from the search results. To determine whether a paper met the inclusion and exclusion criteria, the abstract and introduction were first read in full to grasp the overall aim of the work. Next, text searches were performed to locate the paragraphs that contain the search key phrases. These paragraphs were read to decide the role of knowledge tracing in the works. After this step, we can omit, for example, works that only mention knowledge tracing as related works, but do not cover knowledge tracing of programming students in the main contribution. Finally, we read the full text to determine relevancy and then to extract data.

The inclusion criteria were written to identify works that trace the progressive acquisition of programming knowledge based on empirical student performance in programming-related closed-ended exercises or open-ended coding exercises. Both scripted (e.g. Java, Python) and non-scripted programming language (e.g. block-based programming) were included. The inclusion criteria are as follows.

- The paper models the change of programming knowledge status while students learn programming.
- Domain knowledge of programming is broken down into components that can be traced from empirical performance of students in exercises.

- The paper provides some detail on how to update the student model from student responses to programming-related exercises.

Papers that only discuss architecture issues but lack details of tracing different concepts or skills in the student model were omitted. In addition, studies that examine the programming behavior of students [14] rather than assessing their knowledge level were excluded. The exclusion criteria are listed below.

- The paper does not include specific information on programming knowledge representation.
- The paper models the problem-solving behavior of programming students, but does not estimate the change of student knowledge.
- The paper predicts the overall performance of programming (e.g. estimate course outcome), but does not model relevant knowledge on a finer level.
- The paper is not written in English, or not related to programming (e.g. physics).

After carrying out the inclusion and exclusion criteria, we obtained 12 relevant papers as the systematic sample. See Table I.

IV. Results

A. Knowledge components

Programming is a complicated cognitive process that requires a wide spectrum of conceptual understanding and skills. According to the Box Model proposed by Izu et al. [23], these include understanding textual structure, run-time behavior, and function/purpose of program codes. The scope of relevant code may span from an expression, a block of statements, to a disperse set of statements. Existing works in programming knowledge tracing usually restrained to trace KCs that involve a small textual structure (e.g. within a few lines of code). Many studies in the systematic sample focus on lower-level concrete concepts (e.g. variable, arithmetic expression) and programming constructs (e.g. if, if-else, for-loop) (e.g. [10], [18], [22]). These low-level KCs are often associated with syntactical elements in a programming language, and some works (e.g. [11], [19]) used automated methods (namely JavaParser [24]) to extract them from student submitted code snapshots and sample answers in open-ended exercises.

Some works also considered combined usage of a few low-level KCs as a mid-level KC. In an early work on generating completion exercises that focus on student weakness, Chang et al. [15] first broke down code solution to a simple programming problem as a hierarchy of code templates, each of these involving a few lines of code with specific functionality. One example is a template to calculate the average from the sum and the count. Their system may then generate a completion exercise that requires students to fill in a blank line for the average calculation. This work then traced the templates that students cannot answer correctly. Some other works defined mid-level KCs as structured combination of basic programming constructs, and traced the student knowledge of both low-level and mid-level KCs. For example, Chrysafiadi et al. [16] [17] used both basic constructs (e.g. 'if', 'assignment', 'while') and common code patterns (e.g. count in a while loop). In their work in improving multi-skill knowledge tracing, Huang et al. [13] modelled integration of basic skills (e.g. addition, array access, 'for' iteration) to mid-level skills (e.g. using a 'for' to calculate sum of elements in an array).

B. Domain knowledge model

There are two broad categories of student modelling in the systematic sample. Works in the first category built a domain knowledge model that captures the relationships between KCs, and exploits these relationships in evaluating student knowledge levels. Works in the second category, on the other hand, did not model the relationships between KCs, and just traced the knowledge acquisition of individual KC separately. In this section, we focus on the first category, and describe some approaches taken in the literature to overlay the student model on top of the domain knowledge model for programming.

A domain knowledge model organizes the KCs of a domain (e.g. introductory programming) in a certain topology that represents the relationships between the KCs. Examples of common relationships are 'is-part-of' and 'pre-requisite'. The student model is then realized by assigning a knowledge level for each KC in the domain knowledge model. The relationships between KCs help to propagate updates of knowledge levels of KCs throughout the student model every time a student takes an exercise and new performance measures are reported.

