www.editada.org

International Journal of Combinatorial Optimization Problems and Informatics, 17(1), Jan-April 2026, 307-317. ISSN: 2007-1558. https://doi.org/10.61467/2007.1558.2026.v17i1.1170

Recommending Computational Thinking Learning Paths using Fuzzy Logic

Joel Tapia Flores, María Susana Ávila García, Misael López Ramírez
Universidad de Guanajuato, Mexico.
j.tapiaflores@ugto.mx, Susana.avila@ugto.mx, lopez.misael@ugto.mx

Abstract. Nowadays, computational thinking plays an important role in everyday life, which has led schools to incorporate programming into their curricula in order to support its development. In this research, the focus is placed on the generation of learning paths for children. To this end, a tool was developed to capture user data, which were subsequently processed and analysed using fuzzy logic. As a result, the system was able to identify the next exercise that each user should solve, thereby supporting the construction of personalised learning paths for individual users within the platform.
Keywords: Computational Thinking, Programming, Learning Paths

Article Info
Received September 23, 2025
Accepted July 10, 2025

1 Introduction

According to Guzdial (2008), computer science education will pave the way for making computational thinking a 21st-century literacy accessible to all. This form of analytical thinking, as argued by Wing (2008; M. Wing, 2006), integrates key elements from diverse disciplines: it shares with mathematical thinking the use of concepts such as abstraction and decomposition to solve problems; incorporates engineering approaches for system design, and adopts scientific methodologies to understand human intelligence and cognition.

Furthermore, since programming is a key tool for developing Computational Thinking (M. Wing, 2006), European countries (e.g., France and Spain) have integrated programming into their curricula (Balanskat & Engelhardt, 2015), as highlighted by Guzdial (2008).

However, key questions emerged, such as: What types of tools are effective for developing computational thinking? How can classes be designed to make programming accessible to students with diverse backgrounds? (Guzdial, 2008). These questions spurred pedagogical research into strategies leveraging tools like Scratch (Meerbaum-Salant et al., 2013) and CODE (Kalelioğlu, 2015) to foster computational thinking in early education.

Nevertheless, this has not prevented learning personalization from remaining a challenge. To address this, Saito and Watanobe (2020) proposed a learning path recommendation system that suggests one or more problems to solve to achieve a target score. Their approach considered three key factors: (1) the user's action history in Online Judge (OJ) systems, (2) their specific goals, and (3) their progress along the learning path.

To achieve this, 2,000 problems were categorized into two taxonomies: "what: problem classification" and "how: algorithm classification." The recommendation process involved data manipulation, clustering, training a Long Short-Term Memory (LSTM) network, and generating recommendations. As a result, the system proposed specific problems to solve from the "what" and "how" categories based on the user's last five submitted exercises.

However, while Saito and Watanobe's (2020) approach focuses on exercise personalization, it faces another challenge: the inherent difficulty of text-based programming languages for children. This limitation has driven the adoption of alternative methods, such as visual programming (Repenning, 1993), which leverages natural cognitive abilities to create more intuitive learning experiences.

A notable example is the Kodetu platform developed by Guenaga et al. (2021), which analyzed interactions from over 1,000 students (ages 8-14) to generate indicators of Computational Thinking acquisition. The results revealed that factors like age and

gender significantly influence learning outcomes, while also providing: (1) design recommendations for more effective platforms, and (2) evidence supporting the value of interaction data for assessing cognitive processes.

Grover et al. (2016) employed a mixed-methods approach (K-12, qualitative-quantitative) to analyze: (1) abstract syntax trees (ASTs), (2) user interactions, and (3) screenshots from participants aged 11 to 50. Their results demonstrate that problem-solving time predicts skill level with 63% accuracy, while time between executions shows 55% predictive accuracy. Through statistical analysis, they found a positive correlation between the use of control structures and loops and programmer experience.

