



# Experimental Analysis of First-Grade Students' Block-Based Programming Problem Solving Processes

Gabriele Pozzan  
University of Padua  
Padua, Italy  
gabriele.pozzan@phd.unipd.it

Costanza Padova  
University of Padua  
Padua, Italy  
costanza.padova@phd.unipd.it

Chiara Montuori  
University of Padua  
Padua, Italy  
chiara.montuori@phd.unipd.it

Barbara Arfé  
University of Padua  
Padua, Italy  
barbara.arfe@unipd.it

Tullio Vardanega  
University of Padua  
Padua, Italy  
tullio.vardanega@unipd.it

## ABSTRACT

This work presents an experimental analysis of first-grade students' block-based programming trajectories. These trajectories consist of edit-level program snapshots that capture learners' problem-solving processes in a navigational microworld. Our results highlight the potential of this fine-grained data capture. Snapshot frequencies in trajectories collected before and after a coding intervention showcase the collective progress of the learners. Graph visualizations, in which nodes represent snapshots and directed edges code edits, highlight strategies, pitfalls and debugging procedures. Individual programming trajectories shed light on details of learners' problem-solving processes that less granular analysis would conceal. Various works in the field of Learning Analytics research show the usefulness of collecting fine-grained process data that proceed from programming activities. However, how to analyze this data is still an open question and research on the subject is in an experimental phase. We contribute to this experimentation by analyzing and discussing results collected from 30 first-grade students in a pretest-posttest study.

## CCS CONCEPTS

- **Social and professional topics** → **Computational thinking;**
- **Applied computing** → **Interactive learning environments.**

## KEYWORDS

Learning Analytics, Computational Thinking, Block-based programming, Programming trajectories

## ACM Reference Format:

Gabriele Pozzan, Costanza Padova, Chiara Montuori, Barbara Arfé, and Tullio Vardanega. 2024. Experimental Analysis of First-Grade Students' Block-Based Programming Problem Solving Processes. In *Proceedings of the 2024 Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024)*, July 8–10, 2024, Milan, Italy. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3649217.3653586>



This work is licensed under a Creative Commons Attribution International 4.0 License.

ITiCSE 2024, July 8–10, 2024, Milan, Italy  
© 2024 Copyright held by the owner/author(s).  
ACM ISBN 979-8-4007-0600-4/24/07  
<https://doi.org/10.1145/3649217.3653586>

## 1 INTRODUCTION

Research on Computational Thinking (CT) has seen interest in recent years [10]. Helping learners develop CT skills is a goal often motivated by the need for a digitally-skilled workforce [3, 5]. Other motivations include the role that computation itself has as a field of knowledge [6] and the importance for learners to make the step from passive consumers to creators of digital content [18].

While CT does not equate with programming [21], one of the main ways to develop its skills is indeed through learning to solve problems using programming artifacts [4]. Block-based programming languages have shown potential in helping novices learn how to interact with programmable systems: among the advantages they are reported to offer are reduced cognitive loads and the elimination of syntactic errors [3]. With regard to CT learning problems targeted at novices, microworlds have been used since the very beginnings of research in CT [14] and, in recent years, have grown in variety, goals and intended audiences [16].

Regardless of the specific programming language and problem solving environment, learning activities that involve interaction with a digital system automatically generate a wealth of data, the study of which is the core of Learning Analytics research. We divide this information in *product* data - programming artifacts in some form of "finished" state, e.g. executed programs - and *process* data, i.e. pure human-machine interactions like clicks and keypresses, or code snapshots representing sequences of code edits.

Product data is often used in conjunction with grading rubrics to assess CT competencies [22].

Process data help capture details that highlight strategies and patterns that disappear from finished products. Research in this direction experimented with the analysis of *programming trajectories*, i.e. sequences of code snapshots that show the evolution of learners' programs during problem solving activities [11, 17]. Trajectories capture details such as common errors and strategies used to overcome them. Other research analyzed process data using data mining techniques such as clustering [8, 13].

This work leverages programming trajectories to study problem solving strategies from collective and individual points of view. Being able to visualize and study this information has been reported by CT educators as a valuable contribution [23]: collective insights about learners can help direct educational activities; individual trajectories can show details about thinking processes and support instructors in better assisting learners in need. We aim to 1) show

how to extract information from fine-grained process data, via analyses and visualizations; 2) give examples of how this information could be used to aid educators and learners in real-life learning contexts.

We conducted an experiment with 30 first-grade students, whose programming proficiency we tested before and after a coding intervention. The tests consisted of block-based programming games in navigational microworlds, similar to those offered by the platform Code.org. During these evaluations we collected interaction data that allowed us to reconstruct and analyze the students' programming trajectories.

The remainder of this paper is organized as follows: Section 2 discusses related works. Section 3 outlines our methodology. Section 4 reports our results. Section 5 draws conclusions on this work.