In [16], Chrysafiadi et al. proposed to capture the *impact strength* of a KC on another. In their work, each KC has a numerical knowledge level. When knowledge level of KC_i changes because of updates in performance measure on a related exercise, the knowledge level of another KC_j is updated in proportion to the impact strength between the two KCs. The updates can be positive or negative, and bidirectional links may take different impact strength. In a later work [17], the authors formalized the knowledge model as a Fuzzy Cognitive Map of KCs, and defined fuzzy logic rules to update the knowledge levels in the student model.

In their work on visualizing the knowledge levels of students studying interactive multimedia textbooks, Dragon et al. [20] utilized the intrinsic hierarchy structure of topics in textbooks as the domain knowledge model. The KCs are related by 'is-part-of', and KCs are organized in a directed acyclic graph, with concrete programming concepts (e.g. boolean, string) near the bottom, and programming constructs that depend on the concrete concepts (e.g. boolean expression, if statement) on a higher level. The knowledge level of a higher-level KC is aggregated as a weighted average of the knowledge levels of lower-level KCs linked by 'is-part-of'.

Huang et al. built a Bayesian network based model called CKM-HI [13] to represent the dependencies between basic skills, integrated skills and student performance in exercises. The mastery of each KC (basic or integrated skill) is indicated by probability. The model is trained to learn the conditional

TABLE I
SYSTEMATIC SAMPLE ON KNOWLEDGE TRACING FOR LEARNING PROGRAMMING

	Knowledge components (KC)	Student model	Performance measure	Application
Chang et al. 2000 [15]	templates in predefined solutions	set of mistaken templates	closed-ended exercise	generate completion exercises that focus on student weakness
Kasurinen et al. 2009 [10] †	selected constructs	set of KCs, BKT	check whether submitted code applies KCs that are recommended in sample answer	estimate student knowledge level
Chrysafiadi et al. 2013 [16]	selected concepts and constructs	weighted graph of KCs	closed-ended exercise	adapted sequencing of learning material
Chrysafiadi et al. 2015 [17]	selected concepts and constructs	Fuzzy Cognitive Map of KC	closed-ended exercise	adapted sequencing of learning material
Rivers et al. 2016 [11] †	node types in AST	set of KCs used in sample answer, AFM	compare submitted code with sample answer for correct usage of KCs	identity difficult programming elements
Wang et al. 2017 [12] †	implicit KCs in program embeddings	sequence prediction based on embeddings	correct output in next exercise	predict success in next exercise
Wang et al. 2017 [18]	selected concepts and constructs	set of KCs, IRT-BKT	closed-ended exercise	estimate learning outcome, evaluated by prediction accuracy
Huang et al. 2017 [13] †	selected basic and integration skills	Bayesian network of KCs	closed-ended exercise	improve multi-skill knowledge tracing
Hosseini et al. 2017 [19]	concrete concepts from JavaParser	set of KCs, PFA	a snapshot passes test cases	evaluate clustering methods for student stereotypes
Dragon et al. 2017 [20]	predefined concrete and abstract concepts	hierarchy of KCs, is-part-of	closed-ended exercise	visualize the knowledge level of students
Meliana et al. 2018 [21]	selected abstract concepts	Bayesian network of KCs	closed-ended exercise	sequencing of exercises to mimic good students
Effenburger et al. 2020 [22]	predefined game concepts and constructs	set of KCs, AFM	correct output of program	study issues that affect accuracy of AFM

† seed articles

probability between basic skills and an integrated skill, and also the conditional probability between skills and correct response in closed-ended program comprehension exercises. Afterwards, the model can estimate the knowledge level of each KC and predict how likely a student can answer an exercise correctly based on his/her performance in past exercises. Note: CKM-HI applies the dynamic Bayesian network roll-up mechanism in [25] and does not model the temporal transition probability of KC mastery. It updates the student model in discrete time steps, different from the Bayesian network based works discussed in section IV-D.

C. Performance measure

As defined by the scope of this review, all works in the systematic sample estimate student knowledge level from their performance in exercises. A majority of the works use closed-ended exercises (e.g. multiple choice, fill-in-the-blanks) to evaluate student knowledge in different KCs, and rely on expert input to map exercises to related KCs.

A few other works, on the other hand, assess the student knowledge by analyzing code snapshots submitted for programming problems. These works collected multiple code snapshots from students when they attempted to solve programming problems, and determined the correctness of the snapshots by comparing the snapshots with sample answers or running automated test cases on the snapshots. For example, Karurinen et al. [10] examined Python code snapshots for whether they apply KCs recommended by experts. Rivers et al. [11] compared submitted Python code with a sample answer to look for correct usage of expert selected constructs. In [19], Hosseini et al. extracted Java constructs from a parser, and determined that the KC is used correctly when the code snapshot passes some test cases. Efffenberger et al. [22] studied a grid game programming environment called RoboMission (en.robomise.cz), and regarded the KCs as correctly applied when the code snapshot generates a correct output.