Grover et al. (2016) developed a scalable method for assessing students' problem-solving processes in online learning environments. Their approach incorporated user solutions and actions, student characteristics, timing data, and abstract syntax trees (ASTs), subsequently extracting key features to train and evaluate their machine-learning model. The results demonstrated that their model could predict student problem-solving performance with 85% accuracy.

Robles et al. (2017) examined the prevalence of code duplication (clones) in Scratch projects and its relationship with students' computational thinking. Their methodology involved collecting data on cloned code characteristics, problem difficulty, student age and experience, and computational thinking skill levels. These variables were analyzed using descriptive statistics and linear regression. The results revealed a high frequency of code cloning, which increased with exercise complexity but showed no direct correlation with users' computational thinking levels.

While these studies have identified key factors influencing computational thinking development (including age, gender, user interactions, and use of complex structures), the definition of personalized learning paths remains underexplored. Our research addresses this gap by employing fuzzy logic to generate customized learning trajectories. This predictive approach analyzes performance on completed exercises and, based on these results, automatically determines the next problem to solve—thereby creating adaptive pathways tailored to each student.

2 State of the Art

Recent studies have explored the generation of personalized learning paths, such as the work by Jiang et al. (2020), which focused on predicting the outcome of the next exercise to be solved. This was achieved by creating a new measurement metric called the Approximation Index (AI), where a result of 1 indicates correctness and 0 indicates an incorrect solution, which calculates the difference between the optimal solution's Abstract Syntax Tree (AST) and the ASTs of intermediate solutions generated by participants. For this study, the researchers used the Hour of Code datasets (HOC4 and HOC18) from the CODE platform, along with the features pathLength (which stores the number of intermediate solutions) and currentPerf (which records whether the solution was correct or not). These were processed using both a linear regression model and a Recurrent Neural Network (RNN). The results enabled the identification of users likely to fail the next exercise. While this research proposes a novel method for knowledge tracing and predicting users at risk of failing subsequent exercises, it fails to provide a concrete learning path or recommended exercise to help users continue developing their programming and computational thinking skills.

On the other hand, Moreno-León et al. (2020) analyzed 250 projects from the Scratch platform (50 projects from each of the 5 main categories: animation, art, games, music, and stories) to create a learning path that would allow students to gradually develop their computational thinking through progressively challenging exercises. For this purpose, they employed the Dr. Scratch tool, which evaluates projects on the platform according to the 7 dimensions of CT, while also implementing statistical and cluster analyses. These analyses yielded a learning path consisting of three groups: Group 1 (comprising animation, art, and music projects), Group 2 (stories), and Group 3 (games). While this research generates a learning path within Scratch, it only provides the sequence of categories to follow without specifying the number of projects to complete in each category, resulting in an overly generic learning path.

In this article, we propose a personalized learning path system based on fuzzy logic. This system uses the following input variables: completion time, number of attempts, and errors made in each exercise. As output, the model determines the next exercise to be solved, adapting to each student's pace and skill level. Through this approach, we aim to promote not only programming skills acquisition but also the development of computational thinking. In the following sections, we detail: (1) the fuzzy method generation, (2) practical implementation, and (3) experimental results obtained.

The following table presents a comparison that summarizes the key aspects of previous works alongside our research

Table 1 Comparative Table Summarizing the Key Aspects of Previous Works and Our Own

Research	Platform	Learning Path Generation Method	Modeling Techniques	Level of Personalization Achieved
Jiang et al. (2020)	CODE	Identification of users who would fail to complete the next exercise	Linear Regression Model and RNN	Low level of personalization, since it only predicts which users would fail the next exercise and does not generate a learning path to support user learning
Moreno-León et al. (2020)	Scratch	Learning path formed by three clusters	Statistical analysis and clustering	Medium level of personalization, because it shows the order to follow for the categories, but does not indicate the number of projects to be completed in each one, thus leaving an incomplete learning path
Our research	Own platform	Personalized learning path system	Fuzzy Logic	High level of personalization, as it generates concrete and adaptive learning paths based on performance indicators that help users develop computational thinking and programming concepts

3 Methodology and Tools

Figure 1 illustrates the methodology employed in our research.

