2021 ASEE ANNUAL CONFERENCE
Virtual Meeting | July 26–29, 2021 | Pacific Daylight Time
Paper ID #34873

Using Visualizations of Students' Coding Processes to Detect Patterns Related to Computational Thinking

Dr. Markus Iseli, University of California, Los Angeles

Dr. Iseli is a Senior Research Scientist for CRESST with a focus on integration of engineering and technology for educational purposes. His specialization is in digital signal processing, speech and image analysis, pattern recognition, acoustics, and natural language processing. He has over 15 years of practical expertise as a technology and engineering consultant, applying data analysis and artificial intelligence algorithms for technology-based learning and knowledge assessment systems. Currently, he is involved as a knowledge engineer in various private and publicly funded projects. Dr. Iseli holds a PhD and an MS in electrical engineering from UCLA and from ETH Zürich, Switzerland.

Ms. Tianying Feng, University of California, Los Angeles
Dr. Gregory Chung, University of California, Los Angeles
Ziyue Ruan
Mr. Joe Shochet, codeSpark

Joe Shochet has been developing award-winning interactive experiences for 25 years. In 2014 he co-founded codeSpark, an edtech startup to teach kids the ABCs of computer science. His career started at Disney Imagineering building virtual reality attractions for the theme parks and designing ride concepts. Joe was a lead designer and developer of several virtual worlds including the popular Toontown Online, one of the first 3D virtual worlds for children. More recently he was Vice President at Rebel Entertainment, a division of IAC, focused on social and mobile games. Joe has a Computer Science degree from the University of Virginia, where his research focused on virtual reality, user interface design, and developing Alice3D.

Dr. Amy Strachman, codeSpark

The research reported here was supported by the Institute of Education Sciences, U.S. Department of Education, through Grant R305A190433 to the University of California, Los Angeles. The opinions expressed are those of the authors and do not represent views of the Institute or the U.S. Department of Education.

Using Visualizations of Students' Coding Processes to Detect Patterns Related to Computational Thinking

Introduction

Computational thinking (CT) has emerged as a key topic of interest in K-12 education. Children that are exposed at an early age to STEM curriculum, such as computer programming and computational thinking, demonstrate fewer obstacles entering technical fields [1]. Increased knowledge of programming and computation in early childhood is also associated with better problem solving, decision-making, basic number sense, language skills, and visual memory [2].

As a digital competence, coding is explicitly regarded as a key 21st Century Skill, as the "literacy of today," such that its acquisition is regarded as essential to sustain economic development and competitiveness [3]. Hence, the reliable evaluation of students' process data in context of problem solving tasks that require CT is of great importance.

As opposed to **product data**, which only contain information about *what* the outcome of a problem solving process was (e.g. the final score), **process data** contain information about *how* the problem was solved (e.g. all the actions and problem solving steps). Students' coding processes are thus defined by their actions while coding, as evidenced by "process data," and are evaluated by comparing their action sequences to optimal action sequences.

Prior research on process data analysis [4-9] shows several inherent issues. Their approaches aggregate data and thus loses information which precludes them from being used in more detailed analyses of student behavior. Vector-based approaches often apply dimensionality reduction or normalization and require interpretation of the reduced dimensions, which is often not possible. Network-based or finite state visualizations that show transitions between states (i.e., actions or game-states), are aggregations over the student, game level, or time dimensions and thus lose detailed information along these dimensions. Additionally, these networks only model Markov processes of order one (current state and preceding state) and do not show the frequency of higher-order sequences such as transitions through more than one preceding state. Sequential pattern mining approaches can deal with higher-order sequences, but their results tend to be verbose and need tedious manual analysis.

In summary, prior research has analyzed overall action sequences or code snapshots, but has not interpreted student actions in context of a situation during the problem solving process -- i.e. while students create the solution. A more fine-grained analysis of coding process data is needed, where relevant actions are interpreted as a part of the student's problem solving process.

This paper addresses some of above issues and presents an approach to detect patterns related to computational thinking based on visualizations of students fine-grained actions in situational context.

Participants and Data Collected from codeSpark

Students' coding process data from a class of 22 first grade students (12 females and 10 males, with an average age of 6.6 years, SD=0.6) at a charter school with 97% of the students coming from socioeconomically disadvantaged families, 56% English learners, and 96% Hispanic or Latino ethnicity was collected using codeSpark Academy. Data was collected in six weekly 45-minute in-class sessions, where each student was equipped with an iPad running codeSpark Academy. codeSpark Academy introduces children to programming and computational concepts (sequencing, parameters, loops, events, and conditionals) and combines carefully scaffolded puzzles aligned with the curriculum. The app presents students with puzzles that involve programming the moves of a character to reach a screen target by overcoming several obstacles.

