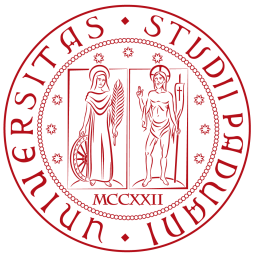

University of Padua

Department of Mathematics "Tullio Levi-Civita"

Master's degree
in Computer Science

Co.Thi.: using Learning Analytics to measure the acquisition of Computational Thinking skills

Supervisor:
Prof. Tullio Vardanega

Student:
Gabriele Pozzan
1192120

Academic year 2021/2022

Abstract

Computational Thinking skills, such as abstraction, debugging, decomposition, generalization and algorithmic thinking, are central assets of the body of knowledge of Informatics, and have general-purpose potential to benefit individuals from all walks of life. Recent literature suggests that: (1) students can acquire those skills since early age; (2) their learning associates positively with motivation to carry on learning Informatics in higher education; (3) these skills correlate positively with cognitive development, in particular with Executive Functioning. Learning Analytics is the practice of collecting ethically-cleared data about students' learning activities, to better understand and improve the learning process. This work leverages Learning Analytics to further the study of the correlation between Computational Thinking and Executive Functioning, and collect deeper and wider quantitative evidence in favor of the argument that Informatics should be taught in schools throughout all of K-12 curricula.

Contents

List of Figures

List of Tables

Chapter 1

Contextual background

This Chapter describes the background of the present work by first stating the problems it seeks to address and then presenting a deep-dive in the state of the art of the relevant literature.

1.1 Introduction

Computation is ubiquitous in present-day society and its relevance to people from all walks of life is tied not only to its practical applications (e.g. the fact that many people interact daily with multiple computers, that algorithms dictate what they find when they search for information online, etc.) but also to its fundamental concepts (which can be lenses through which humans may gain new insights on observed phenomena). Denning and Rosenbloom, in [1], argue that computation should be seen as a fourth fundamental pillar of science (in addition to physical, life and social sciences) because it shares with the others three main characteristics: (1) a distinctive focus that is also relevant to the other domains (computation and information processes), (2) distinctive subdomains in constant interaction with each other (computer science, computer engineering, information technology, etc.) and (3) wide impact on all parts of life (as stated before in this paragraph).

The term Computational Thinking (CT) refers to the set of skills which generalize principles and methods of the Informatics body of knowledge. The general-purpose nature of these skills makes them a good fit for introducing concepts and learning activities tied to Informatics in general education. As such, recent research (cfr. Section 1.4.2) has focused its attention on the effects that CT activities have on cognitive development and shown that they can have positive

impact, in particular on Executive Function (EF) skills such as planning, response inhibition, etc. (for detailed definitions cfr. Section 1.4.1).

The present work builds on the foundations laid by this research (in particular, [2] and [3]) and seeks to advance it with a focus on data collection about CT activities' *processes* and *outcomes*. Human-machine interactions can be monitored to collect a wealth of fine-grained data (e.g. clicks, drag and drop gestures, time intervals, etc.); moreover, digital artifacts (e.g. scripts produced during programming activities) are machine-readable objects, fit for automated analysis.

Therefore, the first problem we seek to address in this work is: how can we augment and automate our data collection techniques in order to expand their depth (in terms of granularity) and scope (i.e. scalability afforded by automation)? We do so by leveraging concepts and techniques belonging to the research field of Learning Analytics (LA, cfr. Section 1.5).

Data, by itself, is just a collection of bits stored in a database and it would be meaningless without interpretation. Thus, the second problem addressed by this work is: what *new information* can we extract from this fine-grained data on CT activities? We do this by combining it with results obtained by standardized cognitive tests (cfr. Section 1.4.3) and looking for interesting correlations (see Chapter 3 for a discussion of our results).

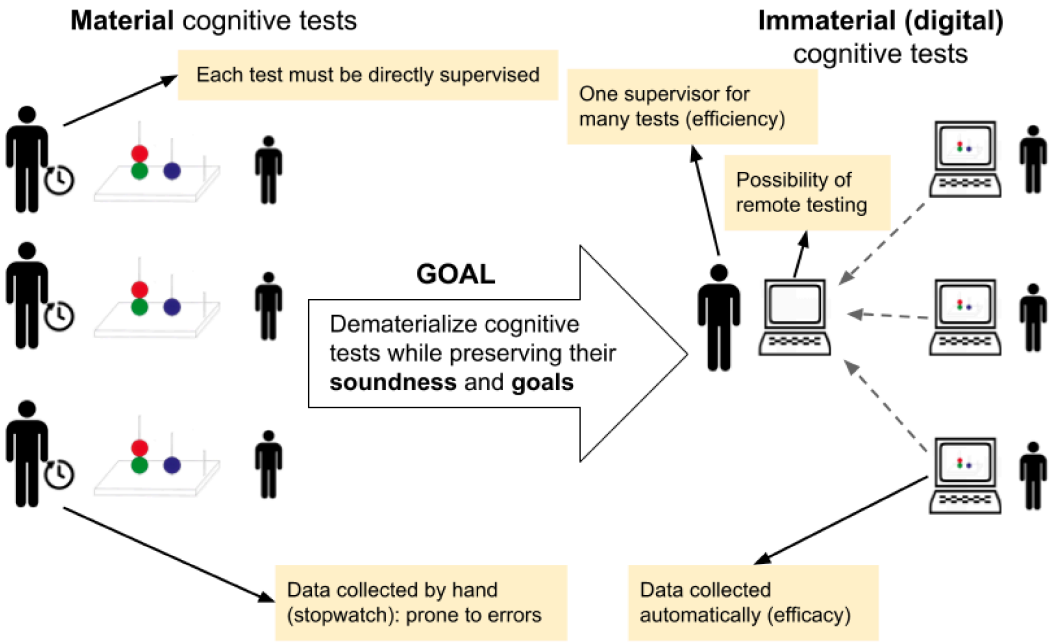

Figure 1.1: Perspective benefits of cognitive tests digitization

This interdisciplinarity suggests perspective objectives which represent some

of the long-term goals of this research: the cognitive tests we use are based on methodologies which predate the massive expansion of computing started during the second half of the 20th century. As such, they usually involve activities which require manual interaction and the co-presence of a researcher/instructor (who is also often tasked with the data collection e.g. by timing the subjects with a chronometer). Arguably, the digitization of these activities could improve both their *efficiency* (e.g. by leveraging automated data collection to change the researcher-subject relationship from one-to-one to one-to-many) and *efficacy* (removing human errors caused by mistakes and delayed reaction times). Figure 1.1 summarizes these arguments.

Of course, this is an open perspective which will require careful work (with a particular focus on standardization and reliability testing) and collaboration with experts in cognitive sciences.

1.2 Computational Thinking

This Section presents a summary of recent literature concerning CT. It starts by reviewing various definitions of CT and by giving a picture of the state of CT activities conducted in Italian schools. Later, it enumerates arguments for the importance of teaching Informatics throughout all grades (starting from primary school) and finally, it presents the constituents of CT and some techniques used for its assessment.

1.2.1 CT definitions

The term *Computational Thinking* was first used by Seymour Papert in his book *Mindstorms: Children, Computers and Powerful Ideas* ([4]) and later popularized by Jeannette M. Wing ([5]) to describe a set of skills and problem solving methods which have their roots in the development of Computer Science (CS) during the 20th century.

Román-González et. al, in [6], discuss how there is little consensus on a precise definition of the term and propose a categorization of definitions in three sets: (1) generic definitions, (2) operational definitions and (3) educational and curricular definitions. Table 1.1 presents examples of different definitions cited by the authors.

Enrico Nardelli, in [11], stresses the central role of the *information-processing agent* in the definition coined by Cuny, Snyder and Wing and reported in the

Definition	Category
CT “is the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent” Cuny, Snyder and Wing, [7]	Generic
CT is "the thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms" Alfred V. Aho, [8]	Generic
CT is a “problem-solving process that includes (but is not limited to) the following characteristics: formulating problems in a way that enables us to use a computer and other tools to help solve them; logically organizing and analyzing data; representing data through abstractions such as models and simulations; automating solutions through algorithmic thinking (a series of ordered steps); identifying, analyzing, and implementing possible solutions with the goal of achieving the most efficient and effective combination of steps and resources; generalizing and transferring this problem solving process to a wide variety of problems” Computer Science Teachers Association (CSTA) & International Society for Technology in Education (ISTE), [9]	Operational
CT can be viewed as a set of computational concepts (sequences, loops, parallelism, events, conditionals, operators and data), computational practices (incremental, iterative development, testing, debugging, reusing, remixing, abstracting, modularizing) and computational perspectives (expressing, connecting, questioning). Summarized from Brennan & Resnick [10]	Educational

Table 1.1: CT definitions divided by category

first row of Table 1.1: the fundamental shift is "from solving problems to having problems solved"; moreover, he generalizes the concept of "solutions to problems" (which can be misleading by driving people to only think about well-defined, abstract problems) to "modeling a situation and specifying the ways an information-processing agent can effectively operate within it to reach an externally specified goal"; this broadens the definition to include complex tasks such as simulations and processes which should impact the real world.

Another way to look for a definition of CT considers its relationship with other disciplines. Wing, in [12], observes that, while CT intersects with both Engineering and Mathematics (in particular, they all share the focus on *problem solving*), it is distinct from them because:

- It is more constrained than Mathematics: the systems which are the product of CT run on physical (thus, limited) computing agents and working within these limitations is a fundamental aspect of CT.

- It is less constrained than Engineering in the sense that virtual simulations and worlds are not constrained by the physical world (at least until we reach the limits of computation discussed in the previous point).

Shute, et. al, in [13] discuss the relationship CT *skills* have with CS and programming from the point of view of their *scope*:

- Programming is an *act*, it means solving specific problems and is one of the benefits of being able to think computationally. Arfé and Vardanega, in [14] corroborate this idea by stating that programming is a tool of CT just like the knowledge of numbers is a tool of arithmetic.

- CS is interested in the theory of computation and information while CT skills are transferable to other domains (and thus, more general). This concept is also cited by a report of the Royal Society (United Kingdom's national academy of sciences) which states that CT "is the process of recognising aspects of computation in the world that surrounds us, and applying tools and techniques from Computer Science to understand and reason about both natural and artificial systems and processes" ([15]).

Figure 1.2 presents a summary of the distinctions made thus far.

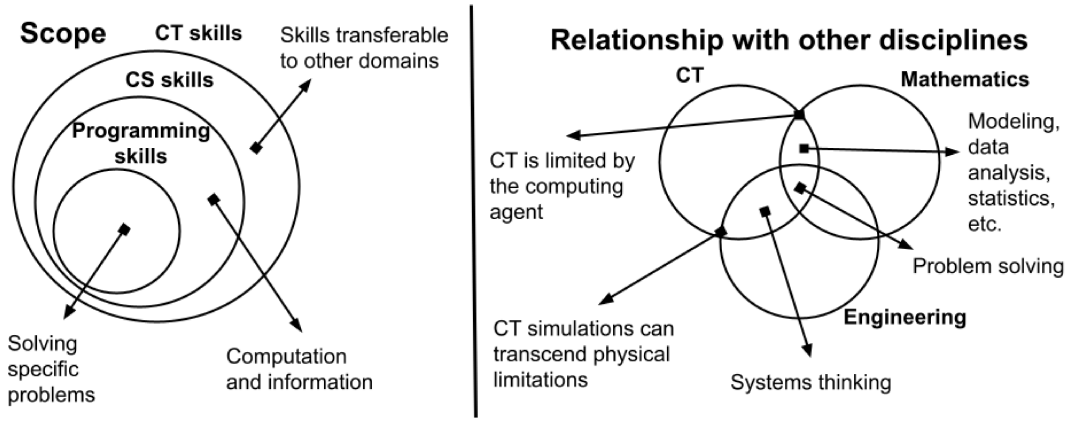

Figure 1.2: Summary of CT relationships with other disciplines

To conclude, we summarize some points made by Nardelli ([11]), who critically approaches the subject of CT and reaches the conclusion that the most important thing about it is its *goal*:

- CT is not *a new way of thinking*, it's just a useful shorthand which helps keeping the focus on *principles* and *methods* (and not on systems and tools);

- the main goal of (studying and defining) CT is arguing for *the value of teaching Informatics* to all students (starting from an early age) and not just future engineers and computer scientists.

The focus on the goal of spreading the teaching of Informatics in schools is particularly important at this time in Italy, as we argue in the next few Sections.

To see what skills constitute CT and how they can be useful *to all students* cfr. Section 1.2.4.

1.2.2 CT in Italian schools

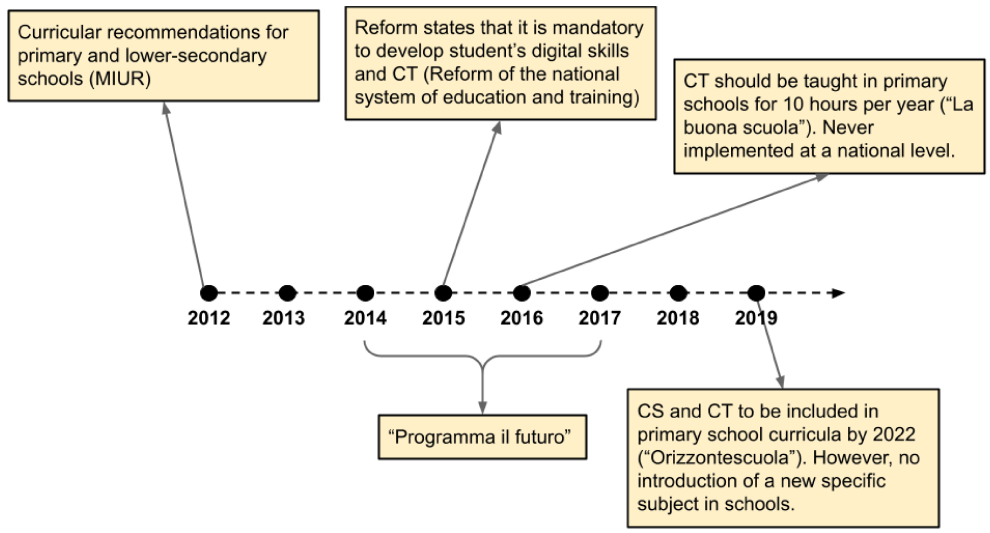

Figure 1.3: Timeline of some CT related initiatives and proposals in Italy (sources: [16, 17])

Michael Lodi, in his doctoral thesis ([16]), paints a picture of the state of CS education in Italian curricula.

Topics and subjects related to computing are divided in two areas: (1) digital competence (familiarity with Information and Communication Technologies, ICT, in work and life related situations); (2) the subject of *Technology* for primary and lower-secondary school and *Informatics* (or, in some cases, more specific CS subjects) for some upper-secondary schools.

The author points out that the national recommendations encourage the introduction of programming languages at the primary and lower-secondary level, but that their enactment is ultimately left to the responsibility of each school. Moreover, because Informatics is not part of the national curriculum for primary

schools, future teachers studying for the Primary Teacher Education degree are not trained in this subject.

Nesen et. al, in [17], cite the aforementioned thesis and argue that a lack of exposure to CS in primary schools can lead to low interest in the field among students, even though the subject is available later in K-12 curricula (e.g. in technical schools at the upper-secondary level).

Both works observe that initiatives like "Programma il futuro"[1] represent a first step towards the diffusion of CT at the lower school levels but that the educational system still lacks concrete reforms aimed at the introduction of Informatics in primary and lower-secondary schools.

Figure 1.3 presents a timeline of recent initiatives and proposals related to CT in Italy: it is clear that there is some interest around the topic. However, the ultimate goal of introducing Informatics as a full-fledged school subject is still far from concretization.

1.2.3 The importance of teaching Informatics

Having reported critically (in the previous Section) on the state of CT activities and CS education in Italy, it is important to answer the question: why is teaching Informatics to primary and lower-secondary school students important?

The literature gives a variety of responses, which we summarize here:

- Corradini et. al, in [18], state that (1) understanding the inner workings of technologies allow people to become more informed citizens (e.g. by fully understanding the consequences of their actions in the digital space); (2) there is a need for workers qualified in digital skills; (3) programming offers a means for strengthening knowledge in other subjects (e.g. to build a physics simulation students must fully understand the concepts before expressing them in a processable way); (4) Informatics is a way to learn problem solving.

- Bau et. al, in [19], again argue for the importance of developing digital expertise for the workforce but also observe that programming can be useful for reaching "other goals" (e.g. self expression).

- Resnick et. al, in [20], discuss how young people, who are often referred to as "digital natives", usually have the ability to interact with *already existing*

[1] `https://programmailfuturo.it/`

technologies (mostly social networks, online games, web browsing, etc.) and that this does not automatically translate to the capability of designing and creating new digital applications (or understanding the inner workings of existing ones). We could rephrase this by saying that learning Informatics allows people to become *creators* of digital technologies, rather than just *consumers*.

- Denning and Rosenbloom, in [1], argue that computing (i.e. CS, or Informatics) shares important characteristics with the other three main domains of science (physical, life and social) and that it should be considered as the fourth (cfr. also Section 1.1).

Two of the four articles mentioned here state the importance of learning Informatics to prepare for high-demand jobs and careers. However, since the present work focuses mostly on young students, this motivation should not be taken for granted: Duncan et. al, in [21] cite (understandable) critiques of this focus on *preparing for work* (reporting that probably we should leave children to enjoy their childhood); on the other hand, they highlight (1) the importance of the general-purpose *principles* and *concepts* of CT as opposed to specific programming languages and skills (this is one of the reasons why CT activities are used as a conduit to Informatics) and (2) the fact that young students learn quickly and can thus develop good attitudes towards programming and Informatics (this is similar to what Nesen et. al, report in [17]).

Arguably, the amount of freely available tools and information on the internet has reached an all-time high in recent years: for an example, students with a grasp of programming (and basic knowledge of the English language) can access high-quality game engines (e.g. Godot[2], Unity[3]) and learn how to use them for free (e.g. by following youtube tutorials); this opens up great possibilities for self expression.

Moreover, as Caspersen et al. discuss in [22]: widespread Informatics education would support research in all sectors and provide fertile ground for (positive) contamination with other subjects. They highlight the importance of *going beyond* simple digital literacy and the fact that knowledge in this area is more and more necessary to navigate the digital space as informed participants.

[2] `https://godotengine.org/`
[3] `https://unity.com/`

1.2.4 CT components

We now review various breakdowns of the components of CT as proposed in the literature, and then some examples of the general-purpose aspects of these skills.

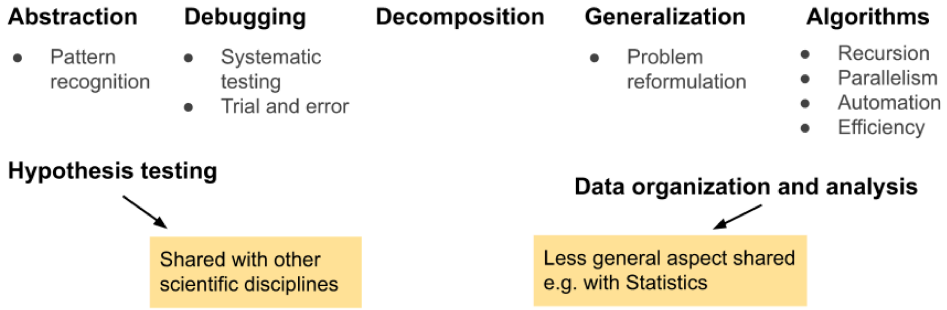

Figure 1.4: CT components and subcategories cited in [13]

Shute et. al, in [13], cite various works which attempt to list the components of CT (summarized in Table 1.2). Figure 1.4 groups the different subcategories mentioned in the articles under a set of major components.

CT component	**Mentioned in**
Abstraction	[5, 23, 24, 25, 26]
Debugging	[5, 24, 25, 26]
Decomposition	[5, 26]
Generalization	[5, 23, 24]
Algorithms	[5, 23, 25, 26]
Hypothesis testing	[25]
Data organization and analysis	[23, 25]

Table 1.2: CT components according to research cited in [13]

These skills clearly represent the daily bread of CS engineers and scholars. However, as mentioned in Section 1.2.1, their usefulness is not limited to this professional group; to give some examples:

- Problem **decomposition** means being able to adopt a *divide et impera* problem solving approach: writing a musical composition may seem like a daunting task, but starting with a melody and then adding chord progressions, harmonies and other instruments simplifies the process (this is also an example of an **iterative** approach).
- **Debugging** means retracing one's steps in a systematic way to understand what went wrong: a baker may realize that their loaves of bread did not

rise because they forgot one passage of dough preparation (namely, burying the yeast in the flour to make sure it does not enter in contact with the salt).

- Familiarity with **algorithms** may encourage people to give precise instructions in any context (e.g. by saying how much flour to use for a recipe in grams instead of cups, which could be of any size).

Of course, the usefulness of CT skills translates to other professional areas as well; to make an example: Nicole D. Anderson, in [26] argues that CT can greatly benefit psychology students (e.g. algorithmic skills could help in designing efficient research plans).

1.2.5 CT assessment techniques

Tang et. al, in a recent review ([27]), list four tools for CT skills assessment found in the literature:

- *Selected- or constructed-response tests*: "traditional" tests based on multiple choice or open-ended questions. Grover and Basu, in [28], show an example of this technique: students are presented with multiple-choice (e.g. select a word based on some logical constraints) and opend-ended questions (e.g. explain how a variable changes during the execution of a loop). For other examples cfr. [29].