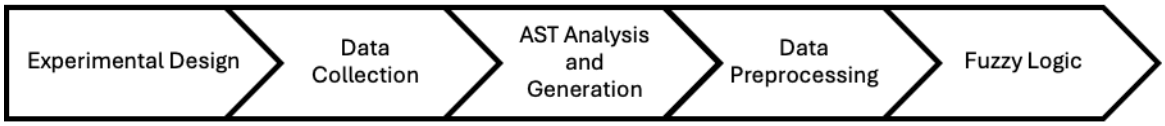

Fig. 1. Methodology used.

3.1 Experimental Design

A. Participants

For our experiment, we selected children aged 8 to 10 years old who were enrolled in the 4th grade of primary school and had no prior experience with programming tools.

A workstation was installed consisting of a table, laptop, and chair for participants. Children participated individually to complete the programming exercises.

Ethical Considerations

Permission forms were signed by participants' guardians, including Participation agreements, Data collection consent forms, these documents specified: The permitted uses of collected data and the right to withdraw from the experiment at any time.

B. Blockly-based Programming Tool

For the development of our tool, we utilized the PHP and JavaScript programming languages along with the Blockly library. Blockly provides a toolbox featuring visual programming blocks as instructional components. Figure 2 displays the main interface of our implemented tool.

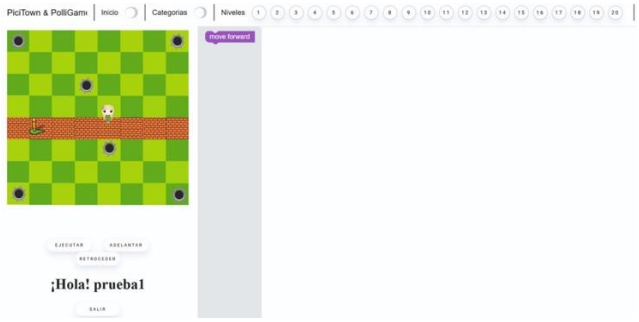

Fig. 2. Main interface of the programming tool.

C. Blockly-based Programming Tool

In Image 3, we present the 20 programming exercises, which are responsible for collecting user data and were designed by us, while in table 1 we list the type of blocks provided to participants to complete the activity. Although the images share two exercises, both the blocks and their solutions are different. For example, examples 1 and 11 share the same activity, however, for exercise 1 only 'move' motion blocks are offered, and for exercise 11 'move' motion blocks and 'repeat_until' repetition blocks are provided.

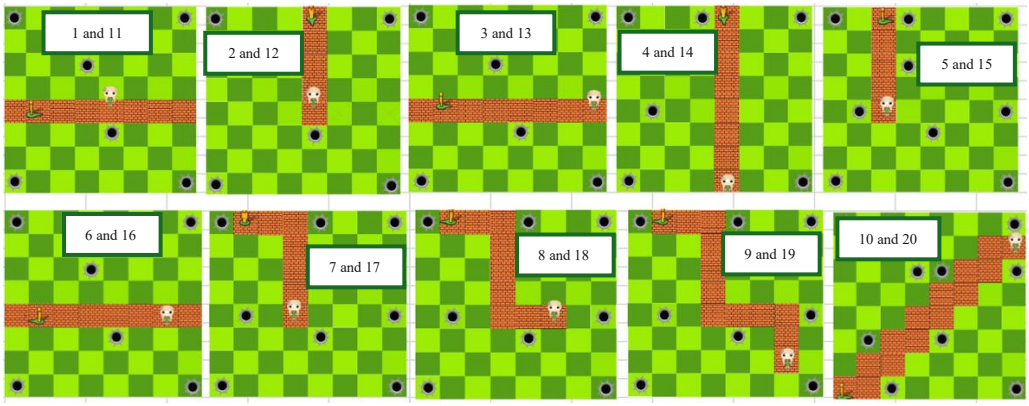

Fig. 3. Exercise portfolio.