To perform process data analysis, we designed a research version of codeSpark Academy, where game states and events, as well as all student actions were logged. During the six weekly in-class **sessions,** a total of 85,058 telemetry events were recorded. Telemetry data contain timestamps with events, which are listed together with their associated parameters in Table 1.

Table 1. Subset of telemetry events as captured in the research version of codeSpark Academy with their visualization. The column "Visualization Markers" contains markers that will be used in our visualizations, which will be discussed in the Methods section.

Telemetry Event	**Marker**
PuzzleStart: Sent at the beginning of every puzzle level	s
PuzzleResult: Sent at the end of every puzzle level	*,2*,3*
CommandAdded: A command tile is successfully added to the code tray	|
CommandParameterChange: A parameter on a command was changed	/
CommandRemoved: A command was removed from the code try	-
CommandReorder: A command was reordered within the code tray	none
StartExecuteTrigger: Sent when the student executes the code	^ info
EndExecuteTrigger: Sent when the code execution stops before	$

The puzzle levels are broken into five chapters based on the CT concept being introduced. We analyzed the first three chapters with the following character, setting, and game mechanics:

1. Chapter 1: "Donut Detective" (17 levels) featuring a detective, introduces the coding interface, has brief tutorials to show students how to enter and run code, and covers basic sequencing of commands like walking and jumping.
2. Chapter 2: "Tool Trouble" (16 levels) featuring a construction worker, introduces the loop command. The construction worker must do repetitive tasks so the ability to loop commands is beneficial. This chapter also introduces new commands.

3. Chapter 3: “Kite Plight” (17 levels) featuring a ninja, provides advanced sequencing puzzles. These puzzles are more difficult than Chapter 1 and often involve back-tracking and looping to get the correct solution.

Figure 1. Game interface representative of chapters 1 through 3.

The game interface, representative of each level for chapters 1-3, is shown in Figure 1. Each level consists of a selection of available commands (Fig. 1, bottom of screen), a code tray (orange-brown box in lower half of screen), and the game world, which contains the character, obstacles, breadcrumbs (green gems), and the final goal (golden fish) to be reached by the character. The goal for the student is to program each step of the character on its path to the goal, avoiding obstacles and collecting all the breadcrumbs on the way. A code length limitation of seven commands in the code tray, forced students to use loops in certain levels. Figure 1 shows the optimal solution in the code tray: On code execution, the ninja character will jump right four times, walk right, unlock the treasure chest (which contains a gem), walk left three times, until arrives at the goal (golden glowing fish).

We distinguish between two phases of game play: 1) the code construction phase, which ends with the student executing the constructed code; and 2) the evaluation phase, where the student observes the character’s moves and can interrupt at any time to reload the level.

In the **code construction phase**, at the beginning of each level sometimes an introductory screen in comic-book format describing the task or the background story is shown. After a cutscene, the level’s game interface shows the character at its initial position. The student can then drag commands into the code tray. Once commands are in the code tray, the student can either change their position relative to each other, change their parameter, or remove them from the tray. The following commands have parameters that can be changed: walk (left or right), jump (left, up, or down), throw (left or right), loop (number of times).

By tapping on the character, the **evaluation phase** begins, where the code in the code tray is executed with the character starting at its current position. If the character does not arrive at the goal, the game pauses; now, the student can either tap on the level reload button (Fig. 1, top right

of the screen), which will move the character back to its initial position but will not change the code in the code tray, or modify the code in the code tray and execute the modified code at the character's current position.

Students receive three "stars" for successful completion of each level -- no stars are awarded if the character does not arrive at the goal. If not all breadcrumbs (e.g. gems) are collected or if multiple code edits and executions were done during one attempt, the student receives only one or two stars. An "**attempt**" is considered the time span between the puzzle start event and a reload or exit event. In our study, students could move to the next level even when not having succeeded in prior levels. However, when stuck on a level, hints - but not the solution - were provided to them.

Methods

To detect patterns in students' coding processes, we created and analyzed visualizations of telemetry data. The focus of the analysis was to generate performance measures that can track a student's performance over time, with performance expressed as a similarity or distance to optimal performance. The final goal was to be able to visualize these measures in intuitive and efficient ways so that instead of having to watch hours of recorded game play on video, relevant information could be displayed at a glance.