## 2 RELATED WORK

Our design foundations are inspired by the work of Pelánek and Effenberger who, in [15], thoroughly discuss the design of block-based programming microworlds. In particular, when addressing data logging, they define *edits* in block-based programming languages every action that changes one block by insertion, deletion, reordering or field change. We analyze data at this level of granularity. We collect code snapshots after each edit and are thus able to reconstruct programming trajectories, from start to finish.

Other works in the literature discuss the analysis of process data collected from programming activities.

Tsung et al., in [23], present BlockLens: a system capable of logging learners' block edits and use them to create programming trajectories. The system is built to meet requirements collected by the authors from educators working with CT in K-12 contexts. These requirements suggest that a system aimed at supporting CT education should be able to: 1) analyze learners' performance collectively on a single task, e.g. by showing the success rate of the task; 2) highlight different problem solving strategies; 3) recognize milestone snapshots, i.e. those that strongly correlate with correct solutions, or early warning signs; 4) summarize problem solving processes; 5) present individual processes and strategies in detail. The paper then proceeds to showcase BlockLens and to present examples of its usage, however, it does not discuss data collected from real educational contexts. We aim to fill this gap by discussing and analyzing data collected from an actual experimentation.

Piech et al., in [17], use edit level program snapshots to automatically generate hints targeted at learners. Their analysis is based on Code.org's 2013 Hour of Code dataset (HoC2013), which only contains *product* data, i.e. code snapshots that learners actually executed. Thus, the authors interpolate code snapshots with the most likely sequence of edits that separates them. This is an interesting solution to the problem of working with lower-granularity datasets. However, the authors observe that, even for simple programming problems, the sets of solutions proposed by learners tend to be very large. In other words: learners tend to collectively submit sets of different solutions with high cardinalities, even for relatively straightforward problems. This, of course, also applies to the sequences of edits that lead to those solutions. Thus, by logging the actual code edits we are able to see "outlier" problem solving

processes that would not be inferred by automated interpolation, such as the misstep presented later in Figure 6.

Jiang et al., in [11], also use the HoC2013 dataset to study learners' problem solving trajectories in block-based programming activities. Their analysis is based on sequences of executed code snapshots. They study snapshot frequencies and how learners' partial solutions evolve with subsequent executions. In particular, they analyze the *Abstract Syntax Tree distance* (AST distance) between partial solutions and final, successful, solutions to a problem. In this work, we use the same algorithm discussed by Jiang et al., i.e. the Zhang-Shasha algorithm [25], to study edit level snapshots.

Zhang et al., in [26], use process data to study undergraduate students' debugging strategies in Java programming tasks. They analyze programming trajectories consisting of sequences of code submissions, similarly to the previously discussed work by Jiang et al. By using an automated error-checking system that labels each submission with the errors it contains - e.g. syntax errors, failed unit tests, etc. - the authors are able to compute metrics such as the rate of style/syntax/unit test errors fixed between subsequent submissions and the time these fixes required. Measuring the time spent on a specific snapshot could reveal insights on problem solving processes, we discuss this further in Section 4.4.

Other research has discussed the collection and analysis of granular data from a data mining point of view. The following works' approach differs from ours in the choice of programming environment - open-ended programming tasks in Scratch vs. navigational microworlds *a la* Code.org - and in the data analysis, based on machine learning techniques such as clustering.

Kesselbacher and Bollin, in [13], collect fine-grained interaction data for a single, open-ended, Scratch programming exercise. The data consists of learners' interactions with the programming task: block creation, deletion, drag-and-drop, field-changes. The low-level interactions, called "block listen events", are then combined to form higher-level "program change events". For example: the program change event "add new block to program" is the combination of the block listen events "create new block", "drag block" and "attach block to program". The authors use vectors representing frequencies of events in learners' problem solving processes to create clusters. For example, they find that those who execute their programs frequently tend to be less successful in solving the task.

Filvà, et. al, in [8] perform clickstream analysis - i.e. analysis of data representing each mouse click and keyboard stroke - on learners interactions with Scratch. As in the previously discussed work by Kesselbacher and Bollin, they use this data to perform clustering analysis on learners' processes.

## 3 METHOD

### 3.1 Experimental setup

We conducted a pilot quasi-experimental study with pre-posttest assessment [9]. We evaluated and showcased techniques to analyze and visualize fine-grained process data. To this end, during an academic semester, we involved a total of 30 first-graders from 2 class groups from the same primary school in northern Italy.

We assigned each class group to a different experimental condition: the *experimental* group consisted of 14 students (M=5, F=9);

the *waiting-list* group consisted of 16 students ( $M=6$ ,  $F=10$ ). The students were 6 years old at the time of the experimentation.

We evaluated both groups' proficiency with block-based programming with a pretest in January 2023 and a posttest in April 2023. The experimental group received a programming intervention between the pre-posttest evaluations. The waiting-list group received the intervention after the posttest.

