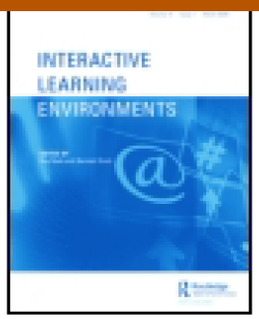

Interactive Learning Environments

ISSN: 1049-4820 (Print) 1744-5191 (Online) Journal homepage: https://www.tandfonline.com/loi/nile20

Algotaurus: an educational computer programming game for beginners

Attila Krajcsi, Csaba Csapodi & Eleonóra Stettner

To cite this article: Attila Krajcsi, Csaba Csapodi & Eleonóra Stettner (2019): Algotaurus: an educational computer programming game for beginners, Interactive Learning Environments, DOI: 10.1080/10494820.2019.1593862

To link to this article: https://doi.org/10.1080/10494820.2019.1593862

INTERACTIVE LEARNING ENVIRONMENTS
https://doi.org/10.1080/10494820.2019.1593862

Routledge
Taylor & Francis Group

Algotaurus: an educational computer programming game for beginners

Attila Krajcsi[a], Csaba Csapodi[b] and Eleonóra Stettner[c]

[a]Department of Cognitive Psychology, Institute of Psychology, ELTE Eötvös Loránd University, Budapest, Hungary; [b]Mathematics Teaching and Education Center, Institute of Mathematics, ELTE Eötvös Loránd University, Budapest, Hungary; [c]Department of Mathematics and Informatics, Kaposvár University, Kaposvár, Hungary

An educational computer game is presented, used for beginner students to introduce some basic concepts of code execution and code writing. In this mini-language microworld game, a code should be written with which a robot can escape from a procedurally generated labyrinth. The game uses a simple language and utilizes a virtual environment, where code execution could be tracked easily. One essential advantage of the software is that after a very short training, students can start experimenting, and they can understand many basic properties of code writing and execution. Based on several pilot teaching classes in both primary schools and universities, the game is an efficient tool to introduce the bases of computer programming, which bases might be harder to demonstrate with other educational tools.

ARTICLE HISTORY
Received 21 August 2018
Accepted 8 March 2019

KEYWORDS
Computer programming; educational game; microworld; mini-language; AlgoTaurus

Computer programming is getting a more and more important skill in the twenty-first century, and computer programming education is becoming more and more essential (see, for example, Futschek & Moschitz, 2011; Kayama et al., 2014; Maloney et al., 2004; Papert, 1993). There are many educational tools with different properties and aims (e.g. Good & Howland, 2017; Maloney et al., 2004; Papert, 1993), and some works has already started to measure the efficiency of those tools (Chen et al., 2017; Chen, Yan, Yang, & Lin, 2005; Fagin & Merkle, 2003; Howland & Good, 2015; Marcelino, Pessoa, Vieira, Salvador, & Mendes, 2018; Papastergiou, 2009; Robins, Rountree, & Rountree, 2003; Tekerek & Altan, 2014; Xinogalos, Satratzemi, & Malliarakis, 2017).

When creating an educational tool, probably all educator and researcher would want to build upon established models of cognitive development, individual differences or learning methods. Unfortunately, for computer programming (and for many subject areas) this is hardly possible. In the case of computational thinking, it is not really understood what cognitive components contribute to computational thinking (Román-González, Pérez-González, & Jiménez-Fernández, 2017), what tests could measure computational thinking in a valid and reliable way (e.g. see Román-González, Pérez-González, Moreno-León, & Robles, 2018) or what computational thinking is at all (e.g. Voogt, Fisser, Good, Mishra, & Yadav, 2015). Because theoretical models cannot offer a solid background, most educational tools rely on the intuition of educators, and many times efficiency measurements offer results that cannot be entirely explained by detailed models. In line with this general problem, the present work relies on the experience with former educational tools, and on observations of educators.

CONTACT Attila Krajcsi krajcsi@gmail.com

While teaching absolute beginners, one critical task is to make some basic rules understandable, such as, the script is executed line by line, the computer relies purely on the code, how the control flow can be used, and so on. Although these basic concepts can be taught with any computer programming language, one group of teaching software appropriate for this aim is the mini-language microworld programming environment, which is a very simple computer language applied in a simple virtual environment, such as the classic Logo with turtle graphics (Papert, 1993). We found this type of educational tool especially useful in cognitive psychology master program,[1] because programming an agent, e.g. a robot is quite close to the thinking of cognitive researchers who try to reverse engineer the mind (Pinker, 1999), and because various optimization issues that are also essential in cognitive sciences can be introduced very efficiently. (See some additional examples why cognitive science education requires further considerations for teaching computer programming, in section Special case study: AlgoTaurus in cognitive psychology major classes.) While there are several microworld programming environments for education purposes, there are important limitations in those systems: some of them are expensive (e.g. the systems that utilize real physical robots, such as the Lego Mindstorms), or even if they are not expensive, they are not free (e.g. several proprietary software), or the language or the environment are not appropriate to demonstrate some essential features. One microworld software that we used successfully in cognitive psychology master programs was the Labirint software, but it also had some important limitations: For example, it was written to run in MS-DOS (therefore, emulator was needed to run on Windows machines), stepwise execution was not available, and in general, because it was a closed source software, it was impossible to extend it. To overcome these limitations, we made a free and open source remake and extension of the Labirint software, called AlgoTaurus.