Defining Performance Measures

We approached quantitative analysis from a machine learning perspective, where we envisioned teaching a computer how to learn to play the game. In order to learn, the computer will need to evaluate a performance measure that tells it how it is doing. Usually this is done by minimizing a distance (or cost) function expressed as a distance between the current solution and the optimal solution or between the current game state and the performance goal. Accordingly, in the code construction phase, where the goal is to construct the optimal code, the distance function is related to a **distance to optimal code**. In the evaluation phase, where the goal is to evaluate and recognize instances when something goes wrong (i.e., the character gets stuck or does not move along the optimal path), the distance function is related to the **estimated number of remaining steps** to reach the goal position.

We defined ***CodeDist*** as the **distance to optimal code.** We chose the Damerau-Levenshtein (DL) edit distance [10], which is defined as the minimum number of simple edit operations required to transform a given string of letters into another string. The following edit operations are considered when calculating the DL distance: insertions, deletions, substitutions of single letters, and transpositions of adjacent letters. For example, the DL distance between the strings "cats" and "fact" is 3 and hence can be achieved by three transformation operations as either cats > fcats (insertion), fcats > facts (transposition), and facts > fact (deletion); or as cats > fats (substitution), fats > facts (insertion), and facts > fact (deletion). In any case, the minimum number of edit operations is 3, which is the DL edit distance. To compare between the current student code and the optimal code, each command in the code corresponds to a letter in the string: Adding a command corresponds to insertion, removing a command corresponds to deletion, and moving a command corresponds to transposition. Note that substitution operations are not possible in the game.

The code in the code tray is represented as an abstract syntax tree (ASTs) in Lisp-like format with parentheses. This format is easy to generate and easy to read. For example, the optimal solution for the level shown in Figure 1 is expressed as follows:
(Loop.4(,Jump.R,),Walk.R,Unlock,Loop.3(,Walk.L,))

Each command consists of a command and an optional parameter, separated by a dot from the command, e.g., "*Loop.4(*" means that the loop command is executed four times. Similarly, the parameters "*x.L*," "*x.U*," "*x.R*" denote parameters "left," "up," and "right." Additional commas before or after parentheses act as delimiters to make the command sequence interpretable as a string to which the DL distance can be applied. Our implementation of the DL distance first calculates the DL distance on the sequence of commands, ignoring the parameters, each operation counting with weight 1, and then adds a weight of 0.5 for each parameter difference. For example the *CodeDist* between *Loop.2(,Walk.L,Jump.U,),Throw.R* and *Throw.R,Loop.3(,Walk.L,),Jump.U* is 4 and uses the following transformations:

Loop.2(,Walk.L,Jump.U,),Throw.R	
Loop.2(,Walk.L,Jump.U,)	(deletion: weight 1)
***Throw.**,Loop.2(,Walk.L,Jump.U,)*	(insertion: weight 1)
*Throw.,Loop.2(,Walk.L,**),Jump.U***	(transposition: weight 1)
*Throw.**R**,Loop.**3**(,Walk.L,),Jump.U*	(change of 2 parameters: 1 = 2 x 0.5)

We defined ***PathDist*** as the **estimated number of remaining steps** to reach the goal position. The optimal path through a level's world is expressed as a list containing the x/y-coordinates of the optimal path positions starting at the character's initial position and ending at the goal position.

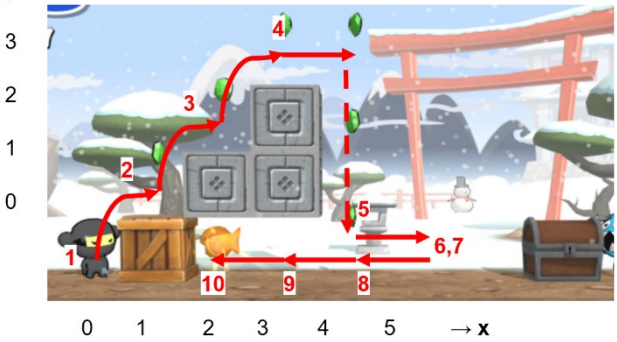

Figure 2. Illustration of the x/y coordinate system used to calculate *PathDist*. The optimal path is indicated by red arrows, which represent the steps that need to be taken along the path.