Table 1. Blocks available for solving programming exercises

Toolbox	Toolbox blocks
1 To 4	Move
5 To 10	Move and Turn
11 To 20	Move, Turn and Repeat_Until

4 Data Collection

The data collected by our platform consists of: The execution time of solutions and the exercise solutions. These solutions are categorized into three types: A) Optimal Solution (SO): Solutions generated with the minimum number of blocks (solutions created by our team); B) Intermediate Solution: Solutions generated by users before reaching a final solution; C) Final Solution: The user's ultimate solution before moving to the next exercise, which may or may not be correct and optimal.

These solutions are stored in XML files to later be converted into Abstract Syntax Trees (AST). Illustration 4 shows the data collection process.

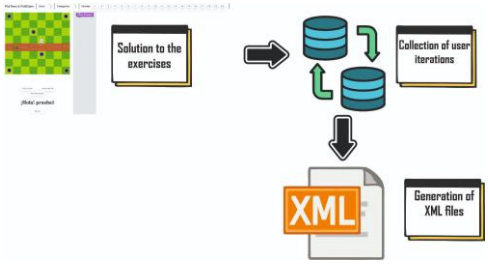

Fig. 4. Data Collection flowchart.

4.1 Generation and analysis of AST

For generating the ASTs, we created a Python program that converts the XML files into JSON-type ASTs. Table 2 shows an example of this conversion.

Table 2. Converting XML to JSON-type AST

XML	AST
<xml xmlns="http://www.w3.org/1999/xhtml"><block type="move" id="b}opX/7(b9=;Vz.];)A+" x="8" y="41"><next><block type="move" id="kf,le($R):m4Ra_{NqO+"><next><block type="move" id="tjoGRHxI,xGv@WFo~~R}"></block></next></block></next></block></xml>	{ "type": "move", "id": "b}opX/7(b9=;Vz.];)A+", "x": "8", "y": "41", "fields": {}, "next": { "type": "move", "id": "kf,le($R):m4Ra_{NqO+", "x": null, "y": null, "fields": {}, "next": { "type": "move", "id": "tjoGRHxI,xGv@WFo~~R}", "x": null, "y": null, "fields": {} } } }

To convert the optimal solutions into ASTs, we first solved the exercises optimally, then transformed them into ASTs. Table 3 shows 3 exercises with their optimal solutions, while Illustration 5 graphically displays this process.

Table 3. Optimal solutions to 3 programming exercises

exercise	Optimal Solutions
1	Move, move, move
5	Turn, move, move, move, move
11	Repeat_until, move

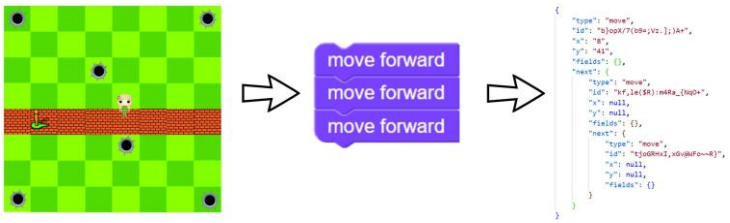

Fig. 5. Process of generating SOs.

We then converted both intermediate and final user solutions into ASTs and developed a Python program to compare them with the optimal ASTs. As a result, we obtained: Whether the solution was correct and optimal or not, the AST height, the error type (when present) the error's position in the tree (when errors existed).

4.2 Data Preprocessing

With the analyzed ASTs, we generated a data dictionary that will help interpret and define the data. Table 4 shows this dictionary.

Table 4. Data dictionary

Field	Meaning
User_ID	Unique user identifier
Attempt	Attempt at the intermediate solution to be reviewed
Problem ID	Problem to solve
Time	Time to solve the exercise
S.O. Difficulty	Difficulty of the problem
S.A. Difficulty	Difficulty made by the user
S.O. Height	Correct height of the problem
S.A. Height	Height made by the user
Success	Result of the exercise (correct/incorrect)
Error type	Error found in the S.I.
Error Height	Position in the AST where the error is located
Total Attempts	Total attempts made to solve the exercise

When generating the data dictionary, we noticed the time was stored in milliseconds, so we converted it to seconds to achieve more standardized values across all fields. We also verified there were no empty fields or duplicate rows in our dataset. Table 5 shows a portion of our database structure after these modifications.