- *Portfolio assessment*: analysis of student's artifacts (programs, games, etc.); these are mostly evaluated using grading rubrics. This technique is popular for studies based on Scratch (this is not surprising, as it is not easy to quantify the learning outcomes of open-ended activities). This is the most interesting category for this thesis. For a summary of techniques found in the literature cfr. Section 1.5.2.

- *Surveys*: often used to evaluate motivation and attitudes towards CT activities.

- *Interviews*: this technique can be very powerful (e.g. asking students to verbalize their problem-solving process can show how deeply they understand CT concepts) but is very costly (indeed, the authors report that it is underutilized).

1.3 Block-based programming languages

In this work, we are interested in the use of block-based programming languages to expose primary-school children to CT: these languages are united by the use of puzzle-piece-like *blocks* to condense programming primitives ([30]). The research field interested in the development and study of block-based language (mostly for educational purposes) is relatively young and gathering some interest, as reported in [19].

1.3.1 Advantages of block-based programming

Block-based programming languages are often explicitly created for younger audiences. For this reason, they tend to present environments and exercises designed to be engaging for children (cfr. Code.org's exercises, which use characters from popular video games like Angry Birds and Plants vs. Zombies). Thus, the most visible advantage they offer is *improving students' motivation.*

Other interesting advantages reported in the literature are: the prevention of syntactic errors (blocks are self-contained functions which can only be connected with other compatible blocks, there are no compile-time errors) and the reduction of the cognitive load of programming (it is easier to recognize an instruction by color and shape than it is to remember its keyword, parameter list, etc.).

Table 1.3 presents a summary of the benefits of block-based programming languages reported in the literature.

One of the advantages which is not explicitly discussed (but is obviously implied in many works) is the fact that block-based languages are usually presented *inside a programming environment* which manages and abstracts away many low-level intricacies in order to give the best possible feedback on program execution (e.g. Scratch handles the logic which connects users' instructions to the rendering of their games on the screen, Code.org translates users' blocks into programs under the hood and then executes them to retrieve input instructions for their maze games, etc.).

1.3.2 Disadvantages of block-based programming

Despite the advantages presented in the previous Section, block-based programming languages can, of course, have their downsides.

Various studies point out how students can take them less seriously, as they are not seen as "real world programming".

Advantage	Notes	Mentioned in
Simplified syntax	Blocks prevent the occurrence of syntactic errors	[31, 32, 33, 34, 35, 21, 19]
Drag and drop	Reduces the interface friction of having to use the keyboard (and, with touch-screens, also the mouse)	[31, 34, 35]
Immediate execution (no compile time)	A weaker point, this is also true for scripting languages e.g. Python	[31]
Metaphors	Instructions for the processor become "giving directions to the sprite", ecc.	[31]
Motivation	Block-based languages are less intimidating	[32, 36]
Puzzle-piece-like blocks	Color and shape help see which commands can be combined	[34, 35]
Recognition vs. recall	Recognizing blocks presented in a list is easier than remembering keywords and primitives (block-based languages reduce the cognitive load of programming)	[34, 35, 19]
Abstraction	Code blocks work as functions and can abstract low-level instructions	[34, 35, 19]
No compile-time errors	Blocks can not be combined if they do not work together	[21]

Table 1.3: Advantages of block-based programming languages found in the literature

Another interesting critique is related to the fact that these languages are less efficient in the way they use screen space (blocks quickly consume vertical space, complex programs soon become unreadable) and in the way they are programmed (dragging a block takes more time than typing, refactoring a block structure can be more complex than simply changing a variable name).

Finally, some authors question the learning outcomes of activities with block-based programming languages: it is not clear if programming skills and understanding of fundamental concepts transfer to other languages. Moreover, block-based languages can actively encourage the formation of bad programming habits ([37]).

Table 1.4 presents a summary of the disadvantages of block-based programming languages reported in the literature.

Disadvantage	Notes	Mentioned in
Not "real programming"	Some studies report this sentiment among the subjects (usually older students)	[38, 34, 35, 39]
Less efficient	Blocks take up more space (programs get longer and more difficult to read), drag-and-drop is inefficient compared to typing	[34, 35, 19]
Misconceptions	The puzzle-piece metaphor can lead students to think that code blocks have a *specific place* and not a variety of uses	[35]
Less expressive / less powerful	Blocks condense instructions but there can not be a block for every function (by abstracting low-level instructions we lose expressiveness)	[34, 35]
Transferability issues	Competencies gained in block-based languages do not directly translate to "traditional" languages	[38, 39]
Incomplete understanding of operators	Abstracting the intricacies of some concepts (variables, loops, boolean logic) leads to less understanding of the same	[28]
Bad programming habits	Extreme bottom-up or top-down approaches which highlight the lack of a design phase	[37]

Table 1.4: Disadvantages of block-based programming languages found in the literature

1.3.3 Distinctions within block-based languages

This Section presents the most important distinction we see in block-based programming languages, for the purposes of this study: the distinction between works using block-based languages in "open-ended" projects and exercises (mostly based on Scratch[4], cfr. Figure 1.5) and works based on curated sets of exercises with fixed solutions (e.g. Code.org).

The first category stems from the philosophy of *constructionism*, originated by Seymour Papert, which asserts the value of learning difficult concepts by building projects ("learning through making", cfr. [40]) with further benefits gained from the emphasis on creativity (the reason why projects are often "open-ended") and,

[4] `https://scratch.mit.edu/`

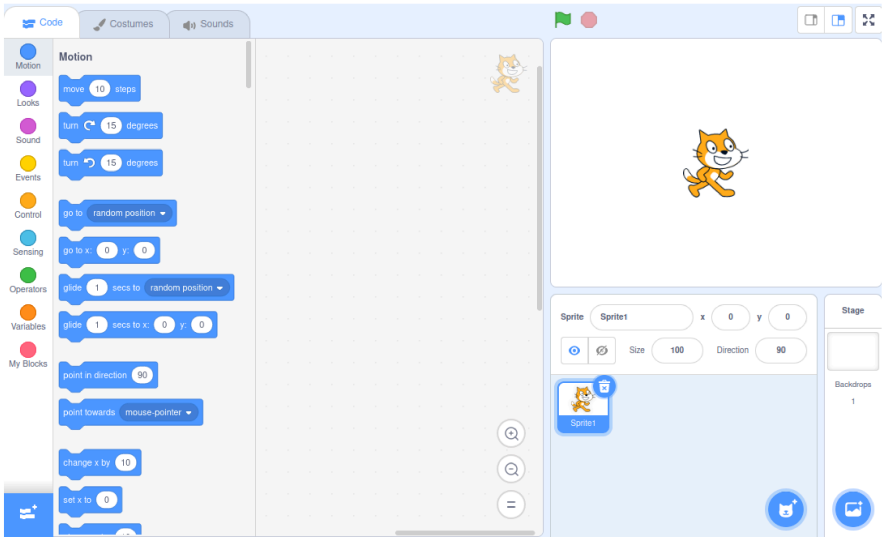

Figure 1.5: A screen capture of the web interface of Scratch

sometimes, collaborative work (be it via group projects or feedback from peers).

The second category (cfr. Figure 1.6) is based on more structured exercises often inspired by classic *microworlds* (cfr. [41]) like *Karel the Robot* (cfr. [42] and, for a practical example, the "Maze" exercises from Code.org) and *Turtle graphics* (cfr. [4, 43] and the "Artist" exercises from Code.org). In these works, experts divide the exercises in sets, based on their complexity, and introduce new concepts (e.g. loops and conditionals) gradually.

Both these categories have their advantages and disadvantages: intuitively, open-ended projects (often with the goal of making a game), with great focus on creativity and group work, can be very engaging for most students (for a work discussing the motivation of students engaged in a Game Jam cfr. [44]); on the other hand, the variety and complexity of tools and concepts require that students digest them gradually. Moreover, it's not easy to quantify the learning outcomes of open-ended activities: this may be the reason why many works concentrate on this goal (for an example cfr. Dr. Scratch [45]).

Structured exercises (e.g. "Maze" and "Artist" from Code.org) offer less freedom of expression and can be seen as more akin to traditional classroom-based activities; however, they can still be engaging as they are, all in all, puzzle games (in particular we can see similarities with the design of famous puzzle games in the way concepts are introduced in isolation first and then in gradually more complex situations, cfr. [46]). Moreover, they can be designed in a way that makes it easier to understand the learning levels of students (e.g. mastery of a complex concept, such as loops, could be deduced by observing their use in situations in

which they are not explicitly required, but would offer the optimal result). Exercises built with a set difficulty can also be used for interesting purposes, like adapting learning paths to students, be it via human-designed exercises extracted from a pool with similar complexity ([47]) or via exercises generated in a procedural way, parameterized by the difficulty level ([48]); for further discussions about this concept see Section 4.2.3.

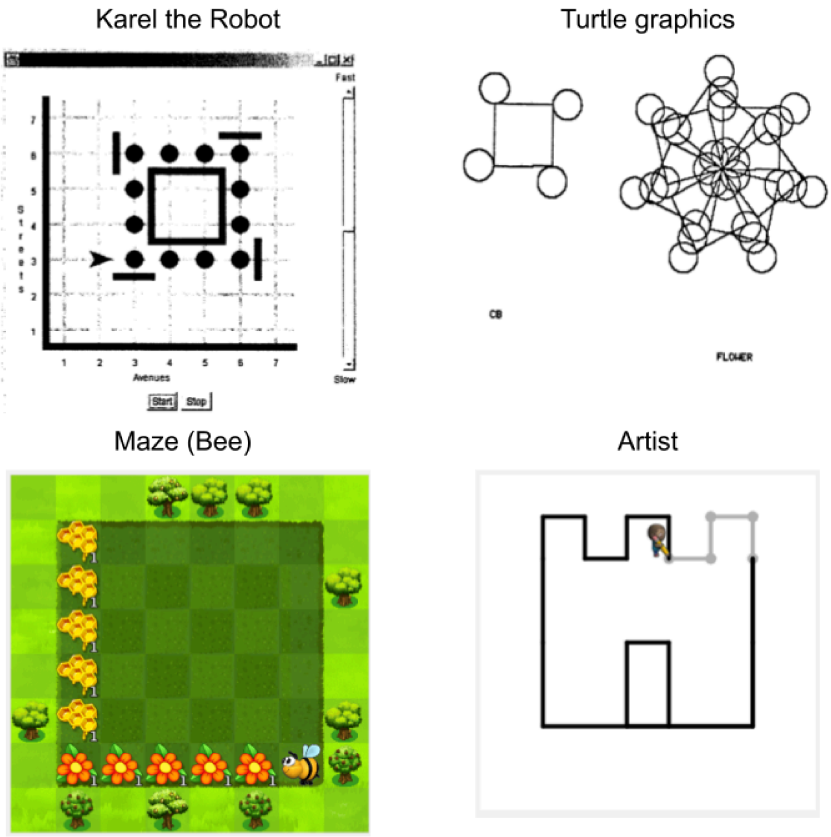

Figure 1.6: Examples of microworld and turtle graphics exercises, with their modern counterparts (sources: [43, 42], and Code.org)

1.4 Executive Functions

This Section presents a brief overview of *Executive Function* (EF) skills and of works which connect them to CT. Finally, it describes the Tower of London (TOL) cognitive test (which is part of our experimental evaluation, cfr. Chapter 3).

1.4.1 EF definition

EF are a set of cognitive skills with a regulatory purpose, i.e. they are used to control one's thinking and behaviour ([49]). Gilbert and Burgess, in [50], draw a distinction between *automatic* and *controlled* mental operations: automatic operations are those well-trained, well-explored actions (and thoughts) which humans can perform virtually automatically (they use "reading a word" as an example); controlled operations, on the other hand, are those that become necessary when new and unexpected situations arise and no automatic response is available (this also encompasses errors and situations in which there is a mismatch between context and automatic response). Thus, EF are the skills which enable "lower-level" mental processes (linked to automatic behaviour) to be controlled by "higher-level" ones (tied to controlled behaviour).

Best and Miller, in [51], review various articles on EF from a developmental point of view; the points of interest for the present study are relative to the temporal development of these skills:

- EF skills development starts very early (their foundations emerge during the 1st year of life) and continues until adolescence/early adulthood.
- EF skills develop at a fast rate during preschool and early school years (cfr. also [3]).

Adele Diamond, in [52], describes *core* and *higher-order* EFs; the core skills are:

- **Inhibition and interference control**: the ability to control impulses and automatic responses, to break habit and exit "autopilot" mode; applied to one's attention, this means also choosing to concentrate on what is important for the task at hand and avoiding external (irrelevant) stimuli; inhibition is also tied to motivation (e.g. resisting the temptation to give up during a difficult task) and the ability to pursue delayed gratification (necessary for long and complex tasks e.g. writing a Master's thesis).
- **Working memory**: the ability to retain information and combine it with incoming data (e.g. sensory stimuli) which is fundamental for any complex multi-step task (be it understanding a paragraph, solving a math problem, building an algorithm).

- **Cognitive flexibility**: "thinking outside the box" (changing perspective on known situations), recognizing opportunities and errors; also, the ability to change point of view, both "spatially" (visualize a scene from a different angle and position) and "personally" (seeing a situation from someone else's perspective); arguably, this skill is fundamental for CT as the latter requires the ability to formulate solutions to problems in a way that is executable by an information-processing agent (cfr. Section 1.2.1).

These core skills combine to form higher-order EFs such as **reasoning** (inductive and deductive logical reasoning), **problem solving** (a core component of CT) and **planning**.

1.4.2 EF and CT

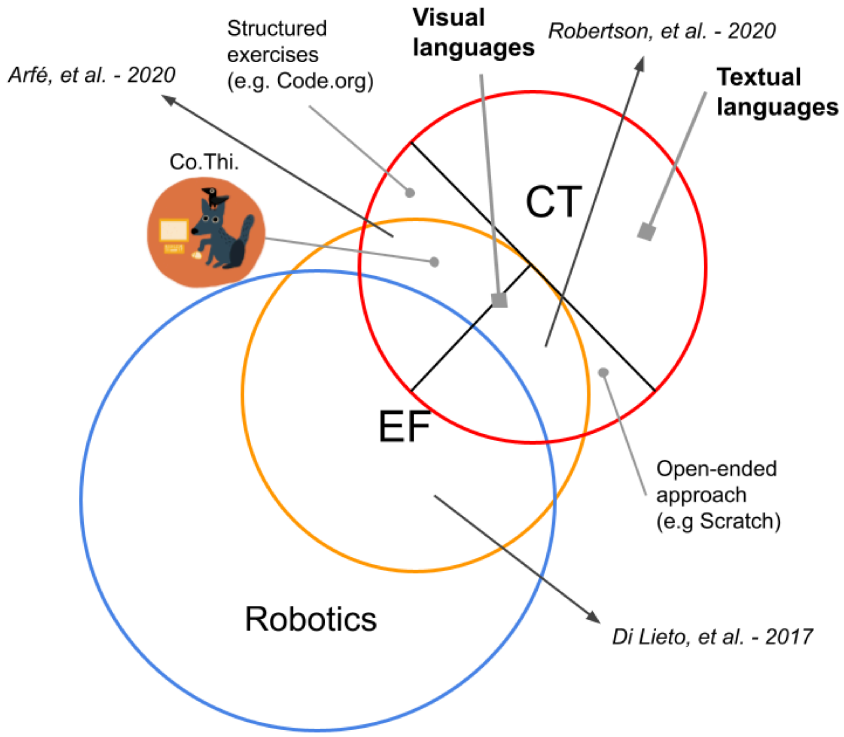

Figure 1.7: "Geographic" map of studies which combine CT and EF

This work builds on the foundations laid by our research group; in particular two recent works ([2, 3]) describe our basic approach and preliminary results: the authors test EF skills development (through standardized tests such as the TOL) of test and control groups of primary school children before and after periods of coding training for the test group (through exercises and activities taken

from Code.org) and standard Science, Technology, Engineering and Mathematics (STEM) activities for the control group. The results show improvement in planning and inhibition skills for the test groups.

To the best of our knowledge, this particular research field (the effect of CT, expressed through programming activities, on EF) is not overly populated; Robertson et. al, in [53], describe an experiment in which 23 students (11-12 year old) participate in a programming workshop with the language Scratch (a debugging task and a creative open-ended task); the authors use the Dr. Scratch tool (cfr. Section 1.5.2) to extract "CT coefficients" (i.e. data about the level of understanding of different concepts of CT like generalization, loops, etc.) and test the participants with the Behavioural Rating Inventory of Executive Function (BRIEF2) assessment of EF skills. The results show correlation between EF skills and the "CT coefficients" collected for debugging and creative programming.

Di Lieto et. al, in two studies ([54, 55]), approach the relationship between EF and CT from the point of view of *robotics*. Both works describe studies conducted with groups of 5 and 6 year old children who participate in workshops revolving around interactions with a simple robot (which can be manually programmed, through a series of buttons, to make it move according to the desired trajectory). The children are tested before and after the workshops with various neuropsychological tests (e.g. Forward and Backward Corsi Block Tapping for visuospatial memory, NEPSY-II for inhibition) and the results show improvements in working memory and inhibition for the test groups.

Figure 1.7 presents a "geographic" map of the cited studies.

1.4.3 The Tower of London test

The *Tower of London* task was first described by Timothy Shallice, in [56] (interestingly, the author reports having taken inspiration for the test from the domain of artificial intelligence and cites the classic problem of the Tower of Hanoi).

The test requires subjects to interact with beads (of different colors: blue, red, green) inserted in pegs (of different height) mounted on a board (see Figure 1.8): the goal is to move *one bead at a time* from a peg to another to reach a specific configuration (shown at the start of the test) with a maximum number of moves (dependent on the difficulty of the final configuration). The starting position of the beads is always the same (green and red beads on the longest peg, blue bead on the medium peg), the task can then vary in difficulty based on the number of moves required to reach the target configuration.

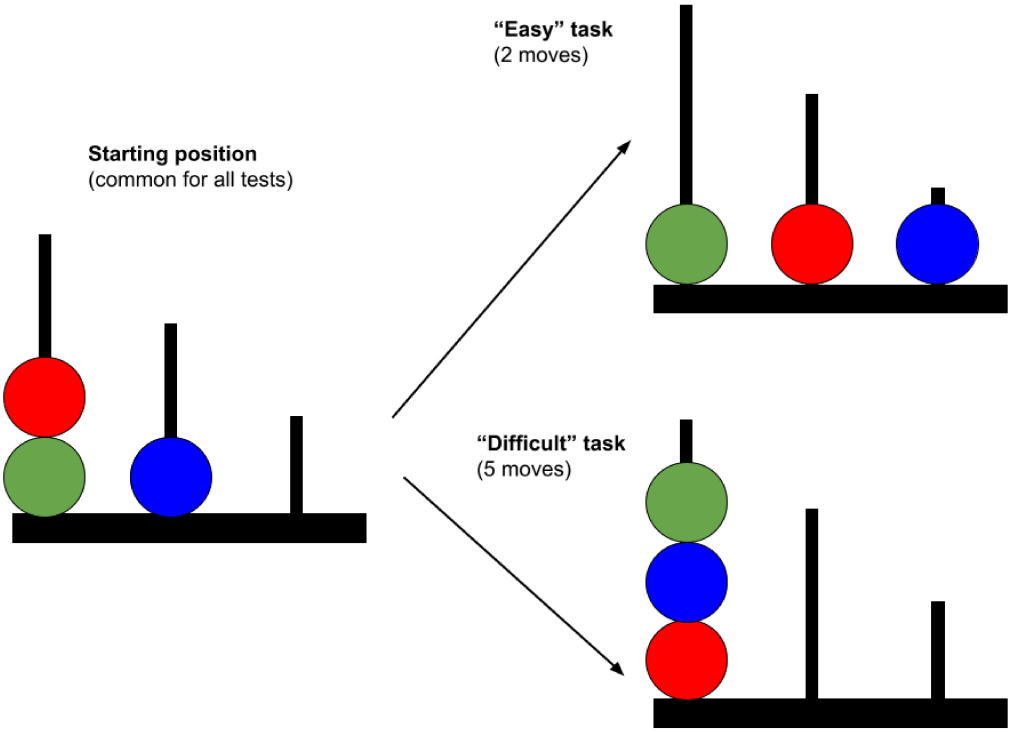

Figure 1.8: TOL starting position and two tasks of different difficulty

We chose to correlate our CT activities' data with TOL results (cfr. Chapter 3) because this test is used as a measure of **problem solving** (arguably a major component of CT and CS in general, Section 1.2.4 does not list it directly, but many of the components of Table 1.2 belong to it): of particular interest is the **planning time** which is measured by taking the interval from the moment the problem is presented to the subject of the test (i.e. the moment they are exposed to the visual stimulus of the target configuration) to the moment they take their first action to solve it (i.e. the moment they remove the first bead from its peg). This measure of planning finds an almost direct application in the context of our CT skills tests (i.e. the block-based programming exercises), for further details cfr. Section 2.4.1.

1.5 Learning Analytics

This Section presents the research field of *Learning Analytics* (LA) by analyzing its definitions and motivations, giving some concrete examples and, finally, discussing some of its ethical ramifications.

1.5.1 The research field of LA

The research field of LA is relatively young: its inception can be traced to the First International Conference on Learning Analytics and Knowledge (LAK2011), during which it was defined as "the measurement, collection, analysis and reporting of data about learners and their contexts, for the purposes of understanding and optimising learning and the environment in which it occurs" ([57]).