This study followed the ethical code of conduct guidelines of, and was approved by, the Institutional Review Board of the University of Padua. We involved the students in the project after receiving signed informed consent from their legal guardians.

### 3.2 Programming intervention

The programming intervention consisted of 8 training sessions, each of the duration of 1 hour. During each session one or more instructors, part of the research group, provided each student with a tablet device. The students used the tablets to solve programming exercises aimed at teaching the following learning goals: sequencing of instructions, use of loops, debugging.

Each exercise presented the students with a tile-based map, containing a goal, a path, obstacles and a sprite to guide to the goal, e.g. the bee of Figure 1. Moreover, each exercise provided a set of code blocks used to program the sprite's movement: move forward 1 tile, turn left/right, repeat instructions a set amount of times.

Students were asked to attempt to solve the programming tasks autonomously: they could ask for help by the instructors - who were instructed never to give away the solutions but only provide hints - and discuss solutions with each other.

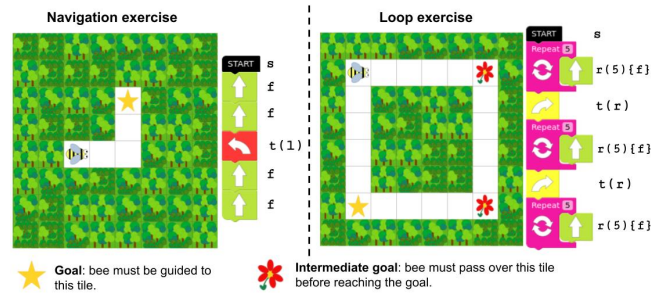
### 3.3 Programming evaluation

We conducted the pre-posttest programming evaluations on individual students. Each student was assigned to an instructor who provided assistance by: 1) conducting an initial briefing that described the goals of the evaluation tasks and the function of the code blocks that would be available to solve them; 2) checking in with the student after a trial exercise to ensure that there were no doubts, and 3) providing assistance in the usage of the tablet when necessary.

After the trial, the evaluation consisted of five exercises of increasing difficulty. Taking inspiration from research by Kanellopoulou et al. [12], we associated difficulty to the lengths of the paths, the number of necessary turn instructions and the complexity of the available code blocks. Easier exercises had short paths, few turns and only basic movement blocks (move forward, turn left/right). Harder exercises had longer paths, more turns and included loop blocks.

Due to the limited time slots negotiated with the schools to conduct the evaluations, the students were given a maximum of 3 attempts per exercise. For each exercise, students were scored  $4 - n$ , where  $n$  was the number of attempts necessary to solve it. Solving an exercise at the first attempt thus gave 3 points, solving it at the third attempt 1 point. Failing to solve an exercise for three times in a row resulted in 0 points. Total scores ranged from 0 to 15.

In this work we analyze two exercises, shown in Figure 1, which stand at the opposites of the spectrum of difficulty. The Navigation exercise requires the sprite to traverse a short path with a



**Figure 1: Exercises used in our data analysis, shown along programs that solve them, presented in block form and textual codification.**

single turn. The Loop exercise involves a longer path, two turns, intermediate goals, and the loop block.

In order to observe improvements in performance and the evolution of problem-solving processes, we used the same exercises for both the pretest and posttest.

## 4 RESULTS & DISCUSSION

### 4.1 Overall pre-posttest performance

The experimental group scored, on average, 2.000 ( $SD = 3.942$ ) points at the pretest and 8.714 ( $SD = 4.874$ ) at the posttest. The waiting-list group scored 0.125 ( $SD = 0.342$ ) points at the pretest and 2.813 ( $SD = 3.270$ ) at the posttest. As the proficiency scores were not normally distributed in both groups, we performed a Wilcoxon/Kruskal-Wallis test to compare the two groups. Before the intervention, the proficiency scores distribution was similar among the two groups ( $\chi^2 = 1.714$ ,  $p = .19$ ,  $df = 1$ ). After the intervention, the proficiency score distribution of the experimental group was significantly different from the waiting-list one ( $\chi^2 = 10.297$ ,  $p = .0013$ ,  $df = 1$ ). We calculated effect size by estimating  $r = Z/\sqrt{N}$  [20]: the effect size of the difference between experimental and waiting-list groups at the posttest was  $r = .58$  (large). These results show that the coding intervention had a significant effect on the programming proficiency of the experimental group.

### 4.2 Code snapshots

We regard as *snapshots* all intermediate programs that learners create while combining blocks to compose exercise solutions. Figures 6 and 7 showcase some examples. In writing, we refer to snapshots using a textual codification: the start blocks at the root of every program are represented by  $s$ ; move forward blocks by  $f$ ; turn left/right blocks by  $t(l/r)$ ; loop blocks repeating instructions  $n$  times by  $r(n)\{\dots\}$ .

Analyzing snapshots in a cumulative way helps capture how the strategies of a group change after a period of exposure to coding.