D. Knowledge tracing algorithms

Contrary to the few works discussed in section IV-B, many works in the systematic sample did not integrate a domain knowledge model in student modelling. Instead, they regarded the KCs as independent and modelled the change of probability of KC mastery along a sequence of related exercises. This approach is called knowledge tracing [26], and builds a temporal model that can predict the correctness of student response in an exercise based on their performance in all previous attempted exercises. Three kinds of techniques are encountered in the literature.

1) Bayesian Knowledge Tracing (BKT): BKT is a commonly used probabilistic method in student modelling, and has been employed in [10], [18] and [21] among the systematic sample. BKT models the learning process of a KC using a Hidden Markov Model (HMM) with two latent states: either the student has learned a KC ('known') or not ('not known'). Initially, a student may know the KC already, with the probability P_{initial}. After each exercise that involves the KC, there is a probability P_{learn} for the student to transit from the 'not known' state to 'known'. However, the two latent states cannot be observed directly, and can only be estimated based on observable performance in the exercises. To accommodate the inference uncertainty between student performance and their actual understanding, BKT model includes the probability P_{guess} that a student does not know a skill, but gets a correct response by chance, and the probability P_{slip} for careless mistakes a student may make, even when he/she knows the skill.

Kasurinen et al. [10] selected programming constructs that are difficult to students (e.g. 'while', 'exception' and 'file') and applied BKT to reveal that about half of the students have learned the constructs. On the other hand, Meliana et al. [21] traced the student learning of the three abstract concepts 'sequence', 'conditional' and 'loop' based on a sequence of exercises in their learning material, and attempted to distill the sequence of exercises taken by good students. In [18], the authors tried to improve prediction accuracy by first using Item Response Theory (IRT) on the student performance data to estimate the initial capability P_{initial} of students in 13 preselected programming concepts and constructs (e.g. array and iteration). They then applied BKT to learn other parameters, and found that their combined IRT-BKT approach outperforms BKT alone.

A limitation of the basic formulation of BKT, as employed in the works in the systematic sample, is that each exercise is restricted to assess a single skill. Typically, BKT builds a separate HMM for each skill. There are extensions of BKT to accommodate exercises that assess multiple skills, e.g. Weakest Knowledge Tracing [27] [28] breaks up the responsibility of success or failure in a multi-skill exercise among several HMMs, and Conjunctive Knowledge Model [25] [29] builds a single Dynamic Bayesian Network (DBN) [30] to model the learning progression of multiple skills. In fact, one work in the systematic sample (Huang et al. [13]) extended DBN with skill integration. Nonetheless, training a DBN for multiple skills and multi-skill exercises has proved to be difficult because of explosion of parameters to learn.

2) Additive Factor Model (AFM): Because of this limitation of BKT, a group of works [11], [19] and [22] in the systematic sample applied another popular approach in knowledge tracing called Additive Factor Model to trace programming knowledge. Instead of closed-ended exercises, these works collected code snapshots submitted by students when they try to solve programming problems. Each snapshot is analyzed to infer the learning progress of a set of skills that are essential in a solution to the programming problem. In some works, each submitted snapshot for a problem is considered as an *opportunity* to test the understanding of the related KCs.

AFM adapts logistic regression, a common statistical method, to knowledge tracing. This model directly supports multi-skill exercises with a boolean Q-matrix Q_{kj}, in which $Q_{kj} = 1$ if the opportunity j requires the skill k. A student is assumed to have a higher probability to learn a skill k after repeated practice of the skill, and the number of practice before this opportunity is stored in N_{ik}. The AFM model includes several parameters to characterize the learning progress of students: θ_i refers to different initial capability of students, β_k refers to difficulty of a KC k, and γ_k indicates how quickly students can learn the KC from doing multiple relevant practice. Given the observed performance p_{ij} of whether a student i answers correctly in the opportunity j, AFM uses the logistic regression equation in (1) to learn the parameters. Afterwards, one can predict the probability that a student can answer a future exercise that requires some of the skills.

$$\ln \frac{p_{ij}}{1 - p_{ij}} = \theta_i + \sum_k \beta_k Q_{kj} + \sum_k (\gamma_k N_{ik}) Q_{kj} \qquad (1)$$

Rivers et al. [11] used the first time that a KC appears in a snapshot (as returned by JavaParser) as an opportunity to assess knowledge of the KC. If that KC appears in an appropriate position when compared to a model answer, the student is considered to have given a correct response. In their work, knowledge tracing with AFM reveals how well students are learning difficult programming elements.