Another field of research concerned with similar problems is that of *Educational Data Mining* (EDM), which predates LA by a few years (the first conference was held in 2008[5]).

Baker, Gašević and Karumbaiah, in [58], differentiate LA and EDM by saying that EDM is mainly concerned with an *entitative* methodological paradigm (reducing a phenomenon into its smaller components and then analyzing the relationships among these components) while LA is mainly concerned with a *dialectical* paradigm (according to which the smaller components can be understood by understanding the whole system first).

To be more specific, we can look to another distinction drawn by Filvà et. al, in [59]: EDM, being a branch of data mining and machine learning, is more concerned with the collection and analysis of educational data; LA, on the other hand, analyzes students' interactions with learning systems to extract behavioural patterns and relate them to learning goals and outcomes.

For the purposes of this study, however, this differentiation is not too important; we use the term LA to refer to studies and techniques based on the definition given in LAK2011. In particular, we are interested in the application of these techniques to CT. One of the primary ways of teaching CT skills is through programming, be it via open-ended projects or increasingly complex and challenging problems: regardless of the modus operandi, the results of these activities are always *machine readable artifacts* (i.e. programs) which naturally lend themselves to automated analysis which can shed light on interesting facts.

LA, as a research field, is quite wide; we could see it as a spectrum, based on the data collection granularity: on one hand it includes works concerned with "high level" techniques and tools, such as Learning Management Systems (LMS) for higher education, similar to Moodle[6], which collect information about (online) class attendance, review of study materials, performance on weekly tests and quizzes ([60]); on the other end it includes works which collect and analyze "low

[5] `https://www.educationaldatamining.org/EDM2008/`
[6] `https://moodle.org/`

level" data including students' actions and timestamps relative to the solution of a single exercise ([59]). This project is focused on this kind of fine-grained data collection and analysis.

The motivations driving LA research are: reducing dropout rates, improving academic performance ([61, 62]), understanding learning strategies and processes, improving curricula, adapting learning paths to students and recommending content ([63]), developing learning and meta-learning skills through feedback ([60]). Other motivations, mainly related to student modeling, are the implementation of "early warning" systems for teachers and tutors ([58]).

Motivation	**Related works**
Student modeling: coefficients to measure the understanding of CT concepts	[64, 45, 65]
Student modeling: student categorization	[59, 66, 67]
Student modeling: performance evaluation	[68, 69]
Student modeling: error evaluation	[70]
Exercise modeling	[69, 47]
Automatic feedback and suggestions	[45, 71]
Adaptive learning	[47, 71]
Teacher dashboard for early warning	[67]
Effects of CT on EF	[2, 3, 54, 55, 53]

Table 1.5: List of LA works divided by motivation

Table 1.5 presents a summary of LA related works, divided by motivation.

1.5.2 Concrete examples

This Section contains a summary of LA techniques (of a "low level" nature, cfr. the previous Section) found in the literature. The following paragraphs distinguish them in works that mostly focus on CT *processes* or *outcomes*. Figure 1.9 shows the kind of information that we can gather from Code.org' exercises by using such techniques.

Works that mostly focus on the CT process

Filvà et. al, in [59], describe an approach to analyze the CT *process* of students programming in Scratch (i.e. "open-ended" tasks, cfr. Section 1.3.3) through *clickstream* (i.e. the collection and analysis of users' clicks while they interact with an application); they use unsupervised clustering algorithms (e.g. k-means)

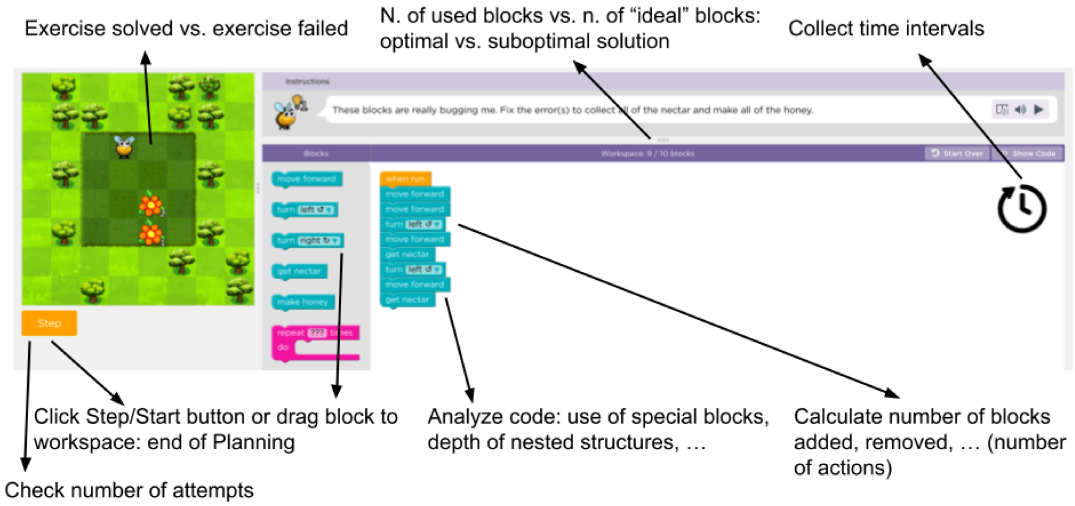

Figure 1.9: Example of information we can gather with LA techniques (exercise taken from Code.org)

to extract student models from the collected data (the subjects of the research are students aged 15-17). Based on this they define the following models:

- **Development-focused** students who concentrate most of their clicks in the script space of the Scratch interface.
- **Organization-focused** students who mostly move around blocks (instead of adding, deleting, modifying them).
- **Design-focused** students who mostly click in the sprite and scenario design areas.
- **Multimedia-focused** students who concentrate on the sound and drawing areas of the interface.

Furthermore, by checking the users' use of the "green-flag" button (which executes the developed code), they draw the following distinctions:

- **Blocked** users who lag behind the others for a variety of reasons (e.g. they could be focusing too much on the design of sprites, or they could simply lack understanding of the programming language). These are the users who click on the green flag below 10% of the average.
- **Trial-and-error** users who keep changing their programs and checking the results (those who click on the green flag above 90% of the average).

- **Balanced** users who develop at a normal pace.

This classification is particularly interesting because the authors find that it correlates with the students' results in the programming tasks: namely, the *trial-and-error* model shows a strong correlation with good results while the *blocked* model correlates with unfinished and very basic programs.

However, these results do not find confirmation in another work, by Kesselbacher and Bollin ([67]), who, in fact, find that the opposite is true. The authors collect very fine-grained data on the solution process and outcome of a single programming task (the subjects are lower- and upper-secondary school students): block creation (with relative type), deletion, drag-and-drop, reordering, etc. Their analysis shows that clicks on the "green-flag" button have a moderate negative correlation with task success. The authors discuss how this could be caused by the fact that novice programmers have an incomplete understanding of programming concepts and need to test them frequently by executing their code. By analyzing clusters of students' data (k-means and 4 clusters found with the "elbow technique") the authors confirm this result and find that a balanced usage of "special" blocks (representing programming concepts like loops, variables, etc.) tends to correlate with successful solutions.

Fields et. al, in [66] describe a different approach for the study of CT processes: the authors collect and analyze frequent snapshots of users' (aged 10-13) programs during a complex multi-day programming task (the development of a Scratch application). They show results related to two groups of metrics: missing initializations (e.g. use of a sprite without setting its initial state) and parallel programming. The results are plotted on a graph which shows the progress of these metrics during the multi-day project, this visual representation highlights interesting patterns, e.g. for the "missing initialization" metrics they find (1) a series of spikes which suggest a phase of efficient debugging (problems are solved as soon as they emerge) and (2) long "plateaus" which show how debugging becomes more difficult as the project's complexity increases.

Works that mostly focus on the CT outcomes

Moreno-León et. al, in [45], describe Dr. Scratch: a tool to analyze Scratch programs, i.e. programming *outcomes*. The tool assigns to each project a "CT score" based on the inferred level of understanding of the following CT concepts: (1) abstraction and problem decomposition, (2) logical thinking, (3) synchronization, (4) parallelism, (5) algorithmic notions of flow control, (6) user interactivity and

(7) data representation (note how some of these, being more "practically focused", are not among those listed in Table 1.2).

Dr. Scratch evaluates the CT concepts based on a rubric, with scores ranging from 0 (null), to 3 (proficient); for example: the use of an "if" block gives 1 point in the "logical thinking" category ("basic" competence), the use of an "if-else" block gives 2 points ("developing" competence), and so on. Moreover, the tool looks for specific bad programming habits (e.g. code repetition, incorrect initialization of object attributes, etc.) and uses them to give feedback to users (this feedback gets more detailed as the CT score increases).

This tool has proved to be quite popular and is used in other works, e.g. [72].

Another interesting approach for the analysis of CT outcomes is presented by Koh et. al, in a study ([32]) which focuses on *semantic* similarities among programs. The authors use vectors to represent users' games, created with the AgentSheets visual programming language[7], by collecting the presence and number of game actions and conditions (e.g. "move sprite up when key up is pressed"). These vectors can be compared (with cosine similarity) between one another or with a set of "canonical" vectors which represent CT patterns (in this case the patterns are related to video games, e.g. collision, hill climbing, transportation, etc.). The similarity score between a program's vector and the "canonical" vectors can be plotted on a radar graph (called "CT pattern graph") whose shape gives information on the usage of the different patterns: interestingly, the authors show how two different implementations of the same game show a very similar shape (same use of CT patterns) but two different scales (one implementation was more efficient than the other).

1.5.3 Ethical considerations

The amount of data collection and control that LA techniques allow should raise questions and (understandable) concerns about the goals and politics of those who use them. Data is not neutral (as a proof, cfr. the Facebook-Cambridge Analytica scandal) and the fact that something is technically possible (e.g., in this case, collecting fine grained data on students and their learning processes) is not, in itself, enough motivation for actually doing it.

This view is shared by Neil Selwyn who, in [73], critically addresses the very existence of LA. Admittedly, the author mostly refers to those techniques which I referred to as of "high-level" granularity in Section 1.5.1, i.e. tools used at an

[7] `https://agentsheets.com/`

institutional level (and which can enable "institutional surveillance"). However, I think it is important to clearly define an overarching goal for studies and activities in this field: for the present work the main (long-term) goal is being able to collect and present quantitative data to strongly support the argument for the inclusion of CS education in primary and lower-secondary schools (cfr. Sections 1.2.2 and 1.2.3).

1.6 Summary

This Chapter presented the goals of this project and grounded them in the state of the art of the recent literature.

We argued for the importance of teaching CT skills *as a conduit* to Informatics and for the importance of research showing the positive effects of CT activities *as quantitative evidence* for the inclusion of Informatics in K-12 curricula.

The interdisciplinary connection with psychological research allows us to leverage the power of *reliable, standardized* assessment techniques for EF skills to show the impact of CT skills on cognitive development.

The CS perspective of this work finds expression in the study of LA techniques to augment our data collection procedures and, potentially, to expand the amount of information that we can gather by observing CT activities' processes and outcomes.

These perspectives and foundations find natural concretization in the development of a platform which combines CT activities and LA data collection techniques, with the goal of correlating CT and EF results. We present this application in the next Chapter.

Chapter 2

Outline of the research project

Figure 2.1: Co.Thi. logo

This Chapter describes in detail the platform Co.Thi. (short for Computational Thinking, Figure 2.1 shows its logo): its structure, goals and how they were achieved.

2.1 Goals and boundaries of the intervention

As stated in Section 1.1, the main goals of this work are:

- To automate and augment the data collection about CT exercises' processes and outcomes.
- To analyze such data to extract new interesting information.

The limited time allocated for the project (~6 months) enforced some constraints on what we could achieve, in particular:

- We had to quickly build a prototype, in order to be able to collect data during some of the CT activities of the research group (cfr. Section 3.1).
- Because of the requirement of the previous point, we could not develop a new series of CT exercises and had to rely on already existing material: the choice (as for the previous publications of the research group: [2, 3]) fell on Code.org[8].

Code.org is a nonprofit dedicated to the goal of *expanding access to CS activities* (according to its website, over 70 million students from more than 180 countries engaged with its contents); its website provides a wealth of CT exercises addressed to students from all grades.

Because of its focus on spreading and easing access to CS activities, Code.org has a strong focus on usability and on being engaging for young users. Because of this, its exercises often use sprites and assets from popular video games and franchises (e.g. Angry Birds, Plants vs. Zombies, Frozen, etc.): this poses some issues tied to copyright, which we shall discuss further in Section 2.6. Moreover, even though the block-based programming language used in Code.org's exercises is built upon Google Blockly, and is therefore potentially capable of listening to users' interactions and collecting fine-grained data (for examples cfr. Section 2.7), the analysis of CT activities is not one of the goals of the platform.

The architecture of Code.org's codebase is shaped by its objectives and, therefore, it offers limited access to Blockly's data collection mechanisms; regardless, it can be augmented by what we call *digital sensors* and it can thus serve our purposes, for more details cfr. Section 2.4.2.

2.2 Overview

The preliminary results published by our research group (cfr. [2, 3]) are based on data collected in two ways: (1) measurements by hand (i.e. using a chronometer to capture planning and execution time intervals) and (2) measurements collected by a native Java application named *Coding Management Studio*.

The downsides of collecting data by hand are easily visible: high probability of human error (be it because of distraction or limited reaction times) and high

[8] `https://code.org`

human cost (every single test must be directly supervised in order to collect its data).

The software *Coding Management Studio*, while being a first step towards automation, has its downsides as well: (1) it is a native Java application, this means that it must be installed on every workstation on which tests are conducted; (2) it presents a browser-like view of an exercise from Code.org, with no access to its internal mechanisms; this means that there is no way of hiding unnecessary parts of the screen, preventing unwanted events (e.g. a login request which blocks the entire exercise) or capturing relevant information (e.g. it collects the planning and execution time intervals by requiring students to click on buttons which explicitly signal the beginning and ending of the respective phases); (3) the collected data is saved on the hard drive of the test workstation and must be gathered by hand (and combined with the rest) by the researchers.

Thus, the application *Co.Thi.* is built to provide these features:

- Being accessible through the **web**, eliminating the need for manual installation and taking advantage of one of the most widespread and standardized runtimes: the *browser*.
- Collecting exercise data **automatically and transparently** for the user, thus solving two problems: (1) eliminating the artificiality of having to explicitly signal the beginning and ending of a phase and (2) reducing human errors in measurements.
- Storing the collected data in a **central database** and providing tools to organize it in (csv) files ready for analysis with statistical software (e.g. SPSS[9]).

The platform consists of a client application which interacts with a backend server and an augmented fork of Code.org's codebase, for more details cfr. the next Section.

2.3 Project architecture and security mechanisms

Figure 2.2 presents an overview of the architecture of the application and a rundown of the technologies that support it.

[9] `https://www.ibm.com/analytics/spss-statistics-software`

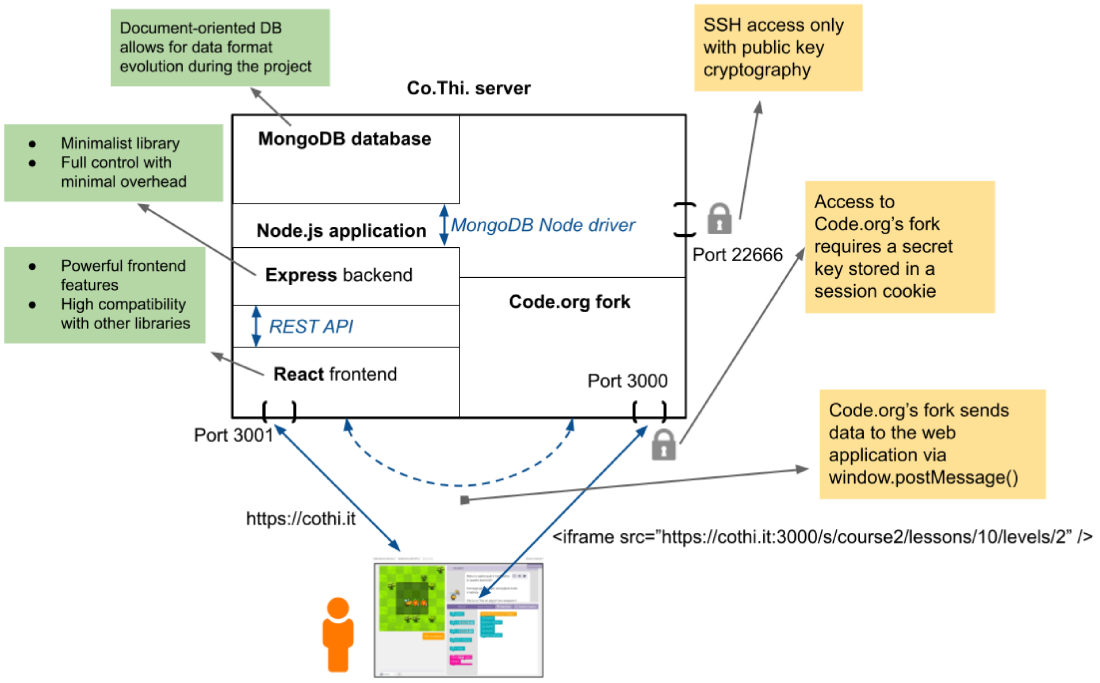

Figure 2.2: Co.Thi. architecture

We considered three alternatives to include Code.org's exercises in the web application:

- Embedding the exercises exposed on Code.org's website through an `iframe`: this proved impossible because Code.org is protected by the Content Security Policy (CSP) **frame-ancestors**[10] which only allows certain domains to embed a page. Moreover, Cross Site Scripting (XSS) protection mechanisms implemented in every major browser prevent javascript access to the DOM of embedded pages (if they are from a different domain).

- Extracting the exercise applications from Code.org's codebase[11]: this is technically possible but complicated by the fact that the codebase (1) is quite extensive (over 200k files with a combined weight of more than 11GB in the cloned repository) and (2) is based on Ruby on rails[12], a framework we are not familiar with.

- Running a fork of Code.org's codebase in parallel to the web application and embedding the exercises via `iframe`: this is the solution we chose in

[10] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors`
[11] `https://github.com/code-dot-org/code-dot-org`
[12] `https://rubyonrails.org/`

order to be able to (1) quickly prototype the application, (2) test it "in the field" with actual users and (3) use it during the CT activities conducted by our research group (for more details cfr. Section 3.1).

In order to enable the embedding of the exercises, the fork of Code.org must be exposed to the internet: this greatly extends the attack surface of the host machine. Thus, as a security precaution, we implemented a server policy which, for every incoming request (be it a POST request carrying content, a GET request for a web page, etc.), checks the presence of a specific session cookie carrying a secret key (the hash of a password): this cookie is set upon login to the Node.js web server and is flagged as `httpOnly` in order to minimize XSS risks[13]. A request without the security cookie gets a `403 Forbidden` response.

Admittedly, strongly motivated attackers could find a way to steal the security cookie and gain access to the exposed fork, however, this is not a high stakes application and the category of attack we can reasonably expect is that of automated scans and tests for common vulnerabilities. In order to shield the application from easy attacks we implemented the following security mechanisms in addition to the security cookie:

- Moving the SSH server from port 22 to port 22666 (usually automated scans only check the most used ports).
- Removing the possibility of SSH access via username and password (only public key cryptography).
- Using a `systemd` service to start and stop Code.org's fork in order to having it up and running only during school hours (thus temporally reducing the attack surface).

On top of the security measures, we extended Code.org's fork in order to (1) have the exercises collect data automatically (cfr. Section 2.4.2) and (2) send this data to the Node.js web application. This data exchange is handled using the `window.postMessage`[14] javascript API: this allows the React client application to work as a data collector and aggregator throughout the exercise solution process. At the end of each exercise, the cumulative data is sent to the backend server with a POST request.

[13] `https://owasp.org/www-community/HttpOnly`
[14] `https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage`

2.4 Data collection

This Sections first gives an overview of Code.org's exercises and their components and then describes our data collection techniques in detail.

2.4.1 Components of a Code.org exercise

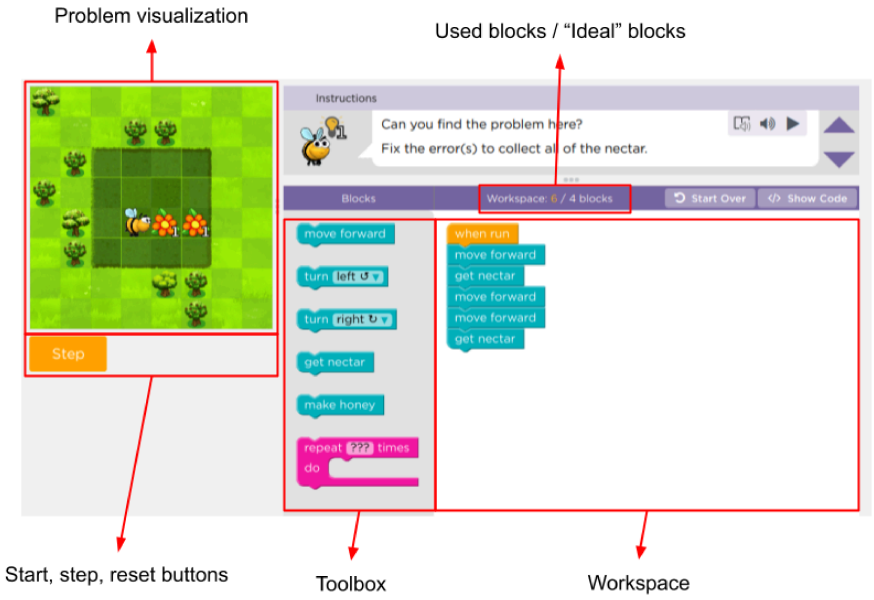

Figure 2.3: Components of a Code.org exercise

Figure 2.3 shows the typical interface for Code.org's exercises.