First, we present the AlgoTaurus software and its main features where we compare it with some other educational tools. Then, we propose some instructions how AlgoTaurus can be used in classrooms. Finally, we present our experiences using AlgoTaurus in primary schools and in higher educations, adding a specific case study how AlgoTaurus can be used in cognitive psychology education and how it can be fit into other computer programming education steps.

1. What is AlgoTaurus?

AlgoTaurus is a mini-language microworld programming environment: a simple computer language (with only seven available statements) used in a simple virtual environment (see Figure 1). In the environment, a random labyrinth is generated (procedural generation) and a robot, called AlgoTaurus, is placed randomly. The task of the user is to write a script that can help AlgoTaurus to find the exit. While executing the code, the user can track (a) what line is executed, and (b) where the robot is in the labyrinth. The software is similar to other mini-language microworlds (see an excellent overview of these educational tools in Brusilovsky, Calabrese, Hvorecky, Kouchnirenko, & Miller, 1997), although the AlgoTaurus language is even simpler than the languages in those former tools.

The main advantage of AlgoTaurus is that even if it does not require any previous experience with computer programming, after a few minutes of instruction, students can start creating code, building some relatively complex solutions, while many basic properties of code execution and code writing and testing can be learned. This knowledge is then easily transferable to any other computer language the student will learn later. AlgoTaurus can be used either with children or with adults. According to our test classes, children as young as 10 years of age and young adult university students can also use it.

Note that there are other types of educational tools teaching computer programming, one popular example is the block-based Scratch environment (*Scratch*, n.d.). See some considerations how different tools fulfill different needs and how they can be selected or combined in education, in section Next steps and additional tasks.

AlgoTaurus is an open source software (written in Python), freely downloadable from its website at https://github.com/AlgoTaurus/algotaurus On Windows, AlgoTaurus can be installed with a simple

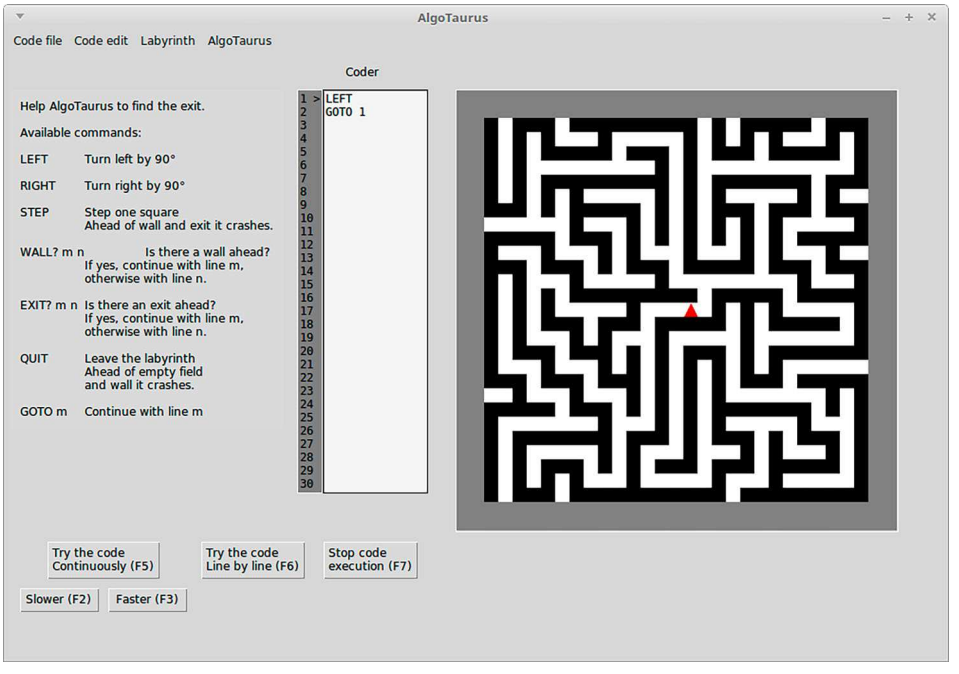

Figure 1. Screenshot of AlgoTaurus.

installer, and it can be launched from the Start menu. On Linux, the source code can be downloaded and run – find simple instructions on the website of AlgoTaurus.