Table 1 presents statistics about the cumulative snapshots the students of the two groups produced during their pre-posttest problem solving processes. The Table shows 1) the number of distinct snapshots produced by the students, and 2) the average AST distance - i.e. the number of edits necessary to transform the snapshots ASTs into those of the closest goal programs - weighted by snapshot

| Navigation Exercise          |                |          |                   |          |
|------------------------------|----------------|----------|-------------------|----------|
| goal: <code>sfft(1)ff</code> |                |          |                   |          |
|                              | # of Snapshots |          | Avg. AST Distance |          |
| Group                        | Pretest        | Posttest | Pretest           | Posttest |
| Experimental                 | 73             | 24       | 6.248             | 2.798    |
| Waiting-list                 | 77             | 64       | 5.538             | 4.959    |

| Loop Exercise   |                |          |                   |          |
|---|----------------|----------|-------------------|----------|
| goal (using loops): <code>sr(5){f}t(r)r(5){f}t(r)r(5){f}</code> |                |          |                   |          |
| goal (without loops): <code>sfffft(r)fffft(r)ffff</code>        |                |          |                   |          |
|   | # of Snapshots |          | Avg. AST Distance |          |
| Group   | Pretest        | Posttest | Pretest           | Posttest |
| Experimental  | 203            | 90       | 12.523            | 7.259    |
| Waiting-list  | 136            | 192      | 12.574            | 11.453   |

**Table 1: Number of distinct snapshots and their average, weighted, AST distance from the goal programs.**

frequency. Goal programs are listed in the rows below the exercise title using the previously described textual codification.

The results show how the problem-solving process of the experimental group becomes more focused after the intervention: the number of distinct code snapshots is more than halved for both exercises. Moreover, the problem-solving process also becomes more successful: the weighted AST distance between the snapshots and the goal programs is approximately halved for both exercises. This means that intermediate snapshots on programming trajectories tend to be closer to actual solutions. On the other hand, students in the waiting-list group continue to produce a large number of different code snapshots - in the case of the Loop exercise, they even increase this number - and only slightly reduce the weighted average AST distance of these snapshots.

AST distances give an idea of how close or far from success programs tend to be. Analyzing which snapshots are actually the most commonly produced by the learners can also highlight specific progresses: e.g. disappearing misconceptions. Tables 2 and 3 show the 5 most frequent snapshots produced by the groups for the Navigation and Loop exercises. Note that we omit the initial snapshot consisting only of the `start` block as it is present in every trajectory. We highlighted in bold the snapshots that lead to a successful goal program, i.e. the prefixes to a successful solution.

The pretest results of Tables 2 and 3 show that the 5 most frequent snapshots for both exercises present a common misconception: the confusion of the `turn right` block - which is a *relative* movement block that rotates the direction of the sprite 90 degrees clockwise, without changing its position - for an *absolute* movement block with the effect of moving the sprite to the right. Note that both exercises indeed require the sprite to initially move to the right (see Figure 1). The effects of the training show in the posttest results: the misconception snapshots disappear from the top 5 of the experimental group. Indeed the experimental group’s top 5 posttest snapshots for both exercises are all part of successful trajectories. The misconception snapshots remain in the waiting-list posttest top 5. However, two of the waiting-list group’s top pretest snapshots for the Navigation exercise, `st(r)t(r)t(r)` and `st(r)t(r)t(r)f`, disappear from the posttest results. This is interesting because these snapshots contain another error: a miscalculation of the distance

| Experimental group      |        |                               |        |
|-------------------------|--------|-------------------------------|--------|
| Pretest                 |        | Posttest                      |        |
| Snapshot                | %/14   | Snapshot                      | %/14   |
| <code>st(r)</code>      | 64.29% | <b><code>sf</code></b>        | 92.86% |
| <b><code>sf</code></b>  | 42.86% | <b><code>sff</code></b>       | 92.86% |
| <code>st(r)t(r)</code>  | 35.71% | <b><code>sfft(1)</code></b>   | 85.72% |
| <code>st(r)f</code>     | 28.57% | <b><code>sfft(1)ff</code></b> | 78.57% |
| <b><code>sff</code></b> | 28.57% | <b><code>sfft(1)f</code></b>  | 71.43% |

| Waiting-list group          |        |                          |        |
|-----------------------------|--------|--------------------------|--------|
| Pretest                     |        | Posttest                 |        |
| Snapshot                    | %/16   | Snapshot                 | %/16   |
| <code>st(r)</code>          | 81.25% | <code>st(r)</code>       | 81.25% |
| <code>st(r)t(r)</code>      | 62.50% | <code>st(r)t(r)</code>   | 75.00% |
| <code>st(r)t(r)t(r)</code>  | 43.75% | <code>st(r)t(r)f</code>  | 62.50% |
| <code>st(r)t(r)t(r)f</code> | 37.50% | <code>st(r)t(r)ff</code> | 62.50% |
| <code>st(r)t(r)f</code>     | 37.50% | <b><code>sf</code></b>   | 43.75% |