Table 5. Database

User_ID	Attempt	Problem ID	Time	S.O. Difficulty	S.A. Difficulty	S.O. Height	S.A. Height	Success	Error Type	Error Height	Total Attempts
1	1	1	23.212	1	1	3	1	No	Missing blocks	1	2
1	2	1	55.27	1	1	3	4	No	Leftover blocks	3	2
1	1	3	174.26	1	1	6	6	Si	Null	0	1
1	1	4	55.658	1	1	7	6	No	Missing blocks	6	3
1	2	4	133.095	1	1	7	7	Si	Null	0	3

4.3 Feature Extraction

Once the features were defined, we analyzed which of them would be part of our fuzzy method. Table 6 shows these selected features.

Table 6. Features for the fuzzy method

User_ID	Unique user identifier
Time	Time to solve the exercise
S.A. Height	Height made by the user
Total Attempts	Total attempts made to solve the exercise
Error Height	Position in the AST where the error is located

5 Fuzzy Logic

Fuzzy logic is a system for dealing with reasoning that is approximate rather than exact. In its broad sense, fuzzy logic is based on fuzzy set theory. It utilizes concepts, principles, and methods developed within fuzzy set theory to create various forms of approximate reasoning (Lotfi A. Zadeh, 1998; Klir & Yuan, 1995).

For these reasons, we selected fuzzy logic for our experiment, as code analysis and learning path generation are not exact processes; they inherently contain uncertainty when determining whether and how a particular user is learning. Therefore, our expected outcome is to recommend the next exercise for generating user-specific learning paths.

5.1 Fuzzy Logic Implementation

For implementing fuzzy logic, the following 4 steps must be performed:

1. **Define the fuzzy variables:** These are the variables that will feed the fuzzy system. For our system, the variables are Time, Solution, height, Total attempts, Error height.

2. **Define the fuzzy sets:** These are the value ranges for each fuzzy variable. For this purpose, we standardized our variable values within a 0 to 1 range, where 1 represents the optimal value and values further from 1 are less ideal. Illustration 6 shows these fuzzy sets.

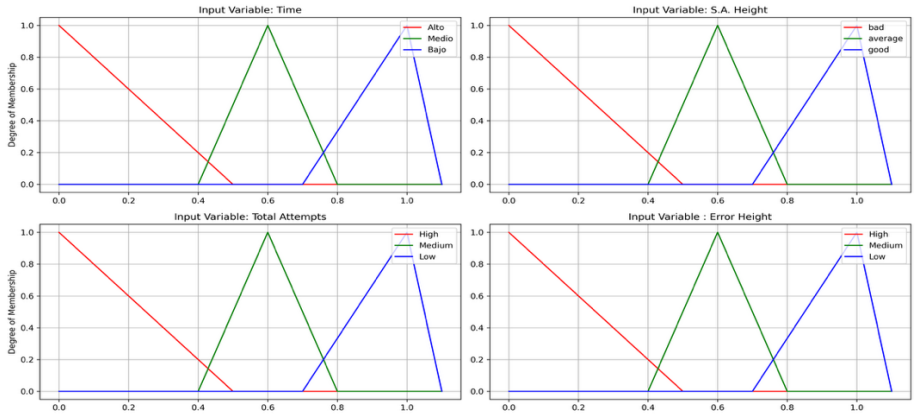

Fig. 6. Values of fuzzy sets

Each input variable has three possible output states: High, Medium, Low for: Time, Total Attempts, and Error Height variables. Bad, Fair, Good for Solution Height (SA) variable. Example: Time variable classification: High time range: 0 to 0.5, Medium time range: 0.4 to 0.8, Low time range: 0.7 to 1.5.