Figure 2 shows the x/y coordinate system with the optimal path overlaid on top of a game play level scenario for which the character's initial position is at the origin (x=0, y=0).The list of the optimal path for the example in the figure is: [1:(0,0), 2:(1,1), 3:(2,2), 4:(3,3), 5:(4,0), 6:(5,0), 7:(5,0), 8:(4,0), 9:(3,0), 10:(2,0)]. The list includes step numbers, separated by colons. This list contains ten steps *(total_steps*=10). Every step corresponds to the execution of one command, hence when the character arrives at position (5,0) in step 6, it unlocks the treasure and stays at position (5,0) in step 7. Depending on at what step the character is currently at (*current_step*), *PathDist* is calculated as: *PathDist = total_steps - current_step*

If the character deviates from the optimal path, *current_step* is defined as the step on the optimal path that is closest to the character's position. Given the optimal path from our example, if the character is at position (-2,0), the closest position on the optimal path is at position (0,0), thus *current_step* = 1, with *PathDist* = 9. If the character is closest to more than one step on the path, the current step which is closest to the previous step (*previous_step*) is taken. E.g., if the character is at position (6,0), steps 6 and 7 on the path are closest. If *previous_step* <= 6, then *current_step* = 6 with *PathDist* = 4; and if *previous_step* >= 7, then *current_step* = 7 with *PathDist* = 3. Depending on the attempted optimal solution, a different optimal path is selected.

Visualization of Performance Measures

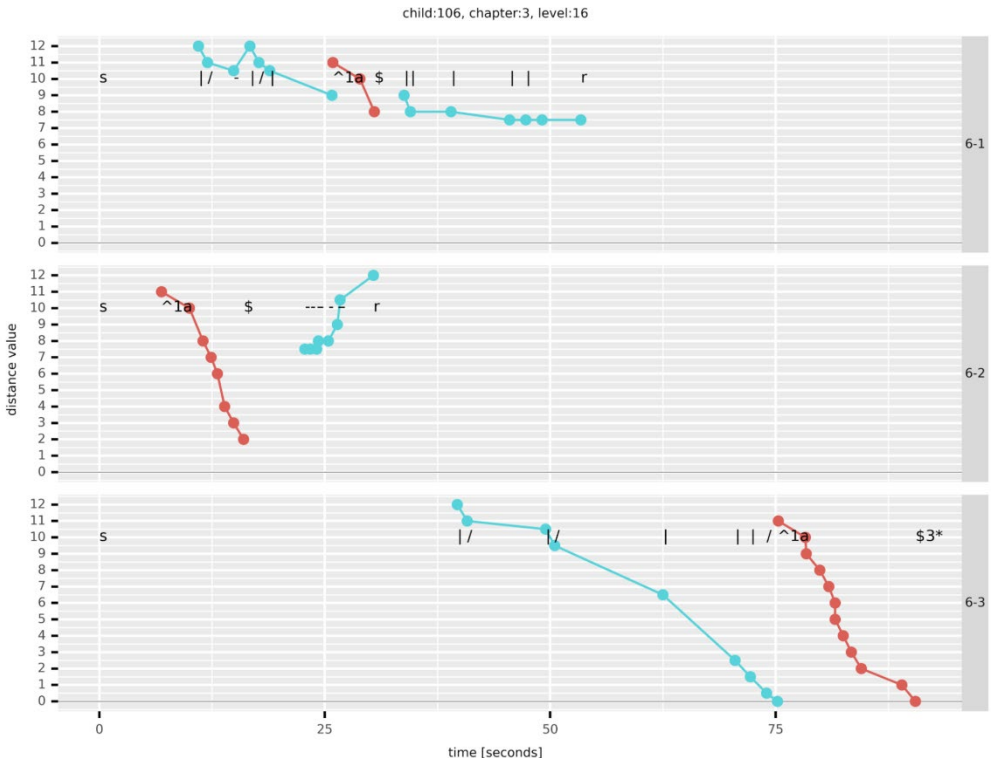

Figure 3. Visualization for child 106, chapter 3, level 16. *CodeDist* (blue curve, code construction phase), *PathDist* (red curve, evaluation phase).