Each exercise presents a **problem** which can be visually analyzed in the *problem visualization* section of the screen. Different exercises can be about different types of problems; in the platform Co.Thi. we included: **maze** exercises, which consist of tile-based maps which a sprite must navigate, avoiding obstacles and possibly performing additional tasks (e.g. collecting nectar as shown in Figure 2.3); **artist** exercises, which take inspiration from classic Turtle graphics (cfr. Section 1.3.3); **jigsaw** exercises, which consist of simple puzzles (and can be useful for developing familiarity with the drag and drop mechanisms of the block-based programming language). Some of the exercises are about **debugging**: they present a problem and a (erroneous) draft solution; users are encouraged to analyze the proposed solution, find the mistakes and correct them (however, nothing prevents them from throwing everything away and starting anew, as reported in Section 3.4).

Aside from the problem, each exercise consists of a series of **tools** (i.e. code blocks available for its solution) which are shown in the *toolbox* section of the screen. Having only some code blocks available for a specific exercise is a form of scaffolding which has the goal of reducing the cognitive load for users: this is clearly a great distinction from "traditional" text-based languages (cfr. Section 1.3.1) but is also a distinction from "open-ended" block based languages like Scratch (in which every code block is always immediately available).

Users compose their programs by dragging code blocks from the toolbox to the *workspace* section of the screen and connecting them in a vertical column (the verticality is suggested by the jigsaw-like bumps and indentations on the blocks).

Each exercise has a (predetermined) **ideal number of blocks** which is shown by a number at the top of the workspace: this is the number of blocks which would be used for the optimal (i.e. the most efficient) solution; suboptimal solutions are allowed as well.

We divide users' solution processes in two phases:

- **Planning phase**: starts as soon as the components of the problem appear in front of the user (visual stimulus) and ends when the user takes their first action to solve the exercise (moving a block, changing a parameter, etc.). The boundaries set for this phase are very similar to those used for the TOL test (cfr. Section 1.4.3).

- **Execution phase**: starts as soon as the planning phase ends and terminates when the exercise is solved (for the first time) or failed for the third time. Note: this happens *as soon as the user clicks the "run" button*, regardless of the time Code.org takes to animate the sprite's traversal of the maze (or the artist's drawing).

2.4.2 Data collection details

Code.org's block-based programming language is based on a fork of Google Blockly[15] and presented in a wrapper which hides some functionalities (or at least makes them hard to reach): most importantly, there is no clear pointer to the exercise `workspace` handler which would allow to listen for Blockly's *Events* (which cover every user interaction we are interested in, cfr. Section 2.7).

[15] `https://developers.google.com/blockly/`

However, by using a *black-box* approach, interesting information can be gathered *directly from the DOM* through the javascript `MutationObserver` API[16]: this interface allows the handling of changes to the web page by listening to events fired by *observers* (a sort of digital *sensor*) attached to specific DOM elements. In particular, we augmented Code.org's code with two digital sensors:

- An observer attached to the `blockUsed` DOM element, which keeps track of the addition and removal of blocks to the user's program.
- An observer attached to the whole document, which checks for the addition of elements with the classes `blocklySelected` and `blocklyDraggable`, thus identifying when a code block is dragged through the screen.

Other useful information can be collected simply by checking for specific function calls and conditions; to make an example, the termination conditions for a user's program are handled by a `switch` statement: it is then sufficient to extend each condition in order to make it signal the relative event.

It should be clear that these augmentations are not particularly intrusive and, thus, data collection can be added to Code.org's exercises without dramatic changes to the codebase. However, a direct access to the `workspace` handle would provide a far better alternative, as we show in Section 2.7.

Code.org's fork sends data to *Co.Thi.*'s client application via javascripts's `window.postMessage` API; Table 2.1 presents a summary of the messages.

The client application listens for the messages sent by Code.org's fork and aggregates them to create a representation of the solution *process*; in particular:

- When it receives the `initData` message it sets the `planningStart` timestamp which signals the beginning of the **planning phase** of the exercise solution process.
- When it receives a `blockDrag` or `textChanged` message it sets the `executionStart` timestamp which signals the beginning of the **execution phase** of the exercise solution process.
- It uses the `blockData` messages to (1) keep track of the number of **actions** (additions and removals of blocks to the program) a user takes to solve the exercise; (2) keep track of the number of *used blocks* (`usedBlocks`) vs. the number of *ideal blocks* (`idealBlocks`): this is later used to determine if a

[16] `https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver`

Message	Sent when	Notes
`initData`	Exercise application is mounted	
`blockDrag`	User starts dragging a block from the toolbox (or the workspace)	This "sensor" is disconnected as soon as the message is sent for the first time
`textChanged`	User changes a block label (i.e. parameter)	This "sensor" is disconnected as soon as the message is sent for the first time
`blockData`	Whenever the "used blocks" label changes value	This sends the number of used blocks and the number of "ideal" blocks
`runButtonClick`	Whenever the user clicks the "run" button	Similar messages are sent for the "step" and "reset" buttons
`success`	User completed the exercise successfully	Attached to this message is the structure of the user's program
`failure...`	When the different failure conditions are met	e.g. exercise not terminated, infinite loop, etc. Attached to this message is the structure of the user's program

Table 2.1: Messages sent by Code.org's fork to the React client application

solution is **optimal**; (3) keep track of the *sequence of actions* of the user: e.g. addition of 3 blocks, removal of 2 blocks, and so on.

- The `success` message sets the `executionEnd` timestamp which signals the end of the **execution phase** of the exercise solution process.
- The different `failure` messages are used to keep track of the user's accuracy score: for each exercise there are 3 attempts and the final score is set as $3 - n$, where n is the number of attempts. The third `failure` message sets the `executionEnd` timestamp.
- The `success` and `failure` messages carry a representation of the relative user's program: this is stored and later sent along with the rest of the data.

As soon as the user terminates an exercise (by issuing the first correct solution or the third incorrect one) the React client sends the aggregated data to the backend server which stores it in the database after some final elaborations; in particular: (1) the backend analyzes the users' programs to determine the *depth* of their nested structures and the presence of *special blocks* (loops and conditionals) and (2) it stores the users' programs and sequences of actions in graph form.

Table 2.2 presents the exercise *process* and *outcome* metrics collected in the database.

Name	Description
`planningTime`	Difference between `executionStart` and `planningStart` (ms)
`executionTime`	Difference between `executionEnd` and `executionStart` (ms)
`success`	`true` if exercise solved in 3 or less attempts
`accuracy`	$3 - n$ where n is the number of attempts
`optimalSolution`	`true` if `usedBlocks` $\leq$ `idealBlocks`
`numOfActions`	Number of additions and removals of blocks from the user's program
`if`	`true` if user's program has a conditional block
`repeat`	`true` if user's program has a loop block
`depth`	Depth of nested block structures in user's program

Table 2.2: Exercise metrics stored in the database

2.5 Typical usage

The web application supports two types of user, described in the next Sections; in particular: Section 2.5.1 describes the typical application usage of the user who has access to the exercises and Section 2.5.2 describes the Admin dashboard.

Coding and *Admin* users receive different *bundles*: these are compressed javascript files which contain the application code along with all its dependencies and are generated with **webpack**[17].

Having separate applications for different kinds of users has the following advantages:

- **Reduced weight**: every application bundle only includes the dependencies and code which are actually used by the respective users.
- **Increased security**: users never receive code they are not allowed to execute, this reduces the risk of client-side manipulations and security bypasses.

The browser retrieves the bundles through a `script` tag (present in the `index.html` page served by the backend) whose `src` field points to a web API designed to discriminate the users' types. This API first checks if the user has an active session (otherwise it serves the *login application* bundle); then, it checks the user type (based on their database record) and finally, it sends the appropriate bundle file.

[17] `https://webpack.js.org/`

2.5.1 "Coding" user

The *Coding* user has access to the *exercise* bundle: this is the application used for the training activities and the tests.

The typical workflow for this application is the following:

1. Select the school and student: we call this **test user**, to differentiate it from the actual *Coding* user.

2. (Optional) If this is the first access for a new test user: select the preferred language (the choice is among Italian, English and French).

3. Select the desired **activity**: this can be a **test** (divided in T1, T2 and T3, cfr. Section 3.1) or a **training** (for which a specific track must be specified).

4. Access the exercise **track** relative to the selected activity, complete it and return to the activity selection page.

Figure 2.4 presents the first three steps of this workflow.

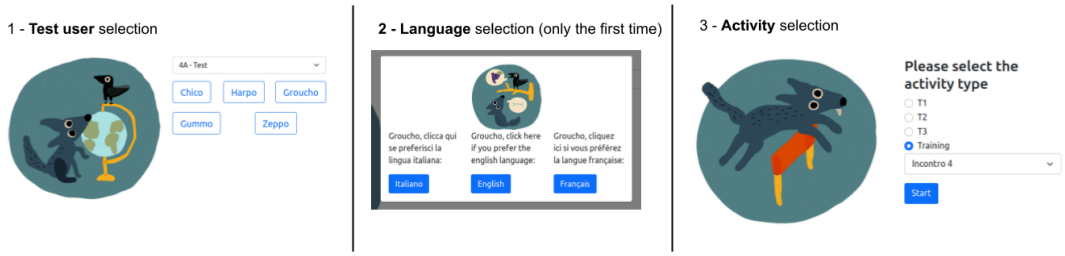

Figure 2.4: User and activity selection in the Co.Thi. web application

The information about the **test user** is stored in a global storage variable handled by the React client application: this is possible because the application routing is entirely handled client-side (with the **React router** library[18]) and thus the application state persists across all the different pages. However, a simple page refresh would cause the bundle to be downloaded again and the application state to be lost, so, as a fail-safe mechanism, the React application stores some key information in the pages' urls.

Figure 2.5 shows the key information saved in an exercise's url: whenever a React component is loaded, its constructor checks the current url and the global storage variables to see if it is lacking some information or if there are

[18] `https://reactrouterdotcom.fly.dev/docs/en/v6`

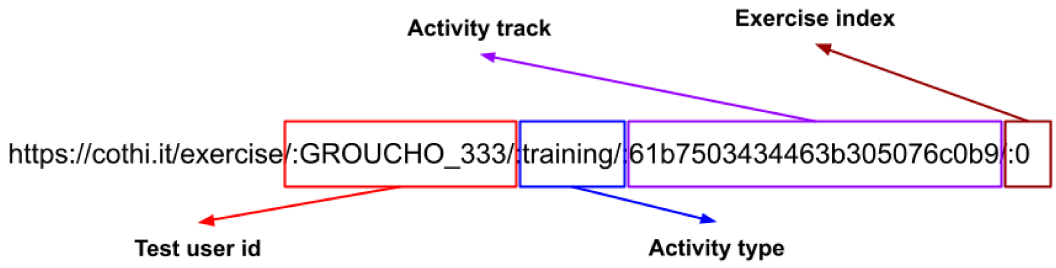

Figure 2.5: Parts of an exercise's url

inconsistencies (e.g. different test user id). In case of problems it uses the url-stored information to retrieve everything it needs from the backend.

This fail-safe mechanism also enables the bookmarking of exercise urls.

2.5.2 "Admin" user

The *Admin* user bundle offers some functionalities designed for the researchers and instructors conducting the training and test activities.

Its main functionalities are:

- A **test user** page which shows which subjects are saved in the application database and allows admins to create and remove test users and test groups (i.e. school classes).
- A **Result upload** page which allows the uploading of the results collected for the cognitive tests (e.g. the TOL) in csv form.
- A **Result download** page which offers the possibility of downloading the combination of the cognitive test results and the data collected for the coding activities in a csv file ready for input to data analysis software (for this project we used R, cfr. Chapter 3).
- A **data analysis** page which shows some interesting information about the users' programs.

Figure 2.6 shows an example of information shown on the data analysis page: graphs representing the *program* or *action sequences* of users. The idea for these representations comes from the interaction networks discussed in [47].

Both of these graphs are directed acyclic and share three nodes (distinguished by their bigger radius):

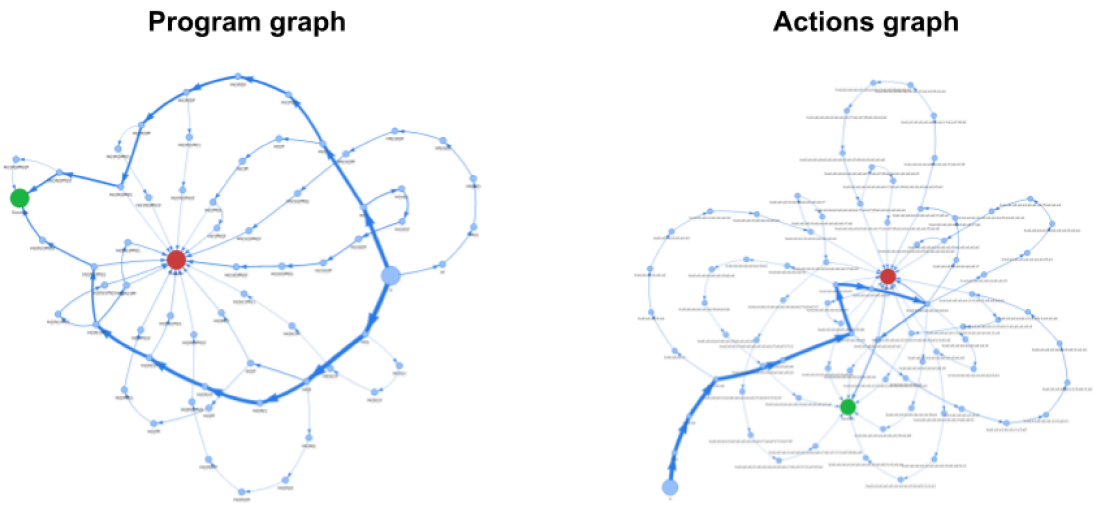

Figure 2.6: Example of graphs shown on the **data analysis** page

- The *blue node* marked with the label **b** indicates the beginning of the program (the `start` block which is common to all solutions).
- The *green node* marked with the label **Success** represents a correct solution.
- The *red node* marked with the label **Failure** represents an erroneous solution.

Each (internal) node in the **program graph** represents a particular prefix of a final solution. Each node in the **actions graph** represents a particular sequence of actions (additions and removals of one or more nodes). The edges' widths indicate the frequency with which they are traversed in users' solutions.

The examples of Figure 2.6 give information about an exercise which requires the user to guide a sprite towards a target destination by taking one of two possible routes (to see the exercise components cfr. Figure A.5). The program graph clearly shows that users solve the problem with two possible solutions (the two bigger routes through the graph, starting from the larger blue node on the right and ending at the green node on the left). The actions graph shows how all users start the solution process with the same actions (addition of the first few blocks, one at a time) but then start taking different approaches: there are a few routes to success and many more to failure.

Figure 2.7 shows the tabularization of the program and actions graphs shown in the data analysis page. The **program graph table** counts how many routes pass through a particular node (i.e. a particular program prefix). The **actions graph table** counts how many routes pass through a particular edge (i.e. a particular action).

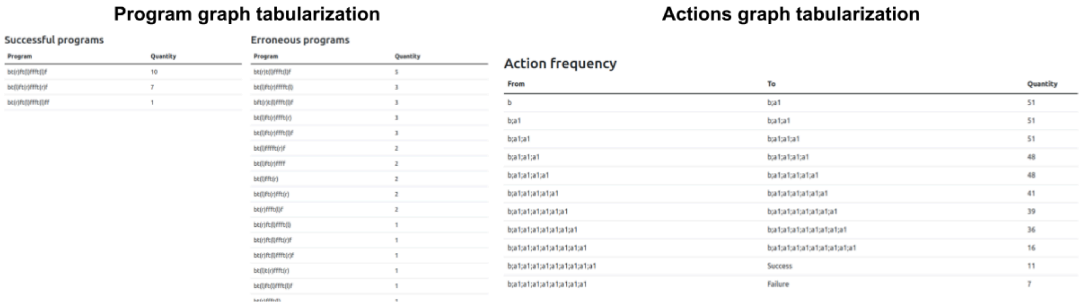

Figure 2.7: Example of tables shown on the **data analysis** page

Section 3.4 presents some results based on these data representations.

2.6 Limitations

This Section describes the limitations of this work, mostly caused by the necessity of working with Code.org's exercises and the resulting application architecture (cfr. Section 2.1):

- Code.org's goals do not include data collection and analysis. Because of this, its codebase does not offer handy mechanisms to gather data on users' interactions with its exercises.
- Code.org's codebase is very extensive and complicated because of its nature as an organically grown open-source project with many different inputs and lots of "insider knowledge" (there is some useful documentation, but it hardly covers most of the intricacies of the codebase).
- Code.org's exercise "apps" bundles can be quite weighty: by minifying the packages we can reduce their size from ~40MB to ~10MB, however, this is still quite large for a web application.
- Code.org' intellectual property and exercises are protected by copyright: we managed to obtain an authorization to use a fork of the codebase for our project, however the process was long and there are no real guarantees for the future. Moreover, some of the exercises use sprites and assets from popular video games (this is clearly motivated by an attempt to make the activities more interesting and engaging), and using them would require separate agreements (again, a time consuming process with no guarantees): for this reason we decided to "reskin" these exercises with Code.org's assets for which we obtained authorization.

On top of these limitations, by observing our subjects' coding activities and discussing with them, we collected the following observations on Code.org's exercises:

- The presence of the **step** button adds some ambiguity to the exercise interface: students find it hard to understand that the possibility of executing the program step-by-step is just a visual helper and that their program is actually fixed once they make the first step (i.e. if they modify their code they have to reset the animation). This is communicated by Code.org with a textual prompt but we think it would be better to directly remove this option (at least during tests).

- Some of the exercises' visualizations (cfr. for example Figure A.6) are difficult to interpret in terms of distances (i.e. users find it hard to see the underlying grid). For this reason we think it would be better to overlay an explicit grid on top of the exercise "maps".

- Some of our users did not like working with some of the exercises' sprites: allowing them to choose their preferred avatar for every activity is an easy addition which would eliminate this problem.

2.7 Co.Thi. 2.0

Having seen in this Chapter how we can expand an existing platform *built for a different purpose* with data collection mechanisms, it is interesting to see what we can achieve by building a new application *with this specific goal as a foundation.*

This proof-of-concept application (labeled *Co.Thi. 2.0*) offers potential for very fine-grained data collection and can be the basis for further experimentation (cfr. Section 4.2).

Figure 2.8 shows the interface of a *Co.Thi. 2.0* exercise designed to teach loops, along with its representation in file form. At the current stage the *problem representation* is very primitive (it adopts conventions from roguelike[19] video games by representing the user sprite with "@" and walls and obstacles with "#"), although it can represent the sprite movement through the maze: a future, more complete, version will have full graphics.

[19] `https://en.wikipedia.org/wiki/Roguelike`

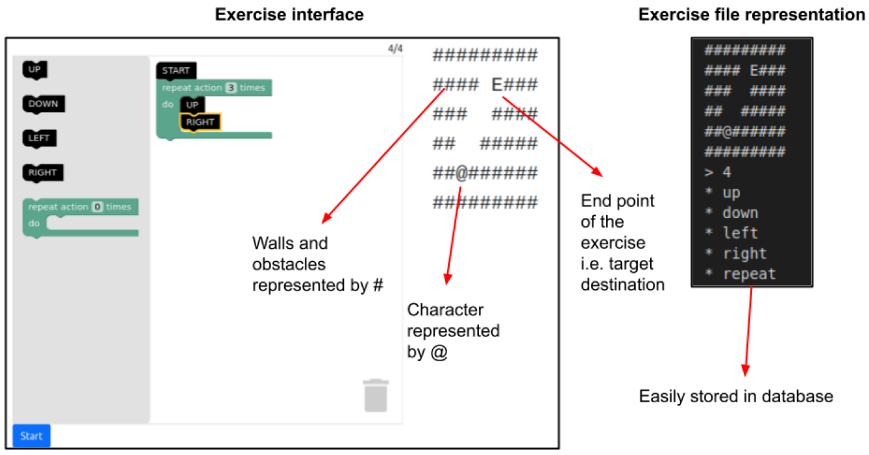

Figure 2.8: Exercise from *Co.Thi. 2.0*, along with its file representation

This new version of Co.Thi. is based on Google Blockly and can leverage the full power of the library: in particular, it listens for *workspace events* to collect the information which in the first version requires `MutationObserver` objects.

Event	Description
`BLOCK_DRAG`	Start or end of a block drag
`BLOCK_CHANGE`	Change to a block field with old value and new value (e.g. number of loop repetitions)
`BLOCK_MOVE`	Change of position with starting and ending position and, possibly, ids of parent blocks
`BLOCK_DELETE`	Deletion of a block or group of blocks
`BLOCK_CREATE`	Creation of a new block (e.g. after having dragged a block from the toolbox to the workspace)
`BLOCK_SELECT`	Selection of a block (by clicking on it)

Table 2.3: Blockly events and information they carry

Table 2.3 presents a list of Blockly Events and the information they carry; by comparing it with the digital sensors of Section 2.4.2 the advantages of having access to this interface are clear. To make some examples:

- The `BLOCK_DRAG` event signals the start or end of a block drag: thus it is possible to determine *how much time it takes* to drag a specific block. By combining this information with e.g. the block type we could try to infer if some blocks cause more indecision (which in this case would be suggested by a particularly long block drag).
- The `BLOCK_MOVE` event signals the start and end position of a block movement, it can also signal the `id` of a parent block if the movement caused

a connection to be created or severed. This information can be used to detect interesting patterns like a block "shuffle" (change of position inside the structure of the program).