**Table 2: Most frequent snapshots for the Navigation exercise.**

| Experimental group         |        |  |        |
|----------------------------|--------|--|--------|
| Pretest                    |        | Posttest   |        |
| Snapshot                   | %/14   | Snapshot   | %/14   |
| <b><code>sf</code></b>     | 50.00% | <b><code>sr(0){}</code></b>                        | 78.57% |
| <code>st(r)</code>         | 42.86% | <b><code>sr(0){f}</code></b>                       | 71.43% |
| <code>st(r)t(r)</code>     | 42.86% | <b><code>sr(5){f}</code></b>                       | 71.43% |
| <code>st(r)t(r)t(r)</code> | 35.71% | <b><code>sr(5){f}t(r)</code></b>                   | 64.29% |
| <b><code>sff</code></b>    | 35.71% | <b><code>sr(5){f}t(r)r(5){f}t(r)r(5){f}</code></b> | 64.29% |

| Waiting-list group             |        |                             |        |
|--------------------------------|--------|-----------------------------|--------|
| Pretest                        |        | Posttest                    |        |
| Snapshot                       | %/16   | Snapshot                    | %/16   |
| <b><code>sf</code></b>         | 56.25% | <code>st(r)</code>          | 50.00% |
| <code>st(r)</code>             | 37.50% | <b><code>sf</code></b>      | 43.75% |
| <code>st(r)t(r)</code>         | 31.25% | <b><code>sr(0){}</code></b> | 31.25% |
| <code>st(r)t(r)t(r)</code>     | 31.25% | <code>st(r)t(r)</code>      | 31.25% |
| <code>st(r)t(r)t(r)t(r)</code> | 25.00% | <code>st(r)t(r)t(r)</code>  | 31.25% |

**Table 3: Most frequent snapshots for the Loop exercise.**

that the sprite has to cover to the right. The waiting-list group overcoming this distance miscalculation could be evidence of their natural cognitive development in the time interval between the pretest and the posttest.

### 4.3 Snapshot transitions & intervals

Figures 2, 3, 4 and 5 show graph representations of the students’ trajectories. Each node represents a code snapshot and each edge a code edit that transformed one snapshot into another. The size of the nodes is weighted on the average time interval the students spent on those particular snapshots. The size of the edges on how many students performed the relative code edit. Nodes are colored according to their AST distance from goal programs: yellow if closer, blue if further away. To reduce noise we only present nodes and edges traversed by more than one trajectory.

These visualizations can help capture at a glance the progress, or lack thereof, of a group. By looking at Figures 2 and 3, which

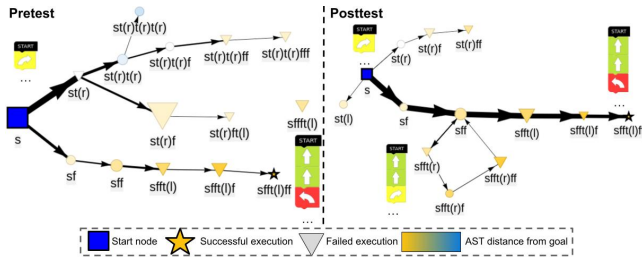


Figure 2: Pretest and posttest trajectories created by the experimental group for the Navigation exercise.

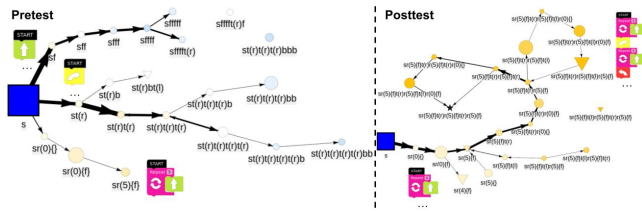


Figure 3: Pretest and posttest trajectories created by the experimental group for the Loop exercise.

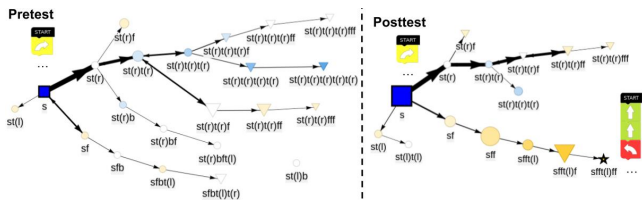


Figure 4: Pretest and posttest trajectories created by the waiting-list group for the Navigation exercise.