The visualization of students' coding processes distinguishes between the code construction phase and the evaluation phase and displays the corresponding measures *CodeDist* and *PathDist,* together with relevant events for each student and game level as a function of time for each session and attempt. A common visualization is depicted in Figure 3: The x-axis shows the time in seconds, whereas the y-axes show the distance measures for *CodeDist* (blue curve, code construction phase) and for *PathDist* (red curve, evaluation phase). To include information about events that happen along the timeline, markers for each of the events were added as defined in Table 1. Analyzing the visualization shown in Figure 3, the following information can be read at a glance. The student (ID 106) played chapter 3, level 16, in session 6 (week 6) using three attempts: 6-1, 6-2, and 6-3. In attempt 1, for example, the student adds a command and changes its parameter (/), then removes the command, emptying the code tray, then adds another command with parameter change, and finally adds one last command reducing CodeDist from 12

to 9. Around 25 seconds in the game, the student executes the code (^) and the character stops moving around second 27 ($), reducing *PathDist* from 11 to 8.

Results & Discussion

Analyses of our visualizations of students' coding processes yielded several patterns that were mapped to the six constructs of computational thinking as extracted by [11]: abstraction, decomposition, generalization, modeling, algorithmic thinking, and evaluation. We will present and discuss the detected patterns, organized by CT construct. The results are summarized in Table 2. All analyses were based on non-tutorial levels in order to reduce inputs resulting from students' unfamiliarity with newly introduced game mechanics or concepts.

Ideal performance looks as depicted in Figure 3, panel 6-3, where both *CodeDist* and *PathDist* are steadily reduced to zero, and the student is using only code additions (|) and parameter changes (/) when editing the code tray.

Non-ideal performance for *CodeDist* manifested by non-steady reduction of distances, e.g., *CodeDist* increases and decreases and edit operations such as deletions and transpositions are used. Non-ideal performance for *PathDist* happens either when *PathDist* does not reduce (character got stuck on its path to the goal) or when *PathDist* drops steeply (character took a shortcut and misses some gems).

The observable behavior of **abstraction** is defined as adding a command and changing parameters at once, or as neglecting distractors or details (see Table 2 below). Examining the visualizations, we originally expected that (a) some students would add all commands and then change their parameters at once and that (b) other students would change the parameter after each added command. However, it could be seen that the second behavior (b) was prevalent, and that the first behavior (a) would only occur if there was a misconception present. Figure 4 shows an example of the first behavior, where it turns out that many students did not realize that the default parameters were set to ".R" instead of ".L". Once students realized that the character was walking in the wrong direction, they interrupted and fixed the code. Another interesting behavioral pattern we found was that after a reset, some students removed all commands from the code tray, perhaps to start fresh and not get distracted with non-functioning code. Figure 5 shows such an example. We are currently still debating whether this behavior is related to abstraction (prioritizing) or decomposition (not mixing different attempts), a combination of both or some other construct.

The CT construct of **decomposition** was defined as the behavior of approaching a level's solution step by step. Figure 6 shows such an example where a student first reduces *CodeDist* from 10.5 to 9, executes the code until execution stops, which reduces the *PathDist* from 7 to 6 (the character has not yet reached its goal position). Without reloading, the student continues editing the code (*CodeDist* = 3), executes the code (*PathDist* = 2), edits the code (*CodeDist* = 0), and executes (*PathDist* = 0), solving the problem and receiving two stars. Because of multiple code executions within the same attempt, only two and not three stars are awarded.