Hosseini et al. [19] studied several methods to group students into stereotypes using demographics, course performance and learning behavior. Among other metrics, they used an extension of AFM called Performance Factor Analysis [31] to verify whether students within a stereotype have similar learning needs. (Performance Factor Analysis differs from AFM in having separate parameters for learning from correct and incorrect responses in exercises.) Each time that a student tests, runs or submits a snapshot is considered an opportunity to evaluate the knowledge of KCs in the snapshot, which are extracted automatically by JavaParser. If the snapshot passes all test cases, the student is assumed to have given correct responses for all the KC discovered.

AFM assumes that repeated practice of a skill results in the same amount of improvement for a student. However,

this may not be true when the exercises have vastly different difficulties. This has been verified by Effenberger et al. [22]. The authors applied AFM on their RoboMission dataset to assess learning of game concepts and programing constructs (e.g. while, repeat, if, else, nested loop). They concluded that differences in exercise difficulty (which is not considered in AFM) may affect the validity of the conclusion drawn from AFM.

3) Deep Knowledge Tracing (DKT): By treating knowledge tracing as a sequence prediction problem, Piech et al. [32] proposed Deep Knowledge Tracing and its two formulations (exercise-indexed and skill-indexed). Skill-indexed DKT employs Long Short-Term Memory (LSTM) recurrent neural networks [33] to predict whether a student can answer an exercise correctly based on the input sequence of the KCs covered and the correctness of answers in previous attempted exercises. As such, DKT builds a single model for all KCs, whereas BKT builds separate model for each KC.

Wang et al. [12] proposed a novel approach to adapt DKT for programming in their experiment on the Hour of Code dataset (code.org/research). Instead of tagging selected or automatically extracted KCs to exercises, the authors used a recursive neural network to learn a meaningful embedding of a submitted program snapshot. The embedding is calculated as a linear combination of the embeddings of child nodes in the Abstract Syntax Tree (AST), and encodes the runtime function of the program in transforming various inputs into corresponding outputs. The authors then trained a recurrent neural network, taking the sequence of embeddings for snapshots submitted for an exercise as input, and whether the student can produce a correct solution in the *next* exercise as output. Although no explicit KCs were used in training the recurrent neural network, the input embeddings can be regarded as *implicit representation* of the syntax (due to the recursive encoding of AST nodes) and semantic (due to the encoding of runtime function) of the snapshots. However, it is arguable that DKT suffers from interpretability problem, especially given the complicated syntactic and semantic content of program snapshots.

V. Discussion

Programming is a creative process, and students have to learn a variety of low-level and high-level skills to gain proficiency in coding solutions for programming problems. Some skill is a pre-requisite to others, while some integrated skill is more than the sum of its components.

One challenge in tracing programming knowledge is proper representation of programming knowledge components and their intricate relationship. Many existing works used basic programming concepts and constructs as targets for knowledge tracing, but did not model the relationship between the KCs. However, it is a common struggle of students to combine the basic KCs to build a solution. In works that feature a domain knowledge model, relationship between basic skills and some integrated skills are fused into the knowledge level update algorithm. However, expert inputs are required in these works to define the skills and assign the weights of the relationship [16] [17]. Moreover, the effectiveness of the domain knowledge model is often limited by the structure of the learning material from which the model is derived [20].

Another challenge is the syntactic and semantic richness of an important evidence for learning programming – program codes. Most existing works use closed-ended exercises to measure student performance, and many among them assign a single KC to each exercise. Such exercises are insufficient to evaluate higher-level programming skills required in designing a complete program. There are a few works that analyze submitted code snapshots for manifestation of KCs, usually via automated test cases or comparison with model answers. Although expert inputs are not required, a common restriction is that only low-level KCs are recognized.

Two works that open up new directions for tracing of more sophisticated programming knowledge are Huang et al. [13], which investigated Bayesian networks to model expert-curated skill integration, and Wang et al. [12], which pioneered the deep knowledge tracing approach to predict future student performance from meaningful embeddings of program snapshots submitted earlier.