This prototype is just a proof-of-concept of the possibility of reproducing Code.org's main qualities and perfect our data collection mechanisms. Further work is needed to make it useful for future research: Section 4.2.1 presents some goals and reflections on this matter.

2.8 Summary

This Chapter presented the main concrete products of this thesis' work: the platform *Co.Thi.* and its prototype follower *Co.Thi. 2.0*.

The main highlights we can extract are:

- Fine-grained data collection about block-based programming activities is, indeed, possible: in the best-case scenario we can ultimately store enough data to recreate the entire solution process, step-by-step with precise time intervals.
- Code.org's codebase could be augmented with kind of data collection, with minimal overhead (especially if Blockly's `workspace` handle was unwrapped and made available to the codebase user). The wealth of data that could be collected by this platform, accessed by millions of users, is easily imaginable.

We were able to extend our data collection beyond the basic accuracy metrics and planning and execution time intervals. The next Chapter presents our experimental evaluation and shows the results we were able to collect with our augmented data collection techniques.

Chapter 3

Experimental evaluation

This Chapter presents the study we conducted with the help of the Co.Thi. platform and the data we were able to collect about primary-school students' CT processes and outcomes.

3.1 Test organization

Because of the timing of the project, we were able to deploy the platform *Co.Thi.* for the first time in December 2021: this was the middle point of an activity, conducted by our research group, which we shall describe in this Section.

This project, informally called *Project Coding*, involves activities very similar to those described in the articles previously published by my research group ([2, 3]): a group of interns from the Department of Psychology (we shall call them *instructors*), under the guidance of a PhD student from the same department, conducted *training* and *test* sessions in two primary schools in the city of Padua, from September 2021 to May 2022.

The project's subjects were students from 1st and 4th grade classes who were divided in a *test* and *control* group (we shall refer to them interchangeably with the terms *students* and *subjects*).

Figure 3.1 presents a timeline of the activities for the two groups. This kind of experiment involves three tests:

- **T1** is the *baseline* CT and EF test: it is done before any other activity and its results are used to check the effects of the *training* on students.
- **T2** is done after the *test* group's training: 2 hours of coding a week, for a month, totaling 8 hours.

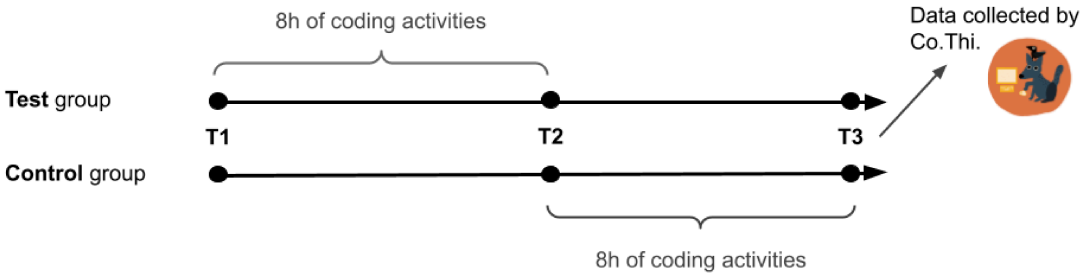

Figure 3.1: Timeline of activities for test and control group

- **T3** is done after the *control* group's training and also works as a *follow-up* for the *test* group (e.g. to check for skill retention).

The T1, T2 and T3 CT and EF tests involve the *same* tasks (i.e. the same coding exercises, the same TOL configurations, etc.).

The timing of the deployment of *Co.Thi.* allowed us to collect on the platform the CT test results for T3.

3.1.1 CT tests

The coding tasks include 4 exercises: Tables 3.1 and 3.2 present an overview of the exercises for the first and fourth grade groups. To see the visual representation, toolbox and "ideal" solution for each exercise cfr. Appendix A.

Exercise	**Concepts**	**Notes**	**Original url**
Exercise 1	Sequences	Simple navigation with absolute directional blocks (up, down, left, right)	`https://studio.code.org/s/course1/lessons/4/levels/9`
Exercise 2	Sequences & Debugging	Exercise initialized with an (erroneous) solution	`https://studio.code.org/s/course1/lessons/5/levels/2`
Exercise 3	Sequences	Navigation in a different context (Turtle graphics)	`https://studio.code.org/s/course1/lessons/8/levels/3`
Exercise 4	Loops	This exercise requires loops involving one or more blocks (repetition of an action or a sequence of actions)	`https://studio.code.org/s/course1/lessons/14/levels/4`

Table 3.1: First grade test exercises overview

Exercise	Concepts	Notes	Original url
Exercise 1	Sequences	Navigation with relative movement blocks (move forward, turn left, turn right)	`https://studio.code.org/s/course2/lessons/3/levels/5`
Exercise 2	Loops	Potentially solvable with nested loops	`https://studio.code.org/s/course2/lessons/6/levels/8`
Exercise 3	Sequences & Debugging	A loop block is available in this exercise's toolbox (but the optimal solution does not require it)	`https://studio.code.org/s/course2/lessons/10/levels/4`
Exercise 4	Conditionals	This exercise requires some actions to be taken only under certain conditions (i.e. collect nectar only if present)	`https://studio.code.org/s/course2/lessons/13/levels/5`

Table 3.2: Fourth grade test exercises overview

Students actually solve 8 exercises during the course of the CT test: each "official exercise" is preceded by a "rehearsal exercise", based on the same concepts and used to refresh the students' skills; however, the subjects are not made aware of this difference (so that they engage each exercise with the same concentration and effort).

The *training* activities consist of workshops conducted in the schools' computer rooms during which the instructors presents students with a series of exercises of increasing difficulty selected in order to cover the CT concepts examined during the tests (in particular: sequences of operations, loops and conditionals). During these activities students work alone at their workstation, attempting to autonomously solve an exercise; they can later compare their results and discuss difficulties (usually the instructors call forth one of the students and ask them to discuss their solution). The instructors offer help to those in need by suggesting different approaches and stimulating their problem solving process but they never give the final solution.

3.1.2 EF tests

The research project involves different EF tests, in particular: the TOL test for planning skills and Nepsy-II and numerical Stroop for inhibition. For this thesis, we only focus on the TOL (cfr. Section 1.4.3).

The rules for this test are:

- The subjects must move only one bead at a time.
- At any time at most one bead can be placed on the shorter peg, two on the middle peg, three on the longer peg.
- When the subject removes a bead from a peg, they must place it down on another before taking another bead (i.e. any bead movement must be sequential).
- Every final configuration (based on its difficulty) has a maximum number of allowed moves.
- The subjects must solve each task (i.e. each final configuration) in under 60 seconds (counted from the moment they receive the visual stimulus to the moment they complete the task).

For each task completed following these rules, students receive 1 **accuracy** point.

3.2 Data overview

Table 3.3 presents the list of metrics we used for our data analysis.

To summarize how we compute these metrics:

- **Tower of London (TOL) planning time (TP)**: measured from the moment the subject receives the (target configuration) visual stimulus, to the moment they make their first move. For the correlation analysis we take the *mean* of the planning times across the tasks and exercises (this also applies to execution times and to coding metrics).
- **TOL execution time (TE)**: measured from the first move to the moment the subject recreates the final configuration.

Metric name	Definition
TP	TOL planning time
TE	TOL execution time
TA	TOL accuracy
CP	Coding planning time
CE	Coding execution time
CA	Coding accuracy
CAO	Coding accuracy + optimality
CAS	Coding accuracy + special blocks
CNA	Coding mean number of actions
CMD	Coding mean depth
CBD	Coding block difference
CSO	Coding sum of optimal solutions
CHP	Planning time change score
CHE	Execution time change score
CHA	Accuracy change score

Table 3.3: Metrics glossary

- **TOL accuracy (TA)**: 1 point for every TOL task completed following the rules (cfr. Section 3.1.2). For the correlation analysis we take the *sum* of the accuracy measures across the tasks and exercises (this also applies to coding metrics).

- **Coding planning time (CP)**: measured from the moment the subject receives the (coding exercise) visual stimulus, to the moment they make their first move (dragging a code block or changing a block parameter).

- **Coding execution time (CE)**: measured from the end of the planning time, to the moment the subjects run a correct solution for the first time or an erroneous solution for the third time.

- **Coding accuracy (CA)**: calculated with $3 - n$, where n is the number of attempts.

- **Coding accuracy + optimality (CAO)**: this is equal to $CA+1$ if the exercise solution has an optimal number of blocks (i.e. $\leq$ than *ideal blocks* of the exercise, cfr. Section 2.4.1); otherwise it is equal to CA.

- **Coding accuracy + special blocks (CAS)**: this is equal to $CAO + (1 \times m)$ where m is a score which gives 1 point for every relevant special block

(loops, conditionals) used in the exercise solution. A *relevant* special block is one which is actually necessary to the (optimal) solution (e.g. a loop block in a loop exercise, etc.): we check this by comparing the blocks used in a solution against a list of exercise annotations.

- **Coding number of actions (CNA)**: this metric counts the number of additions and removals of (one or more) blocks to the subjects' programs (for the correlation we take the *mean* across the exercises).
- **Coding mean depth (CMD)**: *mean* depth of the nested structures of the subjects' programs, across all exercises.
- **Coding block difference (CBD)**: this is calculated as the *sum* of $u - i$ across all exercises, where u is the number of *used blocks* and i is the number of *ideal blocks*. It is an indicator of the "verbosity" of the subjects' solutions.
- **Coding sum of optimal solutions (CSO)**: $1 \times o$, where o is the number of optimal exercise solutions.
- **Planning time change score (CHP)**: this metrics is calculated as the difference between the (coding) planning times measured after the coding training and the one measured before. For *test* subjects this means the difference between planning times at T2 and T1; for *control* subjects the difference is taken between T3 and T2 (this applies to all change scores).
- **Execution time change score (CHE)**: difference between the (coding) execution times measured after and before the coding training.
- **Accuracy change score (CHA)**: difference between the (coding) accuracy measures calculated after and before the coding training.

Our tests included 4 first grade classes (totaling 69 students) and 3 fourth grade classes (totaling 57 students). Tables 3.4 and 3.5 present some details on the metrics collected for these two groups: apart from the change scores (which are computed differently based on the test or control group), we collected all these values at **T3**.

As a first step of analysis, we checked the normality of the data with the *Shapiro-Wilk* test implemented in the `shapiro.test` function[20] of the R programming language. Most of the metrics are not in normal form: this is not

[20] `https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/shapiro.test`

Metric	Normal	Mean	Standard deviation
TP	X	3.18	1.50
TE	✓	15.35	4.13
TA	✓	6.25	2.68
CP	X	11.67	4.37
CE	X	83.74	34.95
CA	X	5.93	1.35
CAO	X	8.96	1.90
CAS	X	9.46	2.23
CNA	X	9.26	3.34
CMD	X	0.24	0.13
CBD	X	1.26	2.29
CSO	X	3.03	0.75
CHP	X	-3.24	8.26
CHE	✓	-21.09	52.05
CHA	X	2.79	2.05

Table 3.4: First grade data characteristics (69 subjects)

particularly surprising because the results were collected after training sessions for all the subjects. Thus we should expect *skewness* towards positive results (e.g. higher accuracy, lower execution times, etc.). Figure 3.2 shows an example of this bias in the results: in particular it is interesting to note how the *CAS* metric makes the skewness more pronounced for the fourth grade results (suggesting a more accurate representation of the actual coding skills of the students).

The negative value of the *Planning time change score* indicates that, in general, students reduced their planning after the training activities: this result is unexpected because careful planning is actually one of the skills that the training attempts to teach. However, as Arfé et. al, discuss in [3], this is probably caused by the increased familiarity that the students develop with the exercise interface (meaning less time spent deciphering its parts): Section 4.2.2 discusses this matter in further detail and proposes some future research directions to tackle this problem.

3.3 Correlations

Because of the non-normal distribution of most of the metrics collected at T3, we decided to perform our correlation analysis with *Spearman's rank correlation coefficient*: differently from the classic *Pearson's correlation*, this method does

Metric	Normal	Mean	Standard deviation
TP	X	2.09	0.78
TE	✓	13.90	3.45
TA	X	8.62	2.20
CP	X	10.24	4.40
CE	X	99.99	50.82
CA	X	4.16	2.07
CAO	X	6.84	2.75
CAS	X	8.42	3.18
CNA	X	12.99	4.69
CMD	X	0.67	0.16
CBD	X	1.39	2.12
CSO	X	3.02	0.72
CHP	✓	-5.92	6.25
CHE	✓	-39.30	60.97
CHA	X	2.67	1.97

Table 3.5: Fourth grade data characteristics (57 subjects)

not assume the normality of the data and is usually recommended with these distributions (Bishara and Hittner, in [74] observe that this is the most commonly recommended method by statistics textbooks).

Tables 3.6 and 3.7 present the correlation coefficients computed for the first and fourth grade subjects. All statistically significant values are presented in bold, the number of asterisks is tied to the p value of the correlation tests ($^{*}p < 0.05$, $^{**}p < 0.01$, $^{***}p < 0.001$).

The first and fourth grade groups share some, but not all, of the significant correlations. We discuss these in the next Subsections.

3.3.1 Shared results

The most interesting results shared by both first grade and fourth grade students are:

- A *positive* correlation between **TOL and coding accuracy** measures: this is stronger for the first grade results (~0.4 vs. ~0.3 for fourth grade results), moreover, the "augmented" coding accuracy metrics show slightly better results compared to the "standard" accuracy. This results suggests that **problem solving skills** (measured by the TOL test) have a role in determining the outcomes of the coding activities.

	TP	TE	TA	CP	CE	CA	CAO	CAS	CNA	CMD	CBD	CSO	CHP	CHE	CHA
TP	1														
TE	−0.101	1													
TA	0.009	0.186	1												
CP	−0.007	0.045	−0.177	1											
CE	−0.024	−0.037	−0.087	0.018	1										
CA	0.021	0.223	**0.396	−0.089	**-0.332	1									
CAO	0.068	0.186	**0.401	−0.089	*-0.278	**0.914	1								
CAS	0.054	*0.251	**0.405	−0.113	*-0.295	**0.898	**0.965	1							
CNA	0.118	0.094	0.055	−0.199	**0.666	−0.085	−0.008	0.012	1						
CMD	−0.053	0.078	*0.312	−0.027	−0.116	**0.420	**0.412	**0.438	−0.025	1					
CBD	0.100	0.115	−0.007	0.035	0.137	0.143	−0.039	0.051	*0.268	−0.011	1				
CSO	0.167	0.127	**0.341	−0.058	−0.130	**0.522	**0.791	**0.751	0.089	**0.325	*-0.275	1			
CHP	0.087	−0.232	−0.129	*0.306	−0.208	0.050	0.111	0.078	−0.107	0.003	−0.110	0.128	1		
CHE	0.151	−0.148	0.048	−0.141	*0.291	−0.007	0.056	0.034	*0.275	0.117	−0.020	0.141	0.184	1	
CHA	−0.093	0.052	0.090	**0.362	−0.032	0.205	0.124	0.105	−0.026	−0.019	0.232	0.014	−0.120	**-0.338	1

Table 3.6: Correlations for first grade students (*$p < 0.05$, **$p < 0.01$, ***$p < 0.001$)

	TP	TE	TA	CP	CE	CA	CAO	CAS	CNA	CMD	CBD	CSO	CHP	CHE	CHA
TP	1														
TE	0.034	1													
TA	**0.354	*-0.321	1												
CP	**0.361	0.055	0.241	1											
CE	0.142	0.082	−0.103	0.188	1										
CA	0.020	−0.169	*0.293	−0.029	**-0.369	1									
CAO	0.059	−0.200	**0.369	0.039	**-0.388	***0.965	1								
CAS	0.036	−0.208	*0.330	0.011	**-0.394	***0.950	***0.988	1							
CNA	0.135	0.031	−0.086	0.051	***0.818	*-0.273	*-0.288	*-0.298	1						
CMD	0.183	0.149	−0.118	0.083	0.081	0.137	0.108	0.135	0.110	1					
CBD	*-0.299	0.146	*-0.341	−0.165	0.045	0.008	−0.043	0.015	−0.105	0.088	1				
CSO	0.158	−0.163	0.260	0.161	−0.018	0.252	**0.403	**0.396	0.082	−0.021	**-0.357	1			
CHP	0.053	0.083	−0.216	−0.055	0.005	−0.032	0.010	0.012	0.070	0.039	−0.188	*0.325	1		
CHE	−0.014	−0.145	0.026	0.056	−0.044	0.020	0.090	0.093	0.044	0.006	−0.100	0.081	0.123	1	
CHA	−0.141	−0.032	0.026	0.051	−0.043	0.251	0.196	0.167	−0.055	0.069	0.052	−0.076	−0.068	**-0.379	1

Table 3.7: Correlations for fourth grade students (*$p < 0.05$, **$p < 0.01$, ***$p < 0.001$)

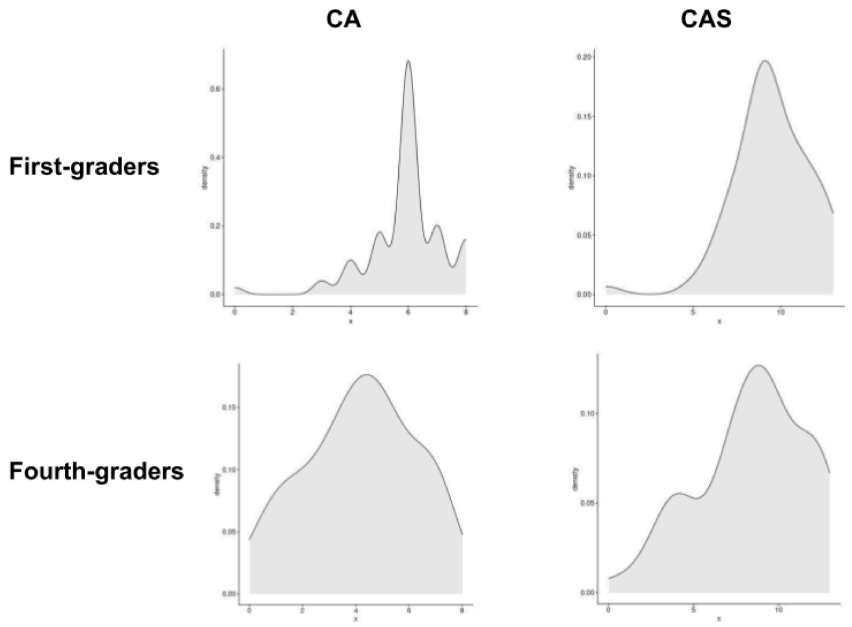

Figure 3.2: Density plot of accuracy metrics for first and fourth grade (CA: coding accuracy, CAS: coding accuracy with special blocks)

- A *negative* correlation between **coding accuracy and execution time**: this is stronger for the fourth grade results (~0.4 vs. ~0.3). This result suggests that spending less time solving a task (possibly because of more confidence, less hesitation) correlates with better results.

- A *positive* correlation between **coding execution time and number of actions**: unsurprisingly, more actions take more time to complete. Interestingly, even though accuracy correlates with execution time and execution time correlates with number of actions, accuracy does not correlate with number of actions (at least for first grade results), empirically showing that correlations are not transitive (cfr. [75]).

- A *positive* correlation between the **sum of optimal coding results and the "augmented" coding accuracy metrics**. This is not surprising: the sum of optimal solutions is actually part of these metrics (cfr. Section 3.2).

- A *negative* correlation between the **coding accuracy and execution time change score**: this means that subjects who improved their results in coding exercises tended to shorten their execution times. This finding corroborates the second result presented in this list.

3.3.2 First-graders' results

The most interesting results for first-graders are:

- A *positive* correlation between the **depth of programs and the accuracy metrics for TOL and coding**. This suggests that the use of special blocks (for first graders: loop blocks) correlates with better exercise solutions (which in turn correlate with better TOL results). This result is not reflected by fourth-graders: possibly, the reason is in the fact that the more complex fourth grade exercises create opportunities for more confusion in the use of special blocks.
- A *positive* correlation between the **sum of optimal coding results and the accuracy metrics for TOL and coding**. This result goes one step further than the one shared by both groups (a correlation with the "augmented" accuracy metrics).
- A *positive* correlation between the **coding accuracy change score and the coding planning time**. This means that subjects who improved their results after the training also tend to spend more time planning.

3.3.3 Fourth-graders' results

The most interesting results for fourth-graders are:

- A *positive* correlation between **coding and TOL planning times**. Even though the planning time change score is (on average) negative, this result shows that fourth graders, having learned this skill (planning), tend to apply it in different contexts.
- A *negative* correlation between the **number of actions and coding accuracy**. This is similar to the results which indicate a negative correlation between coding accuracy and execution time: more actions suggest an insecure approach. This result suggests the importance of further exploring and extending this metric (e.g. by counting also the number of block drag and drop gestures, etc.).
- A *negative* correlation between **block difference and sum of optimal results**. This is not surprising, a higher block difference score indicates the use of more blocks (with respect to the ideal numbers), this also impacts the optimality of the results.