3. **Create the rules**: As the third step, the fuzzy rules are generated. These are created by combining the outputs of our input variables and assigning a result. For example, if we want to make a rule that gives us the result of moving on to the next exercise: the outputs of our variables must have a low time, a good hight, low attempts, and a low error height. This rule would look as follows: ctrl.Rule(time['low'] & heigh['good'] & attemps['low'] & Error high['low'], action['advance']). To ensure that all values produce a result, we create all possible combinations of the outputs of our input variables. Below, we present some of the rules we used for our research.

   - If the time is medium, the height is good, the number of attempts is low, and the error height is low, then move on to the next exercise.

   - If the time is high, the height is bad, the number of attempts is high, and the error height is high, then repeat the exercise.

   - If the time is medium, the height is average, the number of attempts is medium, and the error height is medium, then do a reinforcement exercise.

4. **Membership calculation**: As the final step, the memberships of the variables are calculated. This results in the degree of belonging for each of them. In Illustration 7, the output membership functions are shown. A value less than 1 will result in repeating the exercise, a value greater than 0.5 up to 2 will lead to a reinforcement exercise, and a value greater than 1.5 up to 3 will result in moving on to the next exercise. In the following section, we present the results of our fuzzy system.

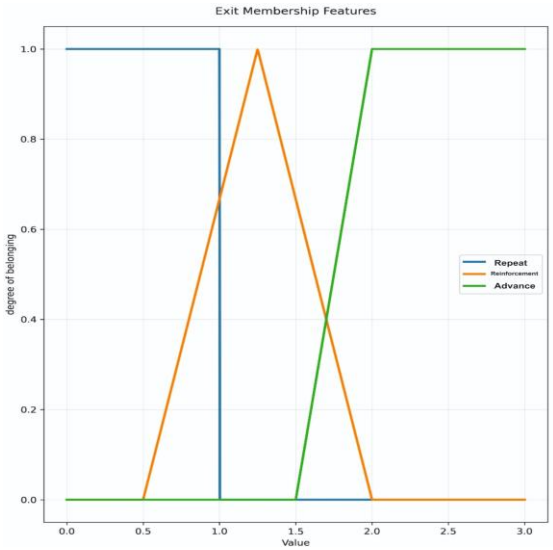

Fig. 7. Output membership functions

6 Results

The results obtained were as follows: Illustration 8 shows the exercises completed by the students on our platform, as well as their results (correct/incorrect) and the number of attempts for each exercise. The graph was created using Python, taking into account the results of the exercises and the number of attempts made by each user.

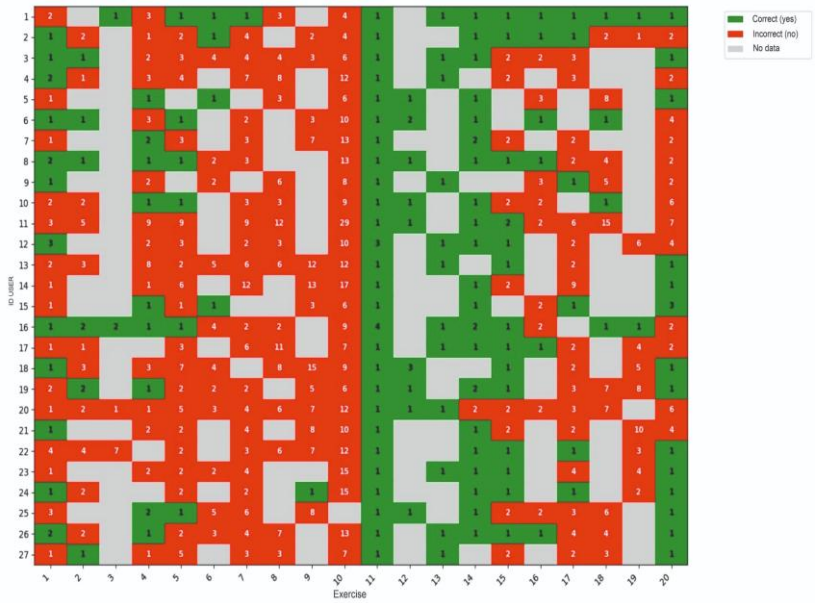

Fig. 8. User participation for problems

Illustration 9 represents the surface generated by our fuzzy system, which models the relationship between the number of attempts, the error, and the recommended action. In our model, the action takes values from 0 to 1, where values closer to 1 recommend moving on to the next exercise, values closer to 0 suggest repeating the exercise, and intermediate values suggest reinforcement exercises.