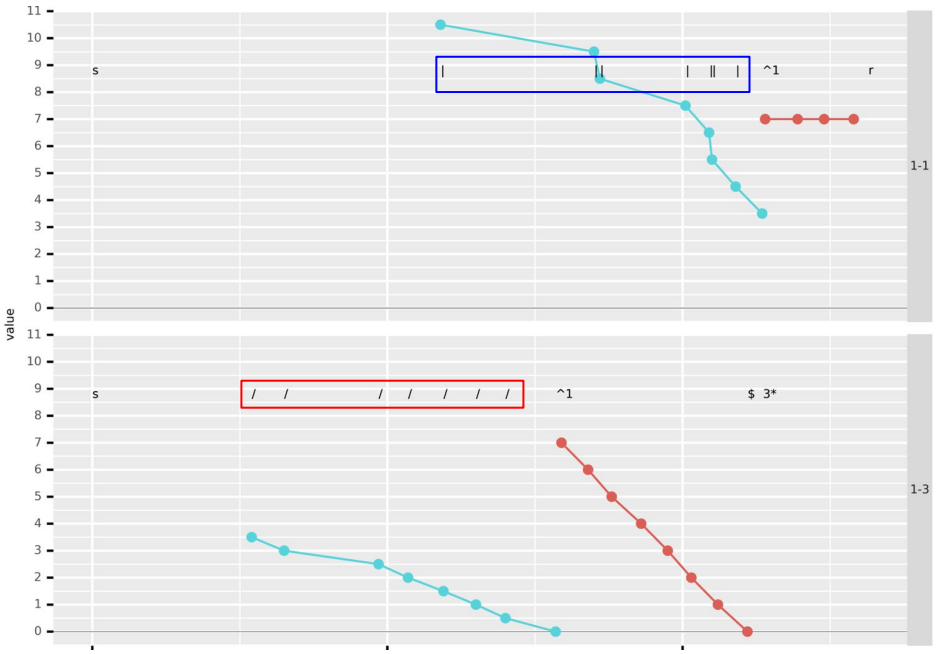

Figure 4. Panel 1-1: Student adds all commands without changing parameters (see blue rectangle), executes the code, reloads before code execution ends. Panel 1-3: Student changes all parameters at once (see red rectangle) and code executes with three stars.

Figure 5. Panel 1-1: Student edits code, executes it, and after unsuccessful completion, reloads the level. Panel 1-2: Student cleans code tray by removing all commands (see red rectangle), adds commands (*CodeDist* = 0), executes the code and receives three stars.

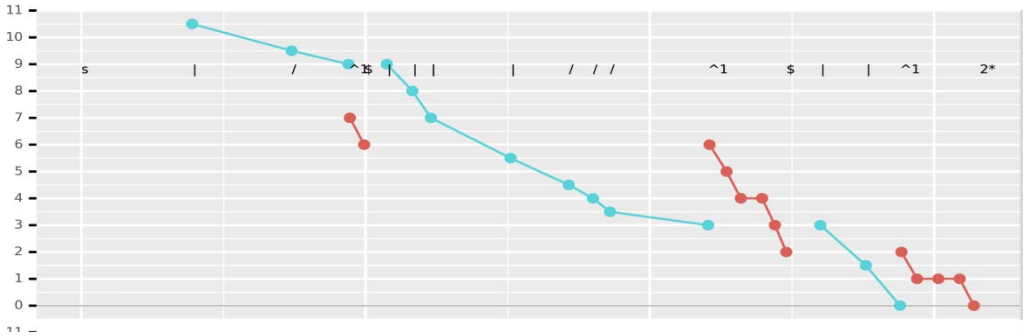

Figure 6. Decomposition: Student edits code, executes (^), edits some more, executes, edits more, executes, eventually gets two stars (not three stars, because of multiple code changes).

Table 2 (see below) defines the behavior of the CT construct of **generalization** to be related to using loops correctly and identifying repeated or similar game elements. From the provided explanations of suboptimal solutions ("no/missing/infinite loop"), we know if a student used loops correctly when needed. Note that not every game level requires the use of loops. Our visualizations currently do not allow for cross-level comparisons, however.

The concepts of **modeling and algorithmic thinking** seem to be intertwined. We believe that a correct model is required to guide algorithmic thinking to generate a correct algorithm. Trying to infer modeling and algorithmic thinking from the observed coding process is not obvious: If the coding process shows a certain pattern, was it due to either presence or lack of modeling or algorithmic thinking or both? For example, Panel 1-1 in Figure 5 shows the student reducing *CodeDist* from six to three, using seven code edits including two edits that do not reduce *CodeDist*. This pattern of not steadily reducing *CodeDist* to zero indicates a lack of either modeling or algorithmic thinking. However, in the student's second attempt (Figure 5, Panel 1-2), after removing all commands from the code tray, the student steadily reduces *CodeDist* from six to zero in four *CodeDist*-reducing edits. This pattern indicates correct modeling and algorithmic thinking skills. After careful evaluation of our data, we decided that we could not infer modeling and algorithmic thinking separately. Thus, the behavior that is required for both constructs should show assembly of commands in the order the algorithm intended, which, using visualizations, could be defined as the steady reduction of *CodeDist* down to zero with or without using deletion or transposition operations (see Table 2).

Finally, the CT construct of **evaluation** uses comparison of the (mental) model with reality: i.e., if the character in the game does not do what it is supposed to do, evaluation should start the debugging process, which will include all the CT constructs, but especially modeling and algorithmic thinking. As defined in Table 2, the behavior for evaluation includes interruptions. Interruptions during the code construction phase indicate that the student is comparing the algorithmic expression in the code tray to the mental model (evaluation, algorithmic thinking) and exits or reloads the level before executing (^) the code, see Figure 7, upper panel. Interruptions during the evaluation phase indicate that the student realizes that the code is not behaving as the mental model predicted (evaluation) and thus reloads the level before the code stops executing ($), see Figure 8 lower panel. Once interrupted, the student will start the debugging process by either updating the mental model (modeling) or the algorithm (algorithmic thinking) or both.