- A *negative* correlation between **block difference and TOL planning and accuracy**. This suggests a correlation between **problem solving** skills and the ability of creating effective and efficient solutions. However, this result is not corroborated by a correlation with coding accuracy measures.

3.4 Other observations

A visual inspection of the data representations shown in the **data analysis admin page** of the *Co.Thi.* application (cfr. Section 2.5.2) suggests some patterns in the subjects' solution processes; in particular:

- Each exercise shows *few* (distinct) correct and *many* (distinct) wrong solutions.
- The students often solve *debugging* exercises by deleting *all the default code* and starting anew (surely an inefficient debugging practice).

Table 3.8 presents quantitative results computed from the **programs** and **actions** graphs and tables in the data analysis page.

Metric	**First-graders**	**Fourth-graders**
Successful programs mean entropy	0.91	1.49
Erroneous programs mean entropy	4.25	4.86
Bad debugging practices	16.36%	31.00%

Table 3.8: Quantitative data on program variety and bad debugging practices

The **program entropy** is a measure of the variety of the program set and is calculated with the formula $-\sum_{j=1}^{n} p_j * log_2(p_j)$ where n is the number of different (successful or erroneous) solutions and p_j is the frequency of a particular solution. The results of Table 3.8 show how erroneous solution sets have a variety ~4 times higher than the successful solution sets (as Leo Tolstoy famously stated in the novel *Anna Karenina*: *"All happy families are alike; each unhappy family is unhappy in its own way."*).

The entropy of the solution sets could be an indicator of the complexity of problems (e.g. a problem requiring loops could be solved without them by simply sequencing the correct number of actions): Table 3.9 presents the entropy of the correct and erroneous solution sets for each exercise, along with the respective success rate (a successful exercise is one solved in less than three attempts). In

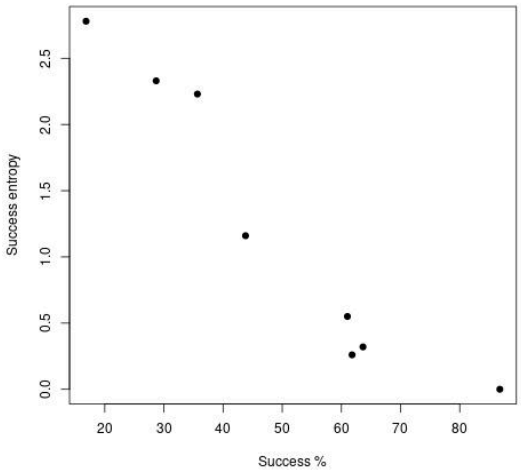

Figure 3.3: Success percentage and entropy correlation: *** – 0.95

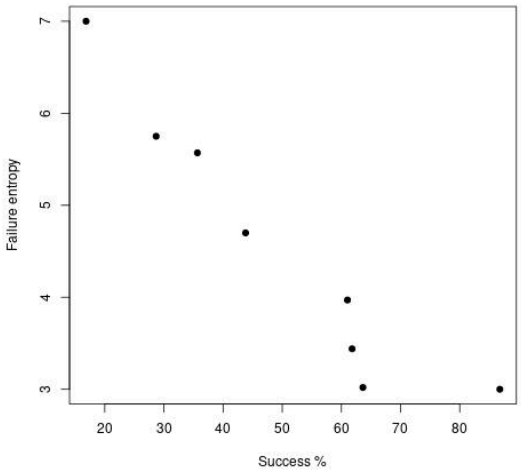

Figure 3.4: Success percentage and failure entropy correlation: *** – 0.95

this case, the *Shapiro-Wilk* test attests the normal distribution of the data so we can use *Pearson's* method to check for correlations: both the success and failure entropy columns correlate with the success percentage with a coefficient of ∼-0.95 and a p-value < 0.001.

Exercise	Grade	Success %	Success entropy	Failure entropy
1	1	61.02	0.55	3.97
2	1	63.64	0.32	3.02
3	1	86.78	0.00	3.00
4	1	16.84	2.78	7.00
1	4	43.80	1.16	4.70
2	4	28.68	2.33	5.75
3	4	35.66	2.23	5.57
4	4	61.80	0.26	3.44

Table 3.9: Success percentage and solution sets' entropies for each test exercise

These results are corroborated by a visual inspection of the scatter plots of the table columns presented in Figures 3.3 and 3.4.

The **bad debugging** measure counts the percentage of (successful or erroneous) actions sequences for debugging exercises which have the removal of all the default program blocks as the first user action. This percentage is not high but nevertheless not negligible for first grade results; however, it is quite high for

fourth grade results, especially considering that, for the fourth grade debugging exercise, this debugging practice means deleting 8 blocks of code (3 for the first grade exercise, cfr. Figures A.2 and A.7).

3.5 Summary

This Chapter presented the experimental evaluation of the platform *Co.Thi.*, inserted in the wider context of a research project involving students from two primary-schools situated in Padua.

The highlights we want to point out are:

- Augmented, more fine-grained, metrics seem to give better representations of the reality of subjects' CT activities. This encourages further refinements and research in this direction.
- Data can give us information regarding (1) users' CT process and outcomes and (2) exercises and activities' qualities (e.g. the complexity of a task based on the variety of solutions that users produce).

The next Chapter presents the conclusions we draw from this project, with an analysis of the information we can extract from raw data and a discussion of the research directions this information suggests.

Chapter 4

Conclusions

This Chapter draws conclusions on the present work. In particular, we discuss the results we were able to achieve and possible directions for future work.

4.1 Discussion

As stated in Section 1.1, the two main goals of this work are (1) the augmentation and automation of data collection about Computational Thinking activities and (2) the extraction of new interesting information with particular attention towards correlation with cognitive results.

With regard to the first goal: Chapter 2 lists simple techniques which can be "plugged-in" in Code.org's codebase to collect Computational Thinking *process* and *outcome* data, thus showing that these augmentations are, indeed, feasible without dramatic changes. However, we also discuss and show examples of how an architecture guided by this goal *from the beginning* is probably the best option if we want to have the best possible data representation of users' interactions with Computational Thinking exercises (cfr. Section 2.7).

As for the second goal: Chapter 3 (in particular Sections 3.3 and 3.4) presents the statistical analysis of the data we were able to collect. The following paragraphs draw some conclusions from the raw results.

The correlation results shared by both groups of students (cfr. Section 3.3.1) are probably the strongest finding of this work. The positive correlation between the accuracy measures of the coding exercises and the Tower Of London tasks suggest that these activities share a "core skill" which we intuitively identify as **problem solving**. This is not surprising but is made relevant by the fact that the Tower of London is a standardized, reliable measure (something we lack for

Computational Thinking skills) which enables us to infer information about the development of Computational Thinking skills. Moreover, these correlations show that *refining* the accuracy metrics (e.g. by taking into account the use of special blocks and the optimal results) can improve the results: we think this is due to the *better representation of the results* that these augmented metrics give (this is also corroborated by the examples shown in Figure 3.2, in which the augmented metrics accentuate the skewness of the data).

The negative correlations between coding execution times and accuracy measures, the positive correlations between number of actions and execution times and the *lack* of a shared correlation between number of actions and accuracy suggest that a more fine-grained data collection and *process* representation could shed some light on behaviors which lead to better results. Figure 4.1 shows examples of detailed data collection on the solution process and relative potential insights: a better representation of the solution process would enable us to detect fine-grained events like block reordering (possibly tied to a "trial and error" approach) and time intervals for each user action, which could help discriminate "easy" actions (i.e. actions taken in a confident way, with small individual intervals) from "hard" actions (i.e. actions taken after longer intervals of doubt).

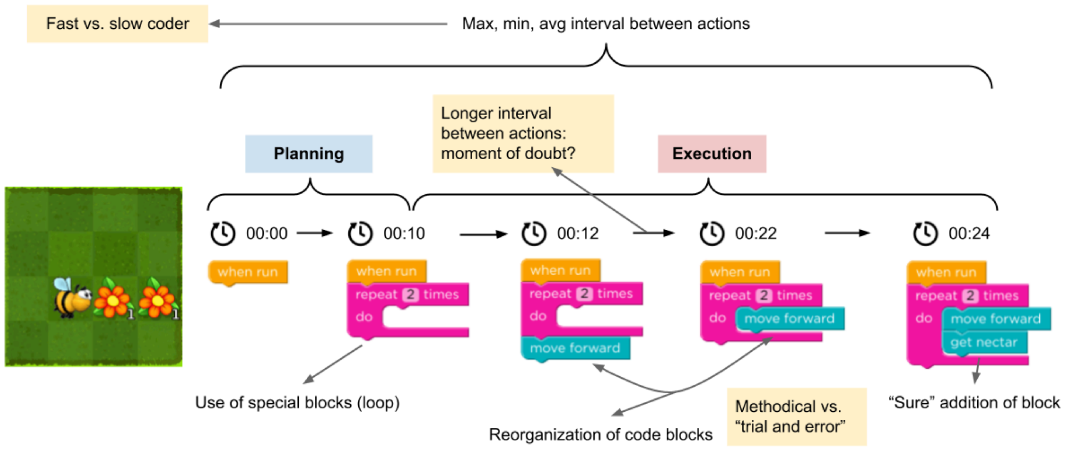

Figure 4.1: Potential insights gained from a more fine-grained analysis of the CT *process*

The fact that results pertaining planning times are not shared between the two groups is unexepcted (planning is a major component of problem solving), this is also true for the negative planning time change score. This suggests that (1) planning times, as we currently define them, probably include other activities (e.g. interface parsing) and (2) greater effort is required in refining these measures;

for further considerations on the matter cfr. Section 4.2.2.

We also show that storing visual representations of the processes and outcomes of the exercise solutions can give interesting information, especially when combined with first-hand observations by the instructors. In particular: the "bad debugging practices" discussed in Section 3.4 were first observed and reported anecdotally during training and test sessions; by inspecting the data we were able to factually show that they are indeed common, especially among fourth-graders. This result indicates that future training activities will need to put more emphasis on debugging strategies.

Finally, collecting users' solutions and actions can give interesting insights; specifically, our program entropy measure shows strong (negative) correlation with success percentages of single exercises. A great variety of erroneous solutions clearly indicate a high level of confusion for a particular exercise: in this case the set of programs includes a spectrum which goes from incomplete solutions to actual errors, Figure 4.2 shows examples of both categories; interestingly, the error shown on the right is probably caused by difficulties in understanding distances in this particular exercise (different subjects lamented the fact that the "grid" underlying Code.org's exercises is not always clearly visible, cfr. Section 2.6).

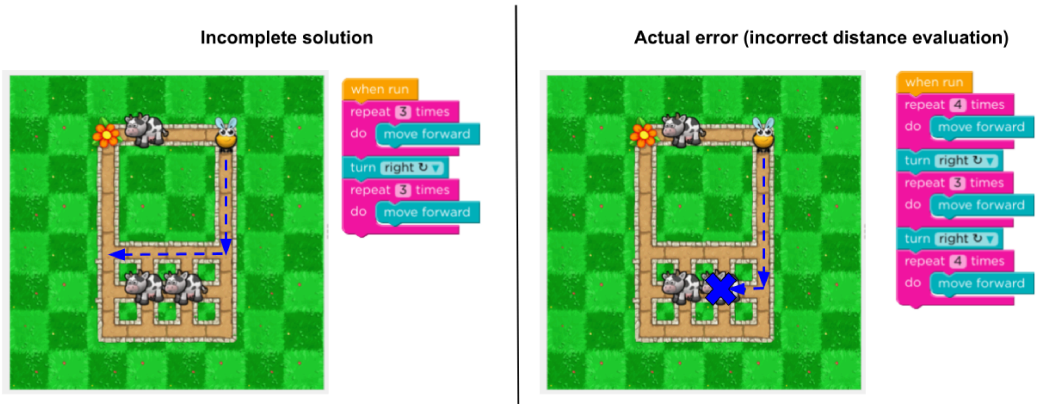

Figure 4.2: Examples of variety in erroneous solutions: incomplete solution vs. actual error. Dashed lines represent repetition of single actions.

On the other hand, a variety of correct solutions indicates that an exercise can be solved in more than one way and this, in turn, means that it is probably more complex or that it requires the use of special blocks. Figure 4.3 shows the two most common correct solutions for an exercise requiring the use of loop blocks: the solution on the left is sub-optimal and shows a lesser understanding of loops (in particular, users seem not to understand that loops can repeat *groups* of actions), however, this is the most common solution for this particular exercise. The

solution on the right shows a deeper understanding of the concept of loops (and is considered optimal by Code.org). Very few users understood the possibility of nesting loop blocks (as shown in Figure A.6).

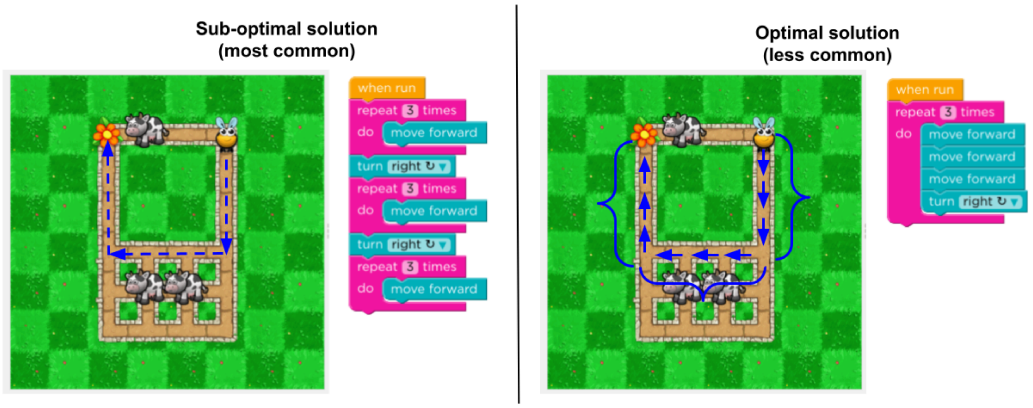

Figure 4.3: Examples of variety in correct solutions: sub-optimal vs. optimal. Dashed lines represent repetition of single actions, brackets indicate repetitions of groups of actions.

To conclude, the program entropy metric can be useful (in retrospect, or possibly as an early warning mechanism during training activities) to assess the difficulty level of exercise tracks and potentially to improve them for future research.

4.2 Future work

This Section presents directions for future work which we could not pursue during the limited time allocated for this thesis.

4.2.1 Extending Co.Thi. 2.0

One of our main avenues for future work is, without doubt, the continuation of the *Co.Thi. 2.0* project: at the present stage it is only a prototype and lacks graphics and exercise materials. This project has many potentials, some of which are possible integrations and augmentations, discussed in the next Sections. Figure 4.4 shows a mockup of a possible future expansion: showing an exercise not only through a tile-based 2d map but also through a 3d projection of the exercise "world". This could help students "step in the shoes" of the exercise characters and coming to grips with *relative* directional instructions (e.g. "turn right", "turn left").

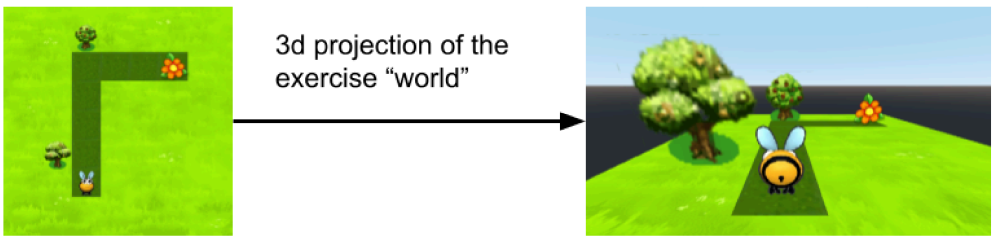

Figure 4.4: Problem visualization in 3d space

Another important aspect of the continuation of this project is the **design of a usable block-based programming language**: *Blockly* provides tools to define and implement code blocks; however, their design (e.g. block labels, colors, behaviour etc.) is left to the library user.

Blockly developers at Google have published some insights and lessons learned which can be useful as a reference for this design work.

Neil Fraser, in [76], cites (among other things):

- The difficulty new users have in understanding and using conditional and loop blocks: these should be made distinct (e.g. by using different colors) and placed in different categories (or clearly separated in the toolbox).
- The fact that *syntax errors* are actually possible in Blockly (albeit in edge cases) when blocks are very close to each other, but not connected: the suggestion is to automatically connect blocks which are suspiciously close to each other without being officially connected.
- The importance of using icons and images whenever possible to help understanding the exact meaning of a block (the example is the already mentioned "turn right" and "turn left" relative directional blocks).
- Avoiding too much scaffolding in the form of default code to "fill" with new blocks: this impacts the motivation of users, specifically the feeling of "code ownership" they have (the satisfaction of completing an exercise, etc.). The suggestion is *building on a user's previous solutions* (this is interesting as it could be a way to mix the open-ended and structured approaches discussed in Section 1.3.3).

Pasternak et. al, in [77], provide some tips for creating a block-based programming language; first they state the importance of focusing on the intended audience and the appropriate scope of the language (e.g. what are the goals of

the users, what types of blocks they need to reach them, etc.). Then they make some observations:

- They argue that icons can be useful for pre-literate users but also warn that images can carry ambiguity and can not really express complex concepts (e.g. a conditional block).
- They state that text-based blocks belong to a spectrum which goes from natural language to code-like language. Designers should choose where to place their block-based language based on the goals of the application (e.g. being a first introduction to programming vs. being a transition to text-based languages). The language could also move along the spectrum, based on the competency of the student.
- Finally, they point out some important usability features designers should keep in mind: consistency in block color, sentence structure, etc. Default parameters to avoid problems due to uninitialized blocks. Testing early and testing often in front of end users.

4.2.2 Augmentation of data collection: eye tracking

One of the most interesting concepts studied by our research group is the possibility of separating the **planning** and **execution** phases of an exercise solution to better understand users' problem solving processes.

However, while the boundaries we set for the planning phase (cfr. Section 2.4.1) are a first step towards this distinction, it is not straightforward to understand what this phase actually includes. In particular, one source of "noise" in the planning phase time intervals we collect could be caused by *unfamiliarity with the interface*, meaning that a portion of the planning time interval is actually spent exploring the interface and understanding its different components.

Indeed, Arfé et. al, in [3], discuss how their subjects' planning times *decreased* after having received training in coding activities: this result was unexpected because part of the training is dedicated to teaching to stop and think (i.e. plan) before acting. The authors hypothesize that this could be caused by the increased familiarity with the interface, meaning that the planning times collected *after the training* more closely approximate users' *actual* planning times.

To confirm this hypothesis we would need to understand where exactly the users focus their attention on the exercise interface: the technique of eye-tracking

could prove useful for this purpose (Figure 4.5 shows an example of useful discrimination powered by eye-tracking).

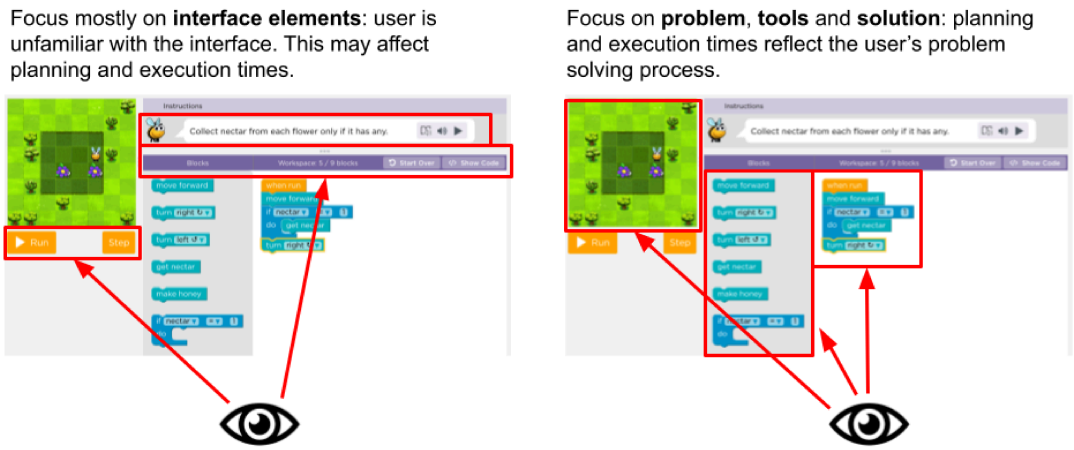

Figure 4.5: Using eye-tracking to understand how much of users' attention is devolved to deciphering the exercise interface

This technology has been widely used, e.g. to study how users approach information presented in web pages (usually, for usability purposes); some recent studies discuss its application to programming interfaces.

Sofia Papavlasopoulou (along with different groups of researchers) wrote several works on the subject. In [78], along with her co-authors, she reports on a study conducted on 44 subjects divided in "kids" (aged 8-12) and "teens" (aged 13-17): the experiment consisted of coding (creating a game with Scratch) and robotics workshops in which the subjects participated wearing eye-tracking glasses. The results show that "teens" spent more time focusing their attention on scripts (users' programs), commands (the programming blocks' "toolbox") and output (the resulting game or animation) while "kids" favoured the section of the interface dedicated to the design of sprites. They also analyzed the "transition patterns" (i.e. the subsequent focus on different parts of the screen) finding that "teens" performed more transitions between scripts and output/robot (suggesting an iterative development approach, with frequent checks) and scripts and commands (suggesting a debugging approach or the intention of finding the correct block for a specific goal). Finally they report on high correlation between focus on the scripts, output and commands sections of the interface and the Relative Learning Gain (RLG, a measure which accounts for the increasing difficulty of learning new things in a field in which one is already knowledgeable).