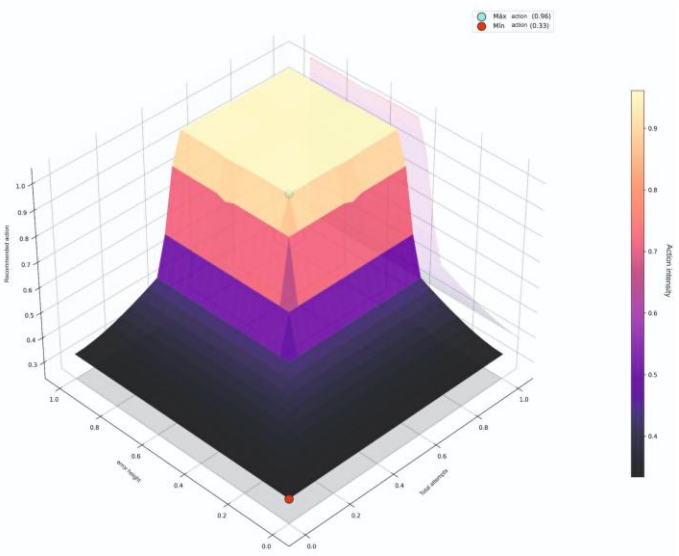

Fig. 9. Attempt-Error-Action Relationship

When the number of attempts is low and the error is also low, the recommended action reaches its maximum value (0.91). This implies that the system considers the student has adequately understood the exercise and can move on. In contrast, when there are more attempts, even if the error is low, the recommended action decreases significantly. This indicates that although the student eventually solved the exercise, multiple attempts were needed, suggesting they have not yet fully mastered the concept and should therefore complete a reinforcement exercise. The darkest region of the graph, with an action value close to 0.0, corresponds to situations with many attempts and high errors, where the system clearly decides not to proceed and recommends repeating the exercise.

In Illustration 10, we present the results of our fuzzy model, which generates personalized learning paths for each of our users. For example, for user 27 in exercise 1, our fuzzy system recommends completing reinforcement exercise 1; for exercise 2, it suggests moving on to exercise 4; and for exercise 4, it recommends completing reinforcement exercises 3 and 4, and so on for the remaining exercises.

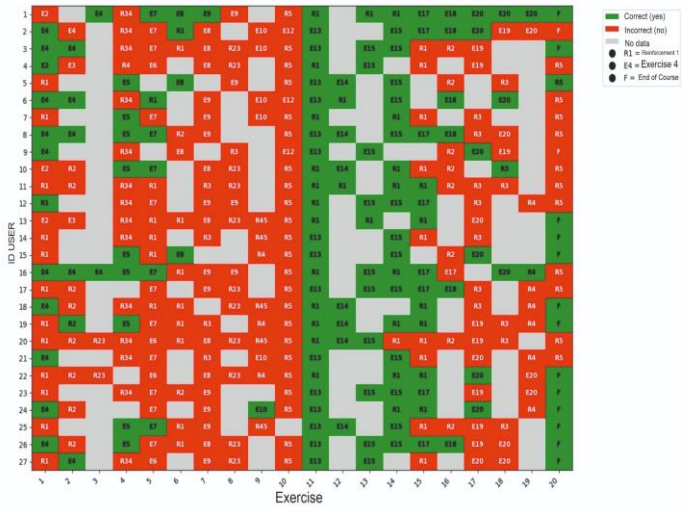

Fig. 10. Learning Path for user 1

These results are consistent with the objective of the system, which seeks to adapt to each student's pace, prioritizing meaningful learning before allowing them to progress to more complex exercises. This ensures that each user gradually develops their computational thinking as they complete and advance through their programming exercises.