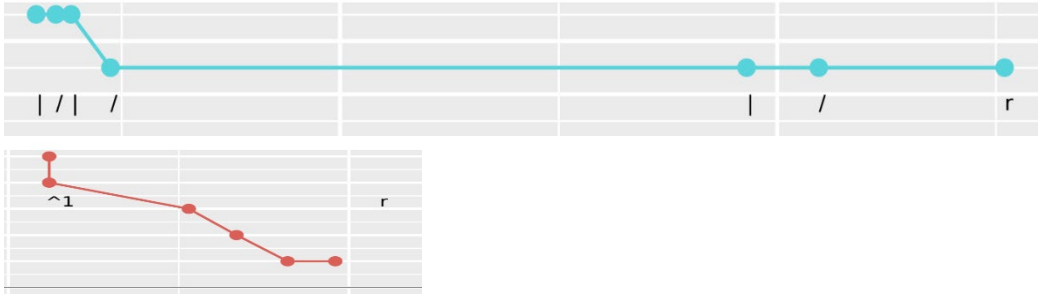

Figure 7. Upper panel: While editing the code, the student might realize that the code does not do what it was supposed to (evaluation, algorithmic thinking), and decides to reload (r) the level instead of executing the code. Lower panel: Student realizes when the character gets stuck (*PathDist* not changing, evaluation) and reloads (r) *before* code execution ends.

Other patterns found in our data were associated with *CodeDist* not being reduced to zero while *PathDist* was reduced to zero with 3 stars. Reasons for this were identified as: 1) The list of solutions was missing a correct, but suboptimal solution; 2) The student's code had superfluous elements that make the character move beyond its goal position (e.g., by using an infinite loop); 3) Loops with repetition 1 were used (Loop.1 was allowed), which was not accounted for.

Table 2 summarizes the results of our analysis using visualizations of students' coding processes. It shows the six CT constructs together with their definitions and behavioral patterns.

Table 2. Definition of computational thinking constructs with possible observable student behavior for the codeSpark Academy game puzzles. Note that after careful evaluation of our data, we decided that we could not infer modeling and algorithmic thinking separately.

CT Construct Definition	codeSpark Student Behavior
Abstraction ○ Simplify a construct (idea, problem factors and components) at different levels of detail (levels of abstraction, LOA). ○ Reduce construct complexity by determining relevant LOAs.	○ Add a command and change parameters at once if needed. ○ Neglect distractors or details.
Decomposition ○ Break/factor a construct (e.g., a problem) into smaller, less complex, easier to process parts.	○ Approach a level's solution by first achieving part of the goal then adding more commands.
Generalization ○ Recognize and identify patterns in space and time, find and exploit similarities and connections (e.g., rules) to other patterns. (exploit = reuse in other situations, transfer).	○ Use loops correctly (e.g., no loops with repetition 1, no loops containing a repetition of the same commands). ○ Identify repeated or similar game elements across chapters and levels.
Modeling ○ Create a (mental, physical, graphical) representation of the past/present/future state and behavior of a construct or concept. ○ Define a cost function (difference between model and goal) that describes the quality of the representation.	○ Make code edits that are mostly correct. ○ Avoid removing and moving commands (deletions and transpositions), once placed in the code tray. ○ Add commands in order, as defined by the algorithm.
Algorithmic Thinking ○ Define a clear sequence of steps (e.g., instructions and rules), given a model ○ Execute a sequence of given steps (an algorithm).	

Evaluation ○ Check/test/analyze if model cost function is minimized. ○ If the cost function is not minimized, start debugging.	○ Interrupt code editing when code does not look right. ○ Interrupt game play execution as soon as an error happens (e.g., character gets stuck).

Conclusion and Future Work

This paper focused on whether patterns of systematic player behavior can be detected using visualizations. We consider this as a first step in the process of developing algorithms to automatically detect these patterns, construct indicators from these patterns, and integrate these indicators into probabilistic or statistical models, such as predictive, discriminative, or generative models of performance.