In [79], the authors study the relationship between gaze patterns and users'

attitudes towards coding (this is interesting as it is an attempt to understand if attitudes, which are commonly self-reported, can be measured, or correlated, with quantitative data). In particular, they use the following gaze measures: (1) *fixation duration*, i.e. how much time a subject spends inspecting a particular piece of information (usually associated with difficulties in understanding it); (2) *saccade amplitude*, i.e. the spatial distance covered when changing focus between different areas of the screen; (3) *change in saccade direction* which, when wider than a certain threshold, can indicate confusion or failed verification of a particular hypothesis. The authors correlate these measures with (self-reported) attitude indicators (perceived learning, intention to code and excitement for the activity) and report interesting results e.g. lower fixation duration for increased perceived learning.

Finally, in [80], the authors use eye-tracking and qualitative data to study gender differences in coding activities. The setup of the study is similar to the two previously mentioned works (children aged 8-17 participating in coding and robotics workshops while wearing eye-tracking glasses). The authors report no significant difference in the RLG and gaze measures for the different genders. On the other hand, the qualitative data shows some differences, for example:

- Only girls reported initial doubts about the coding activities (i.e. not looking forward to it, thinking it's not for them, etc.); however every subject reported satisfaction and increased confidence at the end of the study.

- "Leader" figures emerged only in mixed teams (with the role taken by boys who already knew about coding); girl-only teams reported an equal distribution of roles.

Moreover, the authors observed different approaches during the workshops: females distributed roles and responsibilities, started by designing and planning their games and paid more attention to the paper tutorial provided by the instructors. Males tended to jump straight in the Scratch interface to try things out. The difference between the *qualitative* and *quantitative* results collected by the authors in this work is interesting: it could suggest the role that societal norms and "implicit constraints" play in opening and closing (figurative) doors for children based on their gender.

To sum up, eye-tracking can be an interesting technique to augment LA with different goals. A webcam-based eye-tracking library which would be interesting

to test is WebGazer.js[21], developed by researchers from Brown University ([81]).

4.2.3 Adaptive/Responsive learning

One of the promises of Learning Analytics techniques and, in general, of data collection is the possibility of creating and feeding precise *models* to represent complex realities: in this case, for example, students' learning paths. A system able *react* to such a model could then *act* on the teaching material and *adapt* it to *individual* students' goals and needs: Figure 4.6 presents an overview of this process, seen through a cybernetics metaphor and applied to the context of block-based programming activities.

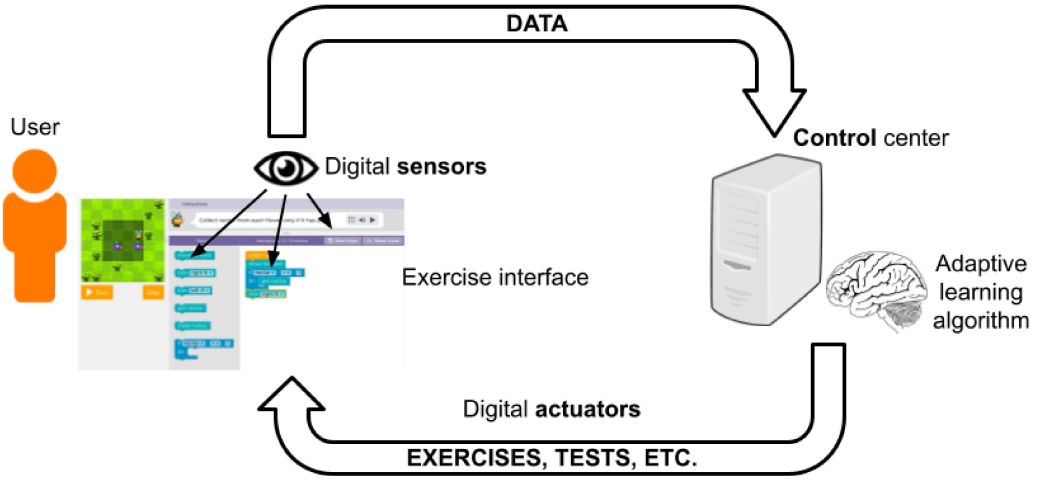

Figure 4.6: Overview of the adaptive learning process, seen through a cybernetics metaphor

It is important to state that every form of teaching is already, to some extent, adaptive: teachers interact with their students and can choose to focus more on some parts of the syllabus, quickly review already mastered materials, etc. However, the (teaching) trajectory they follow must approximate the *average* optimal path for an entire classroom (of course, we are not considering one-on-one tutoring in this case): therefore, the attractiveness of Learning Analytics powered adaptive learning is in the possibility of supporting teachers in creating the best possible learning trajectory for each student (e.g. by generating ad hoc training sessions with exercises specifically designed to overcome individual knowledge gaps).

[21] `https://webgazer.cs.brown.edu/`

Monica Bulger, in [82], approaches this concept critically and draws important distinctions on the meanings included in the buzzword *personalized learning*. In particular, she distinguishes between *resposive learning*, i.e. personalized interfaces to "static" material (more akin to an interactive textbook) and *adaptive learning*, i.e. systems able to evolve in response to students' changing goals and needs (which pursue the ideal of a digital personal tutor). The main difference between the two is that in responsive learning *the content does not change* in response to the students (instead, it is recommended just like products and movies are recommended in platforms like Amazon and Netflix). The author also states the importance of clearly defining the goals of these systems (with an implicit critique of data for data's sake, akin to some considerations presented in Section 1.5.3).

The most important goal we see for the use of adaptive/responsive learning techniques is keeping the students in a state of *flow* (cfr. [83]). By focusing our attention on the specific context discussed in this thesis, i.e. puzzle-like programming activities with block based programming languages, we can imagine an adaptive system which keeps track of students' proficiency with different Computational Thinking concepts (e.g. loops, conditionals) and generates or presents exercises tailored to the resulting needs.

Few researchers are leading the work in the subject of adaptivity: one of them is Tomáš Effenberger (who dedicated to the concept his Master's [84] and doctoral [85] thesis). This author describes a seminal adaptive learning algorithm for block-based programming in [47]; this work describes a programming environment (called RoboMission[22]) similar to Code.org's "mazes": each exercise consists of a grid-based map through which the user must guide a spaceship while collecting some diamonds. Users control the spaceship with a block-based programming language based on Google Blockly. The platform controls users' trajectories through the exercises with a learning algorithm:

- Exercises are divided in sets based on their difficulty: users must reach a certain level of mastery in a particular difficulty level before being able to proceed to the next.

- Users can reach the "mastery level threshold" for a specific set if they present a flawless solution to one of the exercises. Otherwise the mastery level is calculated in order to enable the users to proceed to the next level when they

[22] `https://en.robomise.cz/`

present successful (albeit only "adequate") solutions for all the exercises in the current set.

- In order to determine the quality of users' solution, the platform collects detailed data like: time necessary to program the solution, number of updates to the program, number of attempts, etc.

This algorithm shows the importance of discriminating users' solutions beyond the binary categories of "success" and "failure": this is further discussed by Effenberger and Pelánek in [69]. In this work the authors critique both the binary performance model (it does not give enough information) and the machine learning approach of collecting large amounts of raw data and feeding them to a model (difficult to reuse in different contexts). They instead propose an augmentation of the discrete performance categories to: failure, poor performance, good performance, excellent performance. These categories should be tailored to give information useful to change the learning trajectory: e.g. an exercise solved with poor performance is probably presenting a task too difficult for the user, an exercise solved with excellent performance is probably too easy, etc. To assign exercise solutions to these categories the authors consider: the time necessary to solve them, the number of edits and the number of executions (note: exercises in RoboMission have a maximum number of blocks so the program length metric cannot be used). To give an example: an exercise solution which only required one execution (one attempt) can be classified as *excellent*, a solution which required up to 5 executions as *good*, and so on.

Another researcher interested in the field of adaptive/responsive learning is Benjamin Clement (cfr. his doctoral thesis [86]). For example, in [71], the author describes an approach to choose activities based on the expected learning gain, based on *Multi-Armed Bandit*:

- The algorithm is built upon a set of skills to acquire (e.g. sum of integers, subtraction of integers, sum of decimals, etc.) and a set of exercises and activities.

- The instructors sort the activities in a "canonical trajectory" which is the initial trajectory for every student.

- The student model receives a reward for completing an exercise: this changes over time (e.g. after having successfully completed an exercise multiple times it becomes null).

- The algorithm chooses the activities to propose to the students based on their expected reward (it is parameterized in order to decide the exploration/exploitation ratio).

Another interesting work by Kanellopoulou et. al ([48]) discusses the possibility of automatically determining the difficulty of maze-based programming challenges, with the long-term goal of automatically generating exercises at a specific level of difficulty. This is one of the possible future expansions of *Co.Thi. 2.0* (cfr. Section 2.7).

To conclude, one interesting approach to adaptive/responsive learning which is not overly examined in the literature is that of *distributed learning* (cfr. Lydia Casanova's Master's thesis, which discusses it in the context of CT [87]). In particular, there is some potential in the technique of *spaced repetition* (cfr. [88]):

- In its traditional form, *spaced repetition* consists of dividing a piece of knowledge in a series of questions and answers (combined in *flashcards*) which are proposed to students with a frequency determined by how easily they can recall the relative information.

- Classic tools, like Anki[23], require users to manually signal the difficulty they had in answering a particular question (e.g. an easy question will be seen again in a month, a difficult question in a day, etc.).

- By using LA techniques in the context of block based programming activities we could automate this process: an exercise solved quickly, with the optimal number of code blocks, amounts to an excellent solution (the same exercise can be seen again in a long time); an exercise solved with some difficulties should be proposed again in a shorter period of time.

- This mechanism can also be applied to *concepts*: we do not need to propose time and again the *same exercises*; instead, we can pull exercises from sets (or generate them) based on *category*: e.g. we could frequently propose exercises based on loops to users who have difficulty in understanding them.

[23] `https://apps.ankiweb.net/`

Appendix A

Test exercises

This Appendix shows pictures of the exercises we used for the tests described in Chapter 3. For each exercise we show the problem visualization (i.e. the "maze" or turtle graphics drawing to complete), the toolbox (i.e. the blocks available to solve the problem) and an ideal solution. For debugging exercises we also show the default solution presented by Code.org.

A.1 First grade exercises

Figures A.1, A.2, A.3 and A.4 show the test exercises for first grade students. These exercises are designed to be usable by pre-literate users and use icons to visually show the function of each block.

A.2 Fourth grade exercises

Figures A.5, A.6, A.7 and A.8 show the test exercises for fourth grade students. These exercises leave out icons and use more complex blocks which require users to do some relative spatial reasoning (e.g. "turn left", "turn right").

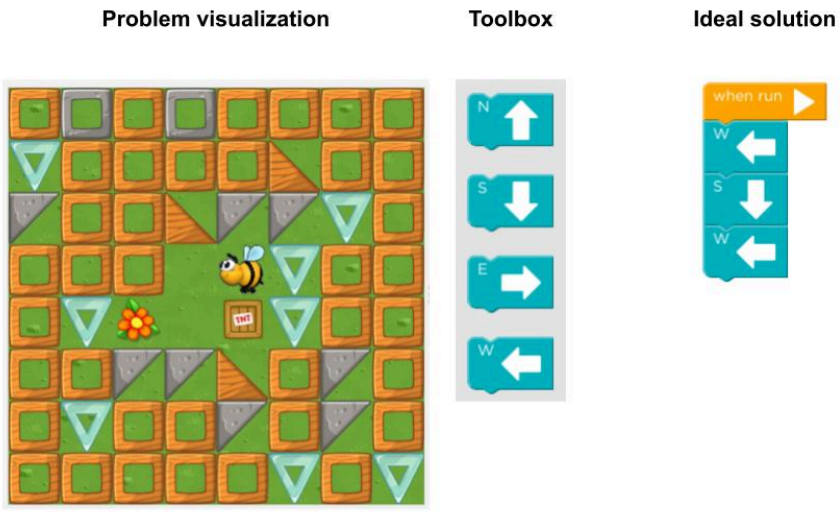

Figure A.1: Test exercise 1 for **first** grade: sequences

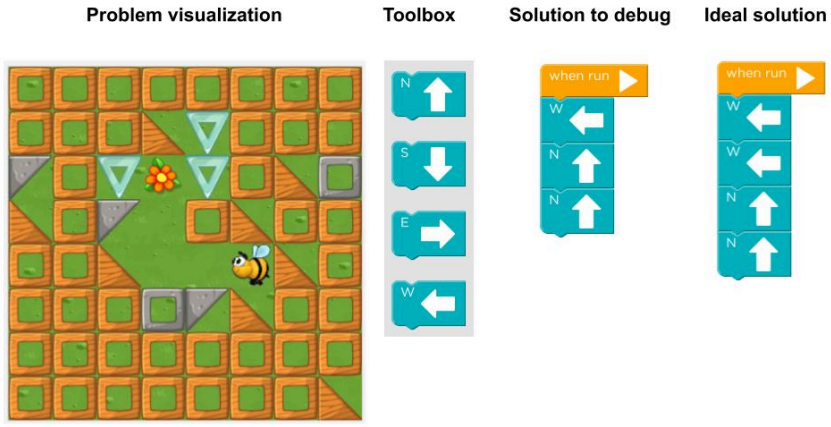

Figure A.2: Test exercise 2 for **first** grade: sequences and debugging

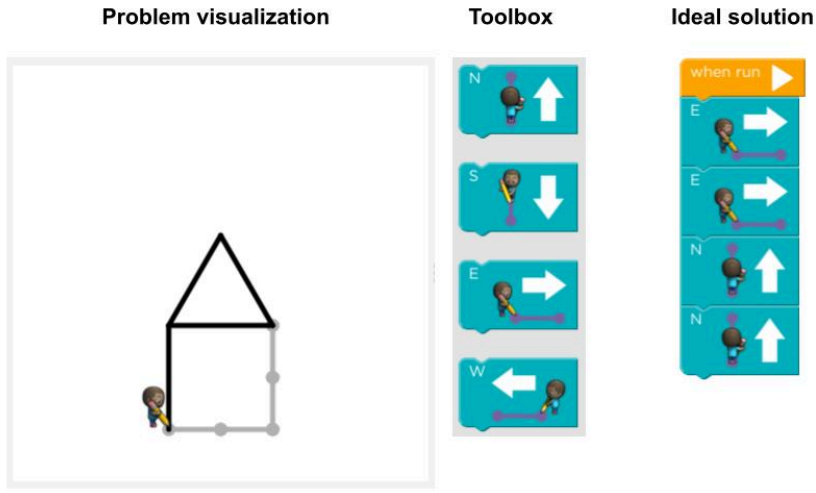

Figure A.3: Test exercise 3 for **first** grade: sequences (turtle graphics)

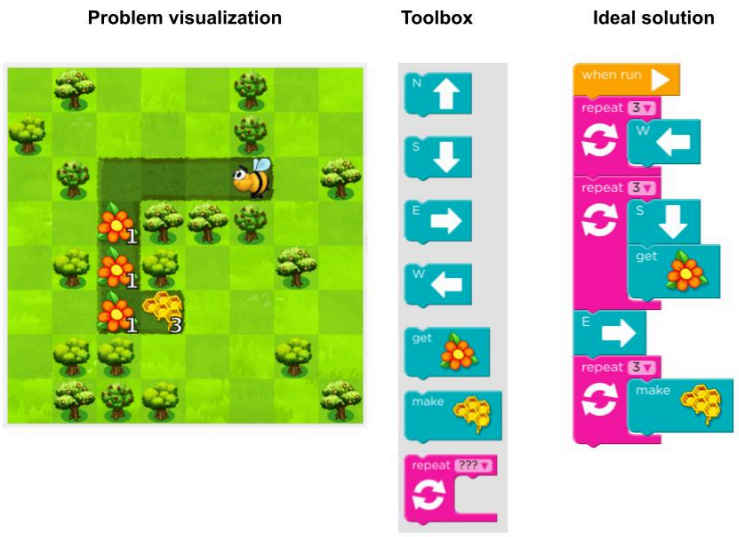

Figure A.4: Test exercise 4 for **first** grade: loops

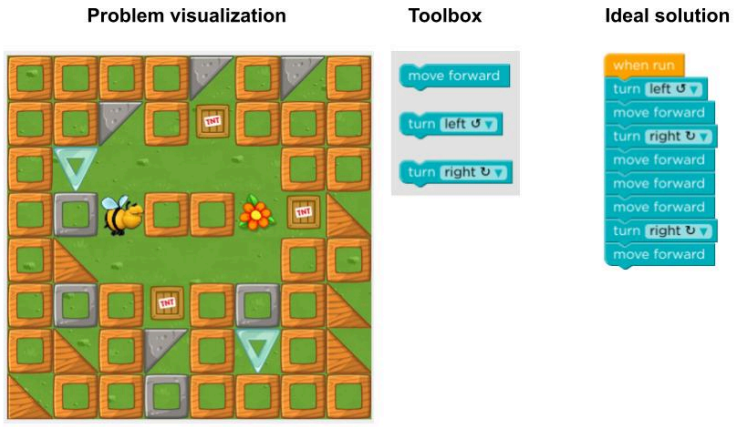

Figure A.5: Test exercise 1 for **fourth** grade: sequences

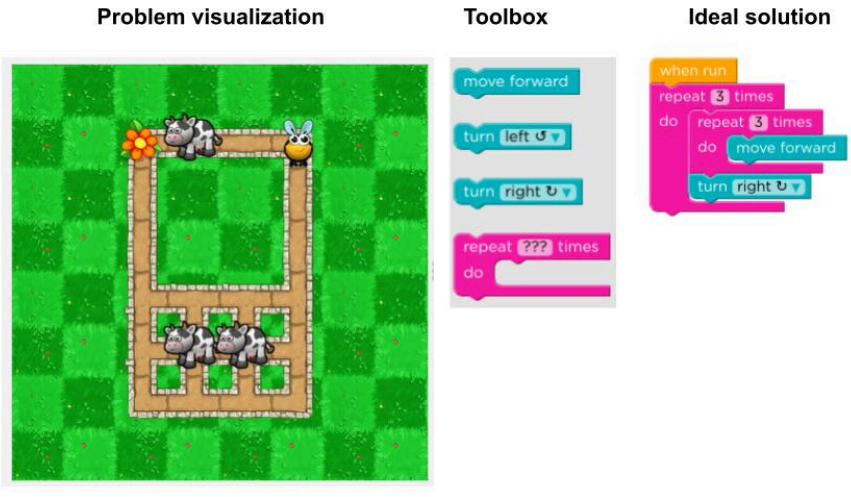

Figure A.6: Test exercise 2 for **fourth** grade: loops

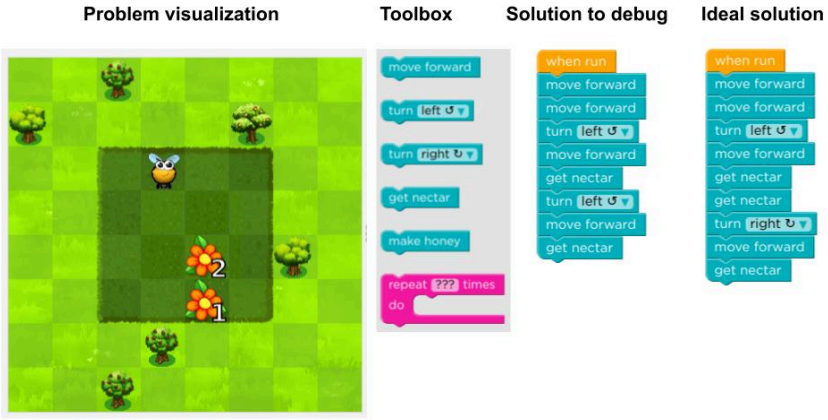

Figure A.7: Test exercise 3 for **fourth** grade: sequences and debugging

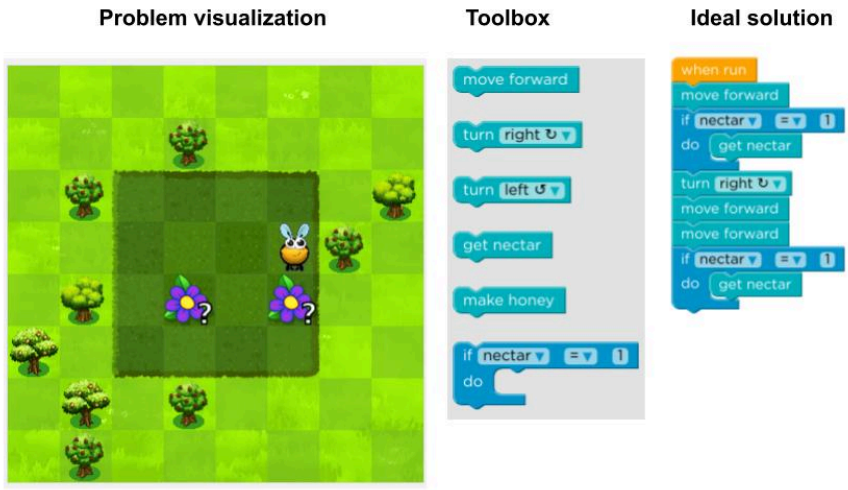

Figure A.8: Test exercise 4 for **fourth** grade: conditionals

References

Contextual background

[1] Peter J Denning and Paul S Rosenbloom. "The profession of IT Computing: the fourth great domain of science". In: *Communications of the ACM* 52.9 (2009), pp. 27–29 (cit. on pp. 3, 10).