AlgoTaurus is a remake and an extension of a similar DOS program, called Labirint (Lukyanov & Volkov, 1993). Compared to the Labirint program, AlgoTaurus includes several new features that are important in classroom use: The help text is continuously visible on the main screen, the code can be executed stepwise, the running speed can be modified, there is a more simple labyrinth type to teach programming more gradually, the code editor is more advanced, the software is cross-platform, the code is openly available, so anyone can change or extend it, etc.

1.1. Main features of AlgoTaurus

There are several components contributing to the efficiency of AlgoTaurus for beginners compared to other computer languages and educational environments. These factors partly explain why a short introduction is sufficient even for absolute beginners, and how the students may understand basic concepts sometimes even implicitly. All these features are in line with the recommendations about successful mini-language microworlds (Brusilovsky et al., 1997). (Even if the paper by Brusilovsky and his colleagues is more than 20 years old, we found that paper to be the most comprehensive work about the critical features of mini-language microworlds and about their evaluation.) This section also describes partially some of our experiences in pilot training sessions.

Small mini-language. AlgoTaurus includes only seven statements (LEFT, RIGHT, STEP, WALL?, EXIT? QUIT and GOTO – these are the statements originally found in the Labirint software), which can be explained in minutes to the students, and they can understand and learn them rather quickly. This small set of statements is even smaller than the usual number of available statements in a mini-language (Brusilovsky et al., 1997). Additionally, while visual programming languages are recommended for beginners, because code builders do not have to carefully consider syntactic

issues, the small language used in AlgoTaurus makes the use of syntactic rules easy, and typically it does not cause syntactic issues.

Easy to understand statements. Four of the statements (LEFT, RIGHT, STEP and EXIT) are very intuitive to the students, and hardly any extra explanations are needed. The rest of the statements (WALL?, EXIT? and GOTO) are related to the very simple control flow, and they could be introduced together with control flow concepts, while the test of the special conditionals (WALL? and EXIT?) are again intuitive, and easy to understand.

Lack of variables. While this feature precludes that variables could be taught with this game, it enables the user to focus on some other basic concepts, such as the control flow, the state of the computer, and so on. It is possible to imagine an extension of the current language where variables are also used, and additional tasks can be added to the software.

"Wild west" of control flow. Many modern computer languages do not include a GOTO statement, because the structure of the code would be hard to track for the programmer. The AlgoTaurus language includes GOTO statement, and together with the conditional WALL? and EXIT? commands one can create condition-controlled loops even with complex structures. While this language enables solutions that are not preferred in computer programming, it can be useful for absolute beginners, because it enables a flexible flow control with only a few statements. Additionally, users can see control flow in a more rough version, and it is also demonstrated why GOTO statement is avoidable (students many times have difficulties to interpret their solution or to read other's code, and one main reason behind that is the GOTO statement). Also, one can imagine that in a modified version of the language, there could be language elements for loops and conditionals, and GOTO could not be used. This is an open possibility to modify AlgoTaurus.

Simultaneous code execution and state tracking. While running the code, the actual line that is executed is highlighted (as in debugging functions of IDEs), and the robot's position and direction are visible in the labyrinth at the same time (similar to the watch function in IDEs, whereas all variables – in fact, the state of the robot and the labyrinths – are tracked here displayed visually). Code can also be executed stepwise, so it can be investigated carefully how the code directs the robot. These are new features of AlgoTaurus compared to the Labirint software, and they are in line with the recommendations of efficient microworld environment use (Brusilovsky et al., 1997).

Easy to understand task. Users can easily understand the goal of the task (i.e. to find the exit of the labyrinth), and they understand the building blocks with which they can work (i.e. the language and the execution of code), so they can start experimenting after a very brief introduction. No long explanation is needed before starting to write some code, a few minutes introduction to the task is usually sufficient.

Carefully selected unambiguous terms. When teaching computer programming, experts take it for granted what running the code, step and other terms mean. However, these are not trivial for beginners, and they have difficulties to understand that both the robot and the code are running, or both the robot and the code could step. In most cases, the user interface does not include the movement and other area related metaphors that are used for code execution. For example, instead of "Run the code", "Try the code – Continuously" is used. Several other ambiguities were avoided. For example, the Help panel does not include the "WALL? *x y*" text, denoting the two line numbers with variable *x* and *y*, but uses the "WALL? *m n*" text, because *x* and *y* is often associated with coordinates, where this association would have been misleading here.

Interesting task. In most cases we observed that students find the task interesting, and compared to other tasks they are very motivated. Also, students who have already learned computer programming, found the task challenging and stimulating.

Actor based execution. In AlgoTaurus program, it is not an abstract computer that executes the code, but a robot. Additionally, all statements can be understood from the viewpoint of the robot.