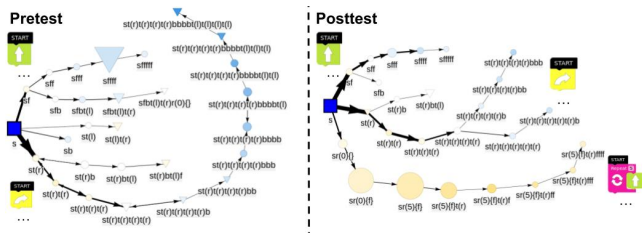


Figure 5: Pretest and posttest trajectories created by the waiting-list group for the Loop exercise.

represent the experimental group's results, we can clearly see how the most traversed trajectories change after the intervention. The pretest trajectories either go to areas of the graphs characterized by misconceptions, such as mistaking turn right for move right, or only partially overlap with trajectories that lead to correct solutions. The posttest trajectories, on the other hand, clearly lead towards the goals, represented by star-shaped nodes. Conversely, the waiting-list group most traversed trajectories remain largely the same between pre- and posttest. It is interesting to observe that

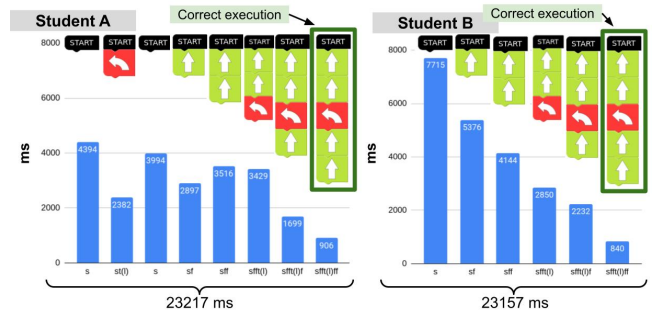


Figure 6: Two different trajectories that solved the Navigation exercise at the first attempt

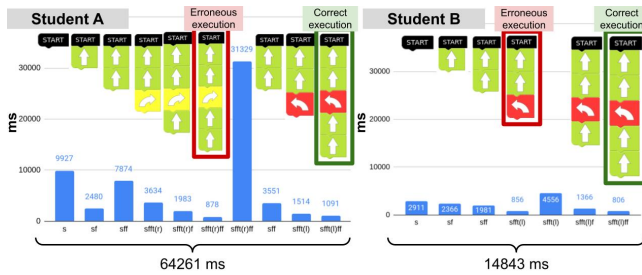
the snapshots leading to the goal in Figure 4 tend to be rather large, indicating more time, and possibly effort, spent on the trajectory.

It is interesting to observe the presence, or lack thereof, of multi-snapshot loops in the graphs: sub-trajectories that start and end on the same snapshot. These only appear in the posttest results of the experimental group. In Figure 2 we can see the loop starting from, and returning to, snapshot sff; in Figure 3 the loop from snapshot sr(5){f}t(r)5{f}. These loops represent common errors - in both cases the choice of an erroneous direction for a turn block - and the students' subsequent debugging processes. While on the pretest results (and, for the waiting-list group, also the posttest results) making errors while programming seems to make the students' trajectories "go off on their own", on the posttest results the errors seem to result only in transient deviations from a generally successful trajectory.

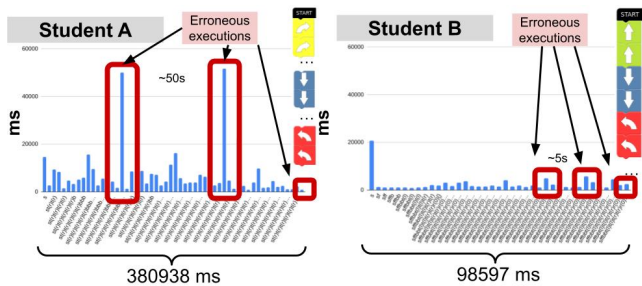
#### 4.4 Individual trajectories

We now present some examples of *individual* trajectories. We want to showcase how inspecting problem-solving processes at a granular level helps recognizing different strategies. Figures 6, 7 and 8 all compare two trajectories, created for the same exercise by two different students. We shall refer to the students as *Student A* and *Student B*, although the names may not be associated to the same people across the Figures. The histogram bars represent the interval of time each student spent on the corresponding snapshot. Executed snapshots are framed and labeled: those that solved the exercise in green, those that did not in red.

Figure 6 presents two trajectories produced by students who solved the Navigation exercise at the first attempt. Both students spent ~23 seconds on the task. However, their strategies are somewhat different. Student A's problem-solving process began with a misstep: connecting a turn left block to the start block. The student immediately corrected this misstep and proceeded to solve the exercise. The sequence of intervals is characterized by peaks and valleys, up until the last three snapshots: once the sfft(1) snapshot is reached, the last few steps are executed with apparent confidence, evidenced by the rapidly decreasing time spent on each snapshot. Student B, conversely, spent a longer time on the initial snapshot, ~7 seconds, and then proceeded to build a correct solution without missteps and with a monotonically decreasing sequence of intervals. Student B's trajectory suggests a careful planning phase, during which the student decided all the steps needed to reach the



**Figure 7: Two different trajectories that solved the Navigation exercise at the second attempt**



**Figure 8: Two different trajectories that did not solve the Loop exercise**

solution. The connection between planning and successful problem solving in coding is discussed in the literature, for examples see [1, 2, 19].