[2] Barbara Arfé et al. "Coding in primary grades boosts children's executive functions". In: *Frontiers in psychology* 10 (2019), p. 2713 (cit. on pp. 4, 19, 23, 30, 47).

[3] Barbara Arfé, Tullio Vardanega, and Lucia Ronconi. "The effects of coding on children's planning and inhibition skills". In: *Computers & Education* 148 (2020), p. 103807 (cit. on pp. 4, 18, 19, 23, 30, 47, 53, 68).

[4] Seymour Papert. "Mindstorms: Computers, children, and powerful ideas". In: *NY: Basic Books* 255 (1980) (cit. on pp. 5, 16).

[5] Jeannette M Wing. "Computational thinking". In: *Communications of the ACM* 49.3 (2006), pp. 33–35 (cit. on pp. 5, 11).

[6] Marcos Román-González, Juan-Carlos Pérez-González, and Carmen Jiménez-Fernández. "Which cognitive abilities underlie computational thinking? Criterion validity of the Computational Thinking Test". In: *Computers in Human Behavior* 72 (2017), pp. 678–691. ISSN: 0747-5632. DOI: https://doi.org/10.1016/j.chb.2016.08.047. URL: https://www.sciencedirect.com/science/article/pii/S0747563216306185 (cit. on p. 5).

[7] Jeanette Wing. "Research notebook: Computational thinking—What and why". In: *The link magazine* 6 (2011), pp. 20–23 (cit. on p. 6).

[8] Alfred V Aho. "Computation and computational thinking". In: *The computer journal* 55.7 (2012), pp. 832–835 (cit. on p. 6).

[10] Karen Brennan and Mitchel Resnick. “New frameworks for studying and assessing the development of computational thinking”. In: *Proceedings of the 2012 annual meeting of the American educational research association, Vancouver, Canada*. Vol. 1. 2012, p. 25 (cit. on p. 6).

[11] Enrico Nardelli. “Do we really need computational thinking?” In: *Communications of the ACM* 62.2 (2019), pp. 32–35 (cit. on pp. 5, 7).

[13] Valerie J Shute, Chen Sun, and Jodi Asbell-Clarke. “Demystifying computational thinking”. In: *Educational Research Review* 22 (2017), pp. 142–158 (cit. on pp. 7, 11).

[14] Barbara Arfé and Tullio Vardanega. “Imparare a ragionare: il ruolo del pensiero computazionale a scuola”. In: *Giornale italiano di psicologia* 46.4 (2019), pp. 765–770 (cit. on p. 7).

[16] Michael Lodi. “Introducing Computational Thinking in K-12 Education: Historical, Epistemological, Pedagogical, Cognitive, and Affective Aspects”. PhD thesis. Dipartimento di Informatica-Scienza e Ingegneria, Alma Mater Studiorum . . ., 2020 (cit. on p. 8).

[17] Yuri Nesen, Brian Fowler, and Emiliana Vegas. “How Italy implemented its computer science education program”. In: (2021) (cit. on pp. 8–10).

[18] Isabella Corradini, Michael Lodi, and Enrico Nardelli. “Computational Thinking in Italian Schools: antitative Data and Teachers’ Sentiment Analysis a er Two Years of “Programma il Futuro” Project”. In: (2017) (cit. on p. 9).

[19] David Bau et al. “Learnable programming: blocks and beyond”. In: *Communications of the ACM* 60.6 (2017), pp. 72–80 (cit. on pp. 9, 13–15).

[20] Mitchel Resnick et al. “Scratch: programming for all”. In: *Communications of the ACM* 52.11 (2009), pp. 60–67 (cit. on p. 9).

[21] Caitlin Duncan, Tim Bell, and Steve Tanimoto. “Should your 8-year-old learn coding?” In: *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*. 2014, pp. 60–69 (cit. on pp. 10, 14).

[22] Michael E Caspersen et al. *Informatics for all the strategy*. ACM, 2018 (cit. on p. 10).

[23] David Barr, John Harrison, and Leslie Conery. “Computational thinking: A digital age skill for everyone.” In: *Learning & Leading with Technology* 38.6 (2011), pp. 20–23 (cit. on p. 11).

[24] Marina Umaschi Bers et al. “Computational thinking and tinkering: Exploration of an early childhood robotics curriculum”. In: *Computers & Education* 72 (2014), pp. 145–157 (cit. on p. 11).

[25] National Research Council et al. *Report of a workshop on the pedagogical aspects of computational thinking*. National Academies Press, 2011 (cit. on p. 11).

[26] Nicole D Anderson. “A call for computational thinking in undergraduate psychology”. In: *Psychology Learning & Teaching* 15.3 (2016), pp. 226–234 (cit. on pp. 11, 12).

[27] Xiaodan Tang et al. “Assessing computational thinking: A systematic review of empirical studies”. In: *Computers & Education* 148 (2020), p. 103798 (cit. on p. 12).

[28] Shuchi Grover and Satabdi Basu. “Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic”. In: *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education*. 2017, pp. 267–272 (cit. on pp. 12, 15).

[29] Marcos Román-González, Juan-Carlos Pérez-González, and Carmen Jiménez-Fernández. “Which cognitive abilities underlie computational thinking? Criterion validity of the Computational Thinking Test”. In: *Computers in human behavior* 72 (2017), pp. 678–691 (cit. on p. 12).

[30] David Weintrop. “Block-based programming in computer science education”. In: *Communications of the ACM* 62.8 (2019), pp. 22–25 (cit. on p. 13).

[31] Georgios Fessakis, Evangelia Gouli, and Elisavet Mavroudi. “Problem solving by 5–6 years old kindergarten children in a computer programming environment: A case study”. In: *Computers & Education* 63 (2013), pp. 87–97 (cit. on p. 14).

[32] Kyu Han Koh et al. “Towards the automatic recognition of computational thinking for adaptive visual language learning”. In: *2010 IEEE symposium on visual languages and human-centric computing*. IEEE. 2010, pp. 59–66 (cit. on pp. 14, 26).

[33] Colleen M Lewis. “How programming environment shapes perception, learning and goals: logo vs. scratch”. In: *Proceedings of the 41st ACM technical symposium on Computer science education*. 2010, pp. 346–350 (cit. on p. 14).

[34] Neil CC Brown et al. “Panel: Future directions of block-based programming”. In: *Proceedings of the 47th ACM technical symposium on computing science education*. 2016, pp. 315–316 (cit. on pp. 14, 15).

[35] David Weintrop and Uri Wilensky. “To block or not to block, that is the question: students’ perceptions of blocks-based programming”. In: *Proceedings of the 14th international conference on interaction design and children*. 2015, pp. 199–208 (cit. on pp. 14, 15).

[36] David Weintrop and Uri Wilensky. “Comparing block-based and text-based programming in high school computer science classrooms”. In: *ACM Transactions on Computing Education (TOCE)* 18.1 (2017), pp. 1–25 (cit. on p. 14).

[37] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. “Habits of programming in scratch”. In: *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. 2011, pp. 168–172 (cit. on pp. 14, 15).

[38] Dale Parsons and Patricia Haden. “Programming osmosis: Knowledge transfer from imperative to visual programming environments”. In: *Procedings of The Twentieth Annual NACCQ Conference*. Hamilton New Zealand. 2007, pp. 209–215 (cit. on p. 15).

[39] Betsy DiSalvo. “Graphical qualities of educational technology: Using drag-and-drop and text-based programs for introductory computer science”. In: *IEEE computer graphics and applications* 34.6 (2014), pp. 12–15 (cit. on p. 15).

[40] Edith Ackermann. “Piaget’s constructivism, Papert’s constructionism: What’s the difference”. In: *Future of learning group publication* 5.3 (2001), p. 438 (cit. on p. 15).

[41] Yannis Papadopoulos and Stergios Tegos. “Using microworlds to introduce programming to novices”. In: *2012 16th Panhellenic Conference on Informatics*. IEEE. 2012, pp. 180–185 (cit. on p. 16).

[42] Byron Weber Becker. “Teaching CS1 with karel the robot in Java”. In: *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*. 2001, pp. 50–54 (cit. on pp. 16, 17).

[43] Cynthia J Solomon and Seymour Papert. “A case study of a young child doing Turtle Graphics in LOGO”. In: *Proceedings of the June 7-10, 1976, national computer conference and exposition*. 1976, pp. 1049–1056 (cit. on pp. 16, 17).

[44] Helen Boulton et al. “The role of game jams in developing informal learning of computational thinking: a cross-European case study”. In: *arXiv preprint arXiv:1805.04458* (2018) (cit. on p. 16).

[45] Jesús Moreno-León, Gregorio Robles, and Marcos Román-González. “Dr. Scratch: Automatic analysis of scratch projects to assess and foster computational thinking”. In: *RED. Revista de Educación a Distancia* 46 (2015), pp. 1–23 (cit. on pp. 16, 23, 25).

[46] Conor Linehan et al. “Learning curves: analysing pace and challenge in four successful puzzle games”. In: *Proceedings of the first ACM SIGCHI annual symposium on Computer-human interaction in play*. 2014, pp. 181–190 (cit. on p. 16).

[47] Tomáš Effenberger and Radek Pelánek. “Towards making block-based programming activities adaptive”. In: *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*. 2018, pp. 1–4 (cit. on pp. 17, 23, 40, 72).

[48] Ioanna Kanellopoulou, Pablo Garaizar, and Mariluz Guenaga. “First Steps Towards Automatically Defining the Difficulty of Maze-Based Programming Challenges”. In: *IEEE Access* 9 (2021), pp. 64211–64223 (cit. on pp. 17, 74).

[49] Akira Miyake and Naomi P Friedman. “The nature and organization of individual differences in executive functions: Four general conclusions”. In: *Current directions in psychological science* 21.1 (2012), pp. 8–14 (cit. on p. 18).

[50] Sam J Gilbert and Paul W Burgess. “Executive function”. In: *Current Biology* 18.3 (2008), R110–R114 (cit. on p. 18).

[51] John R Best and Patricia H Miller. “A developmental perspective on executive function”. In: *Child development* 81.6 (2010), pp. 1641–1660 (cit. on p. 18).

[52] Adele Diamond. “Executive functions”. In: *Annual review of psychology* 64 (2013), pp. 135–168 (cit. on p. 18).

[53] Judy Robertson et al. “The relationship between executive functions and computational thinking”. In: *International Journal of Computer Science Education in Schools* 3.4 (2020), pp. 35–49 (cit. on pp. 20, 23).

[54] Maria Chiara Di Lieto et al. “Educational Robotics intervention on Executive Functions in preschool children: A pilot study”. In: *Computers in human behavior* 71 (2017), pp. 16–23 (cit. on pp. 20, 23).

[55] Maria Chiara Di Lieto et al. “Empowering executive functions in 5-and 6-year-old typically developing children through educational robotics: An RCT study”. In: *Frontiers in psychology* 10 (2020), p. 3084 (cit. on pp. 20, 23).

[56] Timothy Shallice. “Specific impairments of planning”. In: *Philosophical Transactions of the Royal Society of London. B, Biological Sciences* 298.1089 (1982), pp. 199–209 (cit. on p. 20).

[57] Phillip Long. *LAK’11: Proceedings of the 1st International Conference on Learning Analytics and Knowledge, February 27-March 1, 2011, Banff, Alberta, Canada*. ACM, 2011 (cit. on p. 22).

[58] Ryan S Baker, Dragan Gašević, and Shamya Karumbaiah. “Four paradigms in learning analytics: Why paradigm convergence matters”. In: *Computers and Education: Artificial Intelligence* 2 (2021), p. 100021 (cit. on pp. 22, 23).

[59] Daniel Amo Filvà et al. “Clickstream for learning analytics to assess students’ behavior with Scratch”. In: *Future Generation Computer Systems* 93 (2019), pp. 673–686 (cit. on pp. 22, 23).

[60] Fatma Gizem Karaoglan Yilmaz and Ramazan Yilmaz. “Learning Analytics Intervention Improves Students’ Engagement in Online Learning”. In: *Technology, Knowledge and Learning* (2021), pp. 1–12 (cit. on pp. 22, 23).

[61] Yi-Shan Tsai et al. “Learning analytics in European higher education—Trends and barriers”. In: *Computers & Education* 155 (2020), p. 103933 (cit. on p. 23).

[62] Anupam Khan and Soumya K Ghosh. “Student performance analysis and prediction in classroom learning: A review of educational data mining studies”. In: *Education and information technologies* 26.1 (2021), pp. 205–240 (cit. on p. 23).

[63] Meltem Tutar, Austin Wang, and Gulsen Kutluoglu. “Personalized Lecture Recommendations to Facilitate Bite-Sized Learning”. In: *Proceedings of the Eighth ACM Conference on Learning@ Scale*. 2021, pp. 359–362 (cit. on p. 23).

[64] Adelmo ELOY et al. “A data-driven approach to assess computational thinking concepts based on learners’ artifacts”. In: *Informatics in Education* (2021) (cit. on p. 23).

[65] Sofia Papavlasopoulou, Michail N Giannakos, and Letizia Jaccheri. “Discovering children’s competences in coding through the analysis of Scratch projects”. In: *2018 IEEE Global Engineering Education Conference (EDUCON)*. IEEE. 2018, pp. 1127–1133 (cit. on p. 23).

[66] Deborah A Fields et al. “Combining big data and thick data analyses for understanding youth learning trajectories in a summer coding camp”. In: *Proceedings of the 47th ACM technical symposium on computing science education*. 2016, pp. 150–155 (cit. on pp. 23, 25).

[67] Max Kesselbacher and Andreas Bollin. “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*. 2019, pp. 1–10 (cit. on pp. 23, 25).

[68] Andoni Eguiluz et al. “Exploring the progression of early programmers in a set of computational thinking challenges via clickstream analysis”. In: *IEEE Transactions on Emerging Topics in Computing* 8.1 (2017), pp. 256–261 (cit. on p. 23).

[69] Tomáš Effenberger and Radek Pelánek. “Measuring Students’ Performance on Programming Tasks”. In: *Proceedings of the Sixth (2019) ACM Conference on Learning@ Scale*. 2019, pp. 1–4 (cit. on pp. 23, 73).

[70] Radek Pelánek. “Exploring the utility of response times and wrong answers for adaptive learning”. In: *Proceedings of the fifth annual ACM conference on learning at scale*. 2018, pp. 1–4 (cit. on p. 23).

[71] Benjamin Clement et al. “Multi-armed bandits for intelligent tutoring systems”. In: *arXiv preprint arXiv:1310.3174* (2013) (cit. on pp. 23, 73).

[72] Giovanni Maria Troiano et al. “Is my game OK Dr. Scratch? Exploring programming and computational thinking development via metrics in student-designed serious games for STEM”. In: *Proceedings of the 18th ACM international conference on interaction design and children*. 2019, pp. 208–219 (cit. on p. 26).

[73] Neil Selwyn. “Re-imagining ‘Learning Analytics’. . . a case for starting again?” In: *The Internet and Higher Education* 46 (2020), p. 100745 (cit. on p. 26).

Outline of the research project

[2] Barbara Arfé et al. “Coding in primary grades boosts children’s executive functions”. In: *Frontiers in psychology* 10 (2019), p. 2713 (cit. on pp. 4, 19, 23, 30, 47).

[3] Barbara Arfé, Tullio Vardanega, and Lucia Ronconi. “The effects of coding on children’s planning and inhibition skills”. In: *Computers & Education* 148 (2020), p. 103807 (cit. on pp. 4, 18, 19, 23, 30, 47, 53, 68).

[47] Tomáš Effenberger and Radek Pelánek. “Towards making block-based programming activities adaptive”. In: *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*. 2018, pp. 1–4 (cit. on pp. 17, 23, 40, 72).

Experimental evaluation

[2] Barbara Arfé et al. “Coding in primary grades boosts children’s executive functions”. In: *Frontiers in psychology* 10 (2019), p. 2713 (cit. on pp. 4, 19, 23, 30, 47).

[3] Barbara Arfé, Tullio Vardanega, and Lucia Ronconi. “The effects of coding on children’s planning and inhibition skills”. In: *Computers & Education* 148 (2020), p. 103807 (cit. on pp. 4, 18, 19, 23, 30, 47, 53, 68).

[74] Anthony J Bishara and James B Hittner. “Testing the significance of a correlation with nonnormal data: comparison of Pearson, Spearman, transformation, and resampling approaches.” In: *Psychological methods* 17.3 (2012), p. 399 (cit. on p. 54).

[75] Eric Langford, Neil Schwertman, and Margaret Owens. “Is the property of being positively correlated transitive?” In: *The American Statistician* 55.4 (2001), pp. 322–325 (cit. on p. 57).

Conclusions

[3] Barbara Arfé, Tullio Vardanega, and Lucia Ronconi. “The effects of coding on children’s planning and inhibition skills”. In: *Computers & Education* 148 (2020), p. 103807 (cit. on pp. 4, 18, 19, 23, 30, 47, 53, 68).

[47] Tomáš Effenberger and Radek Pelánek. “Towards making block-based programming activities adaptive”. In: *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*. 2018, pp. 1–4 (cit. on pp. 17, 23, 40, 72).

[48] Ioanna Kanellopoulou, Pablo Garaizar, and Mariluz Guenaga. “First Steps Towards Automatically Defining the Difficulty of Maze-Based Programming Challenges”. In: *IEEE Access* 9 (2021), pp. 64211–64223 (cit. on pp. 17, 74).

[69] Tomáš Effenberger and Radek Pelánek. “Measuring Students’ Performance on Programming Tasks”. In: *Proceedings of the Sixth (2019) ACM Conference on Learning@ Scale*. 2019, pp. 1–4 (cit. on pp. 23, 73).

[71] Benjamin Clement et al. “Multi-armed bandits for intelligent tutoring systems”. In: *arXiv preprint arXiv:1310.3174* (2013) (cit. on pp. 23, 73).

[76] Neil Fraser. “Ten things we’ve learned from Blockly”. In: *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. IEEE. 2015, pp. 49–50 (cit. on p. 67).

[77] Erik Pasternak, Rachel Fenichel, and Andrew N Marshall. “Tips for creating a block language with blockly”. In: *2017 IEEE blocks and beyond workshop (B&B)*. IEEE. 2017, pp. 21–24 (cit. on p. 67).

[78] Sofia Papavlasopoulou et al. “Using eye-tracking to unveil differences between kids and teens in coding activities”. In: *proceedings of the 2017 conference on interaction design and children*. 2017, pp. 171–181 (cit. on p. 69).

[79] Sofia Papavlasopoulou, Kshitij Sharma, and Michail N Giannakos. “How do you feel about learning to code? Investigating the effect of children’s attitudes towards coding using eye-tracking”. In: *International Journal of Child-Computer Interaction* 17 (2018), pp. 50–60 (cit. on p. 69).

[80] Sofia Papavlasopoulou, Kshitij Sharma, and Michail N Giannakos. "Coding activities for children: Coupling eye-tracking with qualitative data to investigate gender differences". In: *Computers in Human Behavior* 105 (2020), p. 105939 (cit. on p. 70).

[81] Alexandra Papoutsaki et al. "WebGazer: Scalable Webcam Eye Tracking Using User Interactions". In: *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI. 2016, pp. 3839–3845 (cit. on p. 71).

[82] Monica Bulger. "Personalized learning: The conversations we're not having". In: *Data and Society* 22.1 (2016), pp. 1–29 (cit. on p. 72).

[83] Ashok R Basawapatna et al. "The zones of proximal flow: guiding students through a space of computational thinking skills and challenges". In: *Proceedings of the ninth annual international ACM conference on International computing education research*. 2013, pp. 67–74 (cit. on p. 72).

[84] Tomáš Effenberger. "Adaptive system for learning programming". PhD thesis. Master's thesis, Masaryk University, 2018 (cit. on p. 72).

[85] Tomáš Effenberger. "Adaptive Learning of Programming". In: () (cit. on p. 72).

[86] Benjamin Clement. "Adaptive personalization of pedagogical sequences using machine learning". PhD thesis. Université de Bordeaux, 2018 (cit. on p. 73).

[87] Lydia Casanova De Vilalta. "Effects of distributed learning patterns on elementary student learning of computational thinking". In: (2020) (cit. on p. 74).

[88] Sean HK Kang. "Spaced repetition promotes efficient and effective learning: Policy implications for instruction". In: *Policy Insights from the Behavioral and Brain Sciences* 3.1 (2016), pp. 12–19 (cit. on p. 74).

Online references

Contextual background

[9] CSTA & ISTE (2011). *Operational Definition of Computational Thinking for K–12 Education*. URL: https://cdn.iste.org/www-root/ct-documents/computational-thinking-operational-definition-flyer.pdf (cit. on p. 6).

[12] Jeanette Wing. *Computational Thinking: What and Why? (2010)*. URL: https://www.cs.cmu.edu/link/research-notebook-computational-thinking-what-and-why (cit. on p. 6).

[15] The Royal Society. *Shut down or restart? The way forward for computing in UK schools (2012)*. URL: https://royalsociety.org/-/media/education/computing-in-schools/2012-01-12-computing-in-schools.pdf (cit. on p. 7).