We believe that a possibly fruitful approach is in the middle. Programming schemas [34] (also known as plans [35] and patterns [36]) are prototypical solutions to small-scale problems. (An example of schemas is using 'for' loop to calculate the sum of numbers in an array.) There have been some trail of schemas in some reviewed works [15] [17] [13]. Programming schemas can be extracted from program codes automatically, and a student who combines several schemas to build a solution is manifesting an integrated skill. Moreover, schemas are easier to interpret for students and teachers, and program embeddings based on schemas may strike a balance between interpretability and the syntax and semantic richness.

VI. Conclusion

In this review, we have characterized the distinct challenges in knowledge tracing of programming, and the different approaches existing works used to adapt established techniques of student modelling and knowledge tracing. A lot of research is still required to trace higher-level programming knowledge of students. As programming has become an essential skill to solve real-world engineering problems, and more students (including formal and vocational education) are learning how to program, tracing individual's acquisition of programming knowledge has the potential to pave the way for more effective and enjoyable programming education.

References

[1] Tony Lowe. Explaining novice programmer's struggles, in two parts: Revisiting the ITiCSE 2004 working group's study using dual process theory. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, page 30–36, New York, NY, USA, 2019. Association for Computing Machinery.

[2] Anabela Gomes and Antonio Mendes. A teacher's view about introductory programming teaching and learning: Difficulties, strategies and motivations. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, pages 1–8, 2014.

[3] Ilias O. Pappas, Michail N. Giannakos, and Letizia Jaccheri. Investigating factors influencing students' intention to dropout computer science studies. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, page 198–203, New York, NY, USA, 2016. Association for Computing Machinery.

[4] Kenneth R Koedinger, Albert T Corbett, and Charles Perfetti. The knowledge-learning-instruction framework: Bridging the science-practice chasm to enhance robust student learning. *Cognitive science*, 36(5):757–798, 2012.

[5] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. Intelligent tutoring systems for programming education: A systematic review. In *Proceedings of the 20th Australasian Computing Education Conference*, ACE '18, page 53–62, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Konstantina Chrysafiadi and Maria Virvou. Student modeling approaches: A literature review for the last decade. *Expert Systems with Applications*, 40(11):4715–4729, 2013.

[7] Abir Abyaa, Khalidi I. Mohammed, and Samir Bennani. Learner modelling: systematic review of the literature from the last 5 years. *Educational Technology, Research and Development*, 67(5):1105–1143, 10 2019. Copyright - Educational Technology Research and Development is a copyright of Springer, (2019). All Rights Reserved; Last updated - 2019-09-11.

[8] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. Educational data mining and learning analytics in programming: Literature review and case studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports*, ITICSE-WGR '15, page 41–63, New York, NY, USA, 2015. Association for Computing Machinery.

[9] Arto Hellas, Petri Ihantola, Andrew Petersen, Vangel V. Ajanovski, Mirela Gutica, Timo Hynninen, Antti Knutas, Juho Leinonen, Chris Messom, and Soohyun Nam Liao. Predicting academic performance: A systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE 2018 Companion, page 175–199, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Jussi Kasurinen and Uolevi Nikula. Estimating programming knowledge with Bayesian knowledge tracing. In *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '09, pages 313–317, New York, NY, USA, July 2009. Association for Computing Machinery.

[11] Kelly Rivers, Erik Harpstead, and Ken Koedinger. Learning Curve Analysis for Programming: Which Concepts do Students Struggle With? In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16, pages 143–151, New York, NY, USA, August 2016. Association for Computing Machinery.

[12] Lisa Wang, Angela Sy, Larry Liu, and Chris Piech. Deep Knowledge Tracing On Programming Exercises. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S '17, pages 201–204, New York, NY, USA, April 2017. Association for Computing Machinery.

[13] Yun Huang, Julio Guerra-Hollstein, Jordan Barria-Pineda, and Peter Brusilovsky. Learner Modeling for Integration Skills. In *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization*, UMAP '17, pages 85–93, New York, NY, USA, July 2017. Association for Computing Machinery.

[14] Adam S. Carter, Christopher D. Hundhausen, and Olusola Adesope. The normalized programming state model: Predicting student performance in computing courses based on programming behavior. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, page 141–150, New York, NY, USA, 2015. Association for Computing Machinery.