Figure 7 presents the strategies of two students who solved the Navigation exercise in two attempts. We see two very different processes and overall approaches. Student A: 1) produced a program that contained an erroneous turn right block; 2) spent a long time, ~30 seconds, without further edits after executing the program and seeing it did not solve the exercise; 3) quickly reached a correct solution after this long interval. This suggests a second planning phase, started after having received the execution feedback. Student B, instead, composed a solution very quickly, and apparently executed it halfway, while still incomplete. This could be due to lack of impulse control on the part of the student or it could have been caused by mistakenly pressing the "Execute" button.

Finally, Figure 8 presents the trajectories of two students who failed to solve the Loop exercise at the pretest. We avoid showing the block representation of each snapshot, because the trajectories are long and noisy. Student A's trajectory shows an effort towards solving the exercise, hindered by mistaking relative for absolute movement. It is quite apparent that the student actually attempted to solve the problem: the trajectory shows considerable time without code edits after each erroneous execution. This probably reflects time spent planning the next steps. Student B, conversely, quickly produced a long sequence of semi-random blocks. The inactive intervals after the first two erroneous executions are slightly longer than the other intervals in Student B's trajectory, but not comparable to those of Student A. It looks like this student either was very confused about the exercise or simply "gave up" on trying to

solve it. Student A was ultimately able to solve the exercise at the posttest, student B was not.

## 5 CONCLUSIONS

We presented and analyzed data consisting of fine-grained, edit-level snapshots collected during programming processes. To conclude this work, we now discuss how this information could be of value both to educators and learners in real-life learning contexts.

We recognize that one limitation of this work is the fact that our results reflect the programming strategies of only 2 class groups. This is caused by the experimental nature of our analysis and this work can be the foundation for larger future studies.

We analyzed the most frequent snapshots produced by students during their problem-solving processes (cf. Tables 2 and 3). These showed how certain apparent misconceptions, such as mistaking relative "turn-left" instructions for absolute "move-left" instructions, tend to disappear after learners gain experience by interacting with programming tasks in similar contexts. Arguably, embarking on a programming trajectory consisting of "misconception snapshots" after a training could represent a warning sign. Detecting it could trigger an automated response from the system, e.g. a review of the functions of the available code blocks. Feedback presented "just-in-time" is generally more useful, as it helps learners carry in their working memory the information they need to solve a problem while dealing with it [24]. This early warning could also be relayed to educators and satisfy requirement number 3 discussed in [23].

The graph representations of the programming trajectories of the students, presented in Section 4.3, capture interesting pictures of the general programming proficiency of groups of learners. The most frequently traversed edges highlight common actions performed by the learners. Loops in the graphs represent debugging processes. The size of nodes reflects the average time spent on particular snapshots, possibly indicating moments of doubt. In general, these visualization address requirement number 4 discussed in [23]. Educators could benefit from them: knowing that some snapshots make learners pause longer than average could single out over-complex aspects of exercises or difficulties shared by the whole class group. Moreover, the rate of edges entering and exiting areas of the graph characterized by misconceptions shows "warning" nodes and how learners are able to debug erroneous programs.

Finally, we analyzed some individual learners' trajectories, with a focus on how they highlight differences in approaches that other metrics would equate. Effenberger and Pelánek, in [7], discuss which metrics show more potential in assessing students' performance in programming tasks similar to those used in this work. Their interest, and ours, is not in grading the students but in assessing competencies in order to adapt learning activities to students. They find that number of executions and time spent on task are two automatically computable metrics that tend to correlate with human evaluations. In Section 4.4 we showed how analyzing individual trajectories highlights differences between students who required the same number of attempts to solve an exercise, in one case even requiring a very similar time. This added information could help in reaching finer levels of tuning learning activities to student competencies and learning goals.