7 Conclusions

Programming fosters the development of computational thinking in children, which is why many countries around the world have chosen to include it in their educational curricula. However, questions arise regarding the most effective ways to teach it. As a result, several researchers have used child-friendly programming platforms, such as Scratch and Code, to conduct their studies. These works have produced outcomes such as predictions about the next exercise or the recommended categories in Scratch to start learning programming. However, none of them generate learning paths that adapt to the students' knowledge and learning pace.

For this reason, we developed a tool that allowed us to capture data, which was then processed and analyzed within a fuzzy system. This system takes as input the characteristics of the exercises completed by users and outputs the next exercise to be solved. With these results, we generated personalized learning paths that enable our users to progress and develop their computational thinking at their own pace and learning level.

Our goal is to offer a new approach for developing computational thinking and teaching programming to children—one that other researchers can replicate in their studies and that can serve as a turning point for schools in Latin America to take the development of computational thinking and the teaching of programming seriously within their curricula.

References

Balanskat, A., & Engelhardt, K. (2015). *Computing our future: Computer programming and coding priorities, school curricula and initiatives across Europe*. European Schoolnet. https://www.eun.org/resources/detail?publicationId=43856

Grover, S., Bienkowski, M., Niekrasz, J., & Hauswirth, M. (2016). *Assessing problem-solving process at scale*. In *Proceedings of the 3rd ACM Conference on Learning at Scale (L@S 2016)* (pp. 245–248). ACM. https://doi.org/10.1145/2876034.2893425

Guenaga, M., Eguíluz, A., Garaizar, P., & Gibaja, J. (2021). *How do students develop computational thinking? Assessing early programmers in a maze-based online game*. *Computer Science Education, 31*(2), 259–289. https://doi.org/10.1080/08993408.2021.1903248

Guzdial, M. (2008). *Education: Paving the way for computational thinking*. *Communications of the ACM, 51*(8), 25–27. https://doi.org/10.1145/1378704.1378713

Jiang, B., Wu, S., Yin, C., & Zhang, H. (2020). *Knowledge tracing within single programming practice using problem-solving process data*. *IEEE Transactions on Learning Technologies, 13*(4), 822–832. https://doi.org/10.1109/TLT.2020.3032980

Kalelioğlu, F. (2015). *A new way of teaching programming skills to K–12 students: Code.org*. *Computers in Human Behavior, 52*, 200–210. https://doi.org/10.1016/j.chb.2015.05.047

Klir, G. J. (1995). *Fuzzy logic: Uncovering its meaning and significance*. *Fuzzy Sets and Systems, 69*(2), 129–138.

Zadeh, L. A. (1988). *Fuzzy logic*. *Computer, 21*(4), 83–93.

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

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2013). *Learning computer science concepts with Scratch*. *Computer Science Education, 23*(3), 239–264. https://doi.org/10.1080/08993408.2013.832022

Moreno-León, J., Robles, G., & Román-González, M. (2020). *Towards data-driven learning paths to develop computational thinking with Scratch*. *IEEE Transactions on Emerging Topics in Computing, 8*(1), 193–205. https://doi.org/10.1109/TETC.2017.2734818

Repenning, A. (1993). *AgentSheets: A tool for building domain-oriented visual programming environments* (Demonstration paper).

Robles, G., Moreno-León, J., Aivaloglou, E., & Hermans, F. (2017). *Software clones in Scratch projects: On the presence of copy-and-paste in computational thinking learning*. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM.

Saito, T., & Watanobe, Y. (2020). *Learning path recommendation system for programming education based on neural networks*. *International Journal of Distance Education Technologies, 18*(1), 36–64. https://doi.org/10.4018/IJDET.2020010103

Wing, J. M. (2008). *Computational thinking and thinking about computing*. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 366*(1881), 3717–3725. https://doi.org/10.1098/rsta.2008.0118