[15] Kuo-En Chang, Bea-Chu Chiao, Sei-Wang Chen, and Rong-Shue Hsiao. A programming learning system for beginners – a completion strategy approach. *IEEE Transactions on Education*, 43(2):211–220, May 2000.

[16] Konstantina Chrysafiadi and Maria Virvou. Dynamically Personalized E-Training in Computer Programming and the Language C. *IEEE Transactions on Education*, 56(4):385–392, November 2013.

[17] Konstantina Chrysafiadi and Maria Virvou. Fuzzy Logic for Adaptive Instruction in an E-learning Environment for Computer Programming. *IEEE Transactions on Fuzzy Systems*, 23(1):164–177, February 2015.

[18] Shanshan Wang, Yong Han, Wenjun Wu, and Zhenghui Hu. Modeling student learning outcomes in studying programming language course. In *2017 Seventh International Conference on Information Science and Technology (ICIST)*, pages 263–270, April 2017.

[19] Roya Hosseini, Peter Brusilovsky, Michael Yudelson, and Arto Hellas. Stereotype Modeling for Problem-Solving Performance Predictions in MOOCs and Traditional Courses. In *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization*, UMAP '17, pages 76–84, New York, NY, USA, July 2017. Association for Computing Machinery.

[20] Toby Dragon and Carrie Lindeman. Automated Assessment of Students' Conceptual Understanding: Supporting Students and Teachers Using Data from an Interactive Textbook. In *2017 IEEE International Symposium on Multimedia (ISM)*, pages 567–572, December 2017.

[21] Selly Meliana and Dade Nurjanah. Adopting Good-Learners' Paths in an Intelligent Tutoring System. In *2018 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*, pages 877–882, December 2018. ISSN: 2470-6698.

[22] Tomáš Effenberger, Radek Pelánek, and Jaroslav Čechák. Exploration of the robustness and generalizability of the additive factors model. In *Proceedings of the Tenth International Conference on Learning Analytics & Knowledge*, LAK '20, page 472–479, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, and Renske Weeda. Fostering program comprehension in novice programmers - learning activities and learning trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '19, page 27–52, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Roya Hosseini and Peter Brusilovsky. JavaParser: A fine-grain concept indexing tool for Java problems. In *CEUR Workshop Proceedings*, volume 1009, pages 60–63. University of Pittsburgh, 2013.

[25] Cristina Conati, Abigail Gertner, and Kurt Vanlehn. Using Bayesian networks to manage uncertainty in student modeling. *User modeling and user-adapted interaction*, 12(4):371–417, 2002.

[26] Radek Pelánek. Bayesian knowledge tracing, logistic models, and beyond: an overview of learner modeling techniques. *User Modeling and User-Adapted Interaction*, 27(3-5):313–350, 2017.

[27] Yue Gong, Joseph E Beck, and Neil T Heffernan. Comparing knowledge tracing and performance factor analysis by using multiple model fitting procedures. In *International conference on intelligent tutoring systems*, pages 35–44. Springer, 2010.

[28] José González-Brenes, Yun Huang, and Peter Brusilovsky. General features in knowledge tracing to model multiple subskills, temporal item response theory, and expert knowledge. In *The 7th international conference on educational data mining*, pages 84–91. University of Pittsburgh, 2014.

[29] Michael Mayo and Antonija Mitrovic. Optimising its behaviour with bayesian networks and decision theory. 2001.

[30] Paul Dagum, Adam Galper, Eric Horvitz, and Adam Seiver. Uncertain reasoning and forecasting. *International Journal of Forecasting*, 11(1):73–87, 1995.

[31] Jill-Jênn Vie and Hisashi Kashima. Knowledge tracing machines: Factorization machines for knowledge tracing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 750–757, 2019.

[32] Chris Piech, Jonathan Bassen, Jonathan Huang, Surya Ganguli, Mehran Sahami, Leonidas J Guibas, and Jascha Sohl-Dickstein. Deep knowledge tracing. In *Advances in neural information processing systems*, pages 505–513, 2015.

[33] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[34] Robert S Rist. Learning to program: schema creation, application and evaluation. *Computer Science Education and Research*, pages 175–197, 2004.

[35] Elliot Soloway. Learning to program = learning to construct mechanisms and explanations. *Commun. ACM*, 29(9):850–858, September 1986.

[36] Orna Muller. Pattern oriented instruction and the enhancement of analogical reasoning. In *Proceedings of the First International Workshop on Computing Education Research*, ICER '05, page 57–67, New York, NY, USA, 2005. Association for Computing Machinery.