## REFERENCES

- [1] Barbara Arfé, Tullio Vardanega, Chiara Montuori, and Marta Lavanga. 2019. Coding in primary grades boosts children's executive functions. *Frontiers in psychology* 10 (2019), 2713.
- [2] Barbara Arfé, Tullio Vardanega, and Lucia Ronconi. 2020. The effects of coding on children's planning and inhibition skills. *Computers & Education* 148 (2020), 103807.
- [3] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable programming: blocks and beyond. *Commun. ACM* 60, 6 (2017), 72–80.
- [4] Miles Berry. 2013. Computing in the national curriculum: A guide for primary teachers. (2013).
- [5] Isabella Corradini, Michael Lodi, and Enrico Nardelli. 2017. Computational Thinking in Italian Schools: Quantitative Data and Teachers' Sentiment Analysis after Two Years of "Programma il Futuro". In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) (ITiCSE '17). Association for Computing Machinery, New York, NY, USA, 224–229. <https://doi.org/10.1145/3059009.3059040>
- [6] Peter J Denning and Paul S Rosenbloom. 2009. the profession of it Computing: the Fourth great domain of science. *Commun. ACM* 52, 9 (2009), 27–29.
- [7] Tomáš Effenberger and Radek Pelánek. 2019. Measuring Students' Performance on Programming Tasks. In *Proceedings of the Sixth (2019) ACM Conference on Learning @ Scale* (Chicago, IL, USA) (L@S '19). Association for Computing Machinery, New York, NY, USA, Article 26, 4 pages. <https://doi.org/10.1145/3330430.3333639>
- [8] Daniel Amo Filvà, Marc Alier Forment, Francisco José García-Peñalvo, David Fonseca Escudero, and María José Casañ. 2019. Clickstream for learning analytics to assess students' behavior with Scratch. *Future Generation Computer Systems* 93 (2019), 673–686.
- [9] Barry Gribbons and Joan Herman. 1996. True and quasi-experimental designs. *Practical assessment, research, and evaluation* 5, 1 (1996), 14.
- [10] Ting-Chia Hsu, Shao-Chen Chang, and Yu-Ting Hung. 2018. How to learn and how to teach computational thinking: Suggestions based on a review of the literature. *Computers & Education* 126 (2018), 296–310.
- [11] Bo Jiang, Wei Zhao, Nuan Zhang, and Feiyue Qiu. 2022. Programming trajectories analytics in block-based programming language learning. *Interactive Learning Environments* 30, 1 (2022), 113–126.
- [12] Ioanna Kanellopoulou, Pablo Garaizar, and Mariluz Guenaga. 2021. First steps towards automatically defining the difficulty of maze-based programming challenges. *IEEE Access* 9 (2021), 64211–64223.
- [13] Max Kesselbacher and Andreas Bollin. 2019. Discriminating Programming Strategies in Scratch: Making the Difference between Novice and Experienced Programmers. In *Proceedings of the 14th Workshop in Primary and Secondary Computing Education* (Glasgow, Scotland, UK) (WiPSCE '19). Association for Computing Machinery, New York, NY, USA, Article 20, 10 pages. <https://doi.org/10.1145/3361721.3361727>
- [14] Seymour Papert. 1980. *Mindstorms: Computers, children, and powerful ideas*. NY: Basic Books 255 (1980).
- [15] Radek Pelánek and Tomáš Effenberger. 2022. Design and analysis of microworlds and puzzles for block-based programming. *Computer Science Education* 32, 1 (2022), 66–104.
- [16] Radek Pelánek and Tomáš Effenberger. 2023. The landscape of computational thinking problems for practice and assessment. *ACM Transactions on Computing Education* 23, 2 (2023), 1–29.
- [17] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. 2015. Autonomously Generating Hints by Inferring Problem Solving Policies. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale* (Vancouver, BC, Canada) (L@S '15). Association for Computing Machinery, New York, NY, USA, 195–204. <https://doi.org/10.1145/2724660.2724668>
- [18] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.
- [19] Judy Robertson, Stuart Gray, Martin Toye, and Josephine Booth. 2020. The relationship between executive functions and computational thinking. *International Journal of Computer Science Education in Schools* 3, 4 (2020), 35–49.
- [20] Robert Rosenthal, Harris Cooper, Larry Hedges, et al. 1994. Parametric measures of effect size. *The handbook of research synthesis* 621, 2 (1994), 231–244.
- [21] Valerie J Shute, Chen Sun, and Jodi Asbell-Clarke. 2017. Demystifying computational thinking. *Educational research review* 22 (2017), 142–158.
- [22] Xiaodan Tang, Yue Yin, Qiao Lin, Roxana Hadad, and Xiaoming Zhai. 2020. Assessing computational thinking: A systematic review of empirical studies. *Computers & Education* 148 (2020), 103798.
- [23] Sean Tsung, Huan Wei, Haotian Li, Yong Wang, Meng Xia, and Huamin Qu. 2022. BlockLens: Visual Analytics of Student Coding Behaviors in Block-Based Programming Environments. In *Proceedings of the Ninth ACM Conference on Learning @ Scale* (New York City, NY, USA) (L@S '22). Association for Computing Machinery, New York, NY, USA, 299–303. <https://doi.org/10.1145/3491140.3528298>
- [24] Jeroen JG Van Merriënboer. 2013. Perspectives on problem solving and instruction. *Computers & Education* 64 (2013), 153–160.
- [25] Kaizhong Zhang and Dennis Shasha. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing* 18, 6 (1989), 1245–1262.
- [26] Yingbin Zhang, Luc Paquette, Juan D Pinto, Qianhui Liu, and Aysa Xuemo Fan. 2023. Combining latent profile analysis and programming traces to understand novices' differences in debugging. *Education and Information Technologies* 28, 4 (2023), 4673–4701.