We presented a visualization approach for the analysis of coding process data. This approach showed the following benefits: (a) It does not require the definition of process states; (b) It does not accumulate data (either across students or over time) and thus preserves the raw information aspect of the data; (c) It is goal-oriented, by being based on well-defined and measurable performance objectives; (d) It facilitates the definition of specific performance similarity measures for each performance objective (e.g. or distance to optimal code or distance to optimal path), and thus facilitates scoring; (e) It is independent of sequence data length and thus enables time series analysis (e.g. frequency, pauses, etc.) (f) It can visualize each student's performance for each measure as a function of time; and (g) It can be used to inform the feature extraction process by facilitating pattern identification.

The visualizations of student process data (a) clearly showed groups of patterns that represent different strategies related to the computational thinking constructs: abstraction, decomposition, generalization, modeling, algorithmic thinking, and evaluation (see results in Table 2); and (b) were shown to be able to guide algorithm development to detect patterns related to computational thinking.

While the current research is based on process data from the codeSpark Academy game, the same approach could be used with other programs such as Scratch or Scratch Jr. If adopted by third parties, our work could contribute to the automated scoring of programs and the implications that flow from this capability—individualized and immediate diagnostic feedback to the student and diagnostic feedback to teachers, incorporated into large-scale assessment programs such as NAEP or PISA that would allow those programs to offer assessments of computational thinking.

In future research we plan to: a) Extract new, visualization-informed measures derived from the time series used in these visualizations. E.g., first and second order derivatives, means and variances to infer CT constructs can be used. b) Use the newly extracted measures to provide real-time feedback to students and/or teachers. c) Explore the use of the newly extracted measures as new features for inputs to machine learning models and algorithms.

References

[1] H. M. Madill, R. G. Campbell, D. M. Cullen, A. A. Einsiedel, A.-L. Ciccocioppo, and M.-A. Armour, "Developing Career Commitment in STEM-related Fields: Myths versus Reality," in *Women and Minorities in Science, Technology, Engineering and Mathematics*, Edward Elgar Publishing, 2007.

[2] L. P. Flannery, B. Silverman, E. R. Kazakoff, M. U. Bers, P. Bontá, and M. Resnick, "Designing ScratchJr: support for early childhood learning through computer programming," in *Proceedings of the 12th International Conference on Interaction Design and Children - IDC '13*, New York, New York, 2013, pp. 1–10, doi: 10.1145/2485760.2485785.

[3] S. Bocconi, A. Chioccariello, G. Dettori, A. Ferrari, and K. Engelhardt, *Developing Computational Thinking in Compulsory Education*. European Commission, Joint Research Centre, 2016.

[4] A. A. Supianto, T. Y. Christyawan, M. Hafis, Y. Hayashi, T. Hirashima, and N. Hasanah, "Feature Dimensionality Reduction for Visualization and Clustering on Learning Process Data," in *2019 International Conference on Sustainable Information Engineering and Technology (SIET)*, Sep. 2019, pp. 84–89, doi: 10.1109/SIET48054.2019.8986020.

[5] M. Zhu, Z. Shu, and A. A. von Davier, "Using Networks to Visualize and Analyze Process Data for Educational Assessment," *Journal of Educational Measurement*, vol. 53, no. 2, pp. 190–211, 2016, doi: https://doi.org/10.1111/jedm.12107.

[6] J. Hao, Z. Shu, and A. von Davier, "Analyzing Process Data from Game/Scenario-Based Tasks: An Edit Distance Approach," *Journal of Educational Data Mining*, vol. 7, no. 1, pp. 33–50, 2015.

[7] D. Jurafsky, *Speech & language processing*. Pearson Education India, 2000.

[8] M. Kong and L. Pollock, "Semi-Automatically Mining Students' Common Scratch Programming Behaviors," in *Koli Calling'20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*, 2020, pp. 1–7.

[9] B. Jiang, S. Wu, C. Yin, and H. Zhang, "Knowledge Tracing Within Single Programming Practice Using Problem-Solving Process Data," *IEEE Transactions on Learning Technologies*, vol. 13, no. 4, pp. 822–832, 2020.

[10] R. A. Wagner and R. Lowrance, "An extension of the string-to-string correction problem," *Journal of the ACM (JACM)*, vol. 22, no. 2, pp. 177–183, 1975.

[11] M. R. Iseli, Y. Zhang, G. K. W. K. Chung, J. Shochet, A. Strachman, and G. Hosford, "Defining Computational Thinking using Semantic Analysis of Prior Definitions," presented at the Connected Learning Summit, Boston, Massachusetts, 2021.