Humans are evolutionary prepared to understand other people's, animals' or any actors' viewpoints (for example see an accessible description in Gopnik, Meltzoff, & Kuhl, 1999), and this ability is utilized in the robot-perspective game.

In itself none of the features mentioned so far are unique, because those features can be found in other tools as well. However, to our knowledge those features are not combined the same way in any former computer programming educational tool. For example, there are several games using actor-based execution (e.g. LOGO, see Papert, 1993), or even using a robot to find an exit, but they use a more complex language which cannot be used in a single class or session (see a few of them in the continuously updated Programming games category at Wikipedia at https://en.wikipedia.org/wiki/Category:Programming_games); or in other cases, when a limited language is used for labyrinth problems (e.g. several tutorials at https://code.org/), the labyrinths are not generated procedurally (they are not random) which makes explanation of few concepts more complicated; or the language does not include any possibility to deviate from sequential execution (i.e. no conditionals, loops, etc.). Additionally, some of the alternative environments are proprietary or commercial which constrains their use or extension; or some of them include tasks that are not appropriate to demonstrate some concepts, such as optimization or how the properties of the outer environment are utilized in the code. Overall, while AlgoTaurus is mostly similar to some other mini-language microworld robot games (Brusilovsky et al., 1997), it includes even smaller language, to enable initial use by students even faster.

These properties also determine when AlgoTaurus can or cannot be efficient. Because of the limited language and environment, AlgoTaurus can be used only to introduce the first steps in computer programming, and more advanced topics could be introduced with other educational tools or with real computer languages (see some additional recommendations below). However, it is suitable to teach the basics for beginners who has no former computer programming knowledge.

2. Suggested use of AlgoTaurus

There are various considerations one should conceive when teaching new material. For example, according to Griffin (2004) in math instruction, teaching material should be built upon the children's current knowledge, it should follow the students' natural development, it should teach both conceptual understanding and computational fluency, it should provide opportunity for explorations, and it should provide examples how the material is used in non-teaching environments. These considerations also may apply to computer programming learning, although it is not trivial how to implement them. These considerations could be particularly important for younger students. Therefore, here, we propose a possible series of steps of a class using AlgoTaurus based on our previous experiences, to support the application of AlgoTaurus in the classroom environment.

Before introducing the instruction steps, it is essential to clarify that AlgoTaurus, as a tool to teach beginners, can be used to reach two aims. The main aim is that students should understand how generally a code is executed, and how this execution is reflected in the states of the computer. This is the essential aim of the use of this program. A second aim is to create an algorithm that can find the exit. Although this is an interesting puzzle, and students find the task entertaining, in fact, this is independent of the main aim, weather students understand the main rules of code execution. (This is one reason why students with former knowledge of other computer programming language also find the task interesting.) Therefore, a student can learn and understand how lines of the script are processed and how they should be created, even if he/she cannot solve the labyrinth task. If someone can solve the simple Four walls version (Figure 2, see some further details below), it is a sign that bases of code creation and code running are understood appropriately, and these bases could be acquired even if the Depth first version was not solved. Therefore, it is essential for the teachers to understand, that from the viewpoint of computer programming teaching, the main aim is that students should understand the logics of code execution and code creation (e.g. by solving the Four walls problem), but it has only secondary importance whether the Depth first version will be

solved or not. Depth first version is only used in AlgoTaurus to make the task intriguing for the students, and to implicitly explain some properties of code execution and code creation, but it can serve its purpose even if the student will not solve that puzzle.

2.1. Example class instructions

At least 60–90 min should be provided for the main class, because according to our pilot classes, the fastest students can solve the task within 30–45 min, and at that time, slower students still keep trying to solve the problem.

First, a short introduction and background story (especially useful for younger children) could be given about the task itself, while the students can have a look at the user interface of AlgoTaurus.

> Your robot, called AlgoTaurus, is captured in a labyrinth on planet Amateru. She has to find the exit fields around the labyrinth (gray area around the labyrinth) and leave planet Amateru. AlgoTaurus can safely launch her rockets only *next* to an exit field (but not *on* an exit field or inside the labyrinth). The labyrinth can change from time to time: every time before AlgoTaurus starts to find her way from the labyrinth, the labyrinth can create a new form. The memory of AlgoTaurus was erased, and only you can send a code to her, so AlgoTaurus can execute the code. AlgoTaurus can move around in the labyrinth, she has detectors to tell whether there is a wall or an exit in front of her, and she can start her engines to start her interstellar escape. Be careful, if AlgoTaurus steps into a wall or exit field, or if she starts her engine inside the labyrinth and not in front of the exit area, she will be destroyed, and never gets back home. Write the code, and help AlgoTaurus to find her way back to planet Earth.

Then the teacher can explain the user interface. On the right, one can see AlgoTaurus (red triangle) and the labyrinth. In the middle, the user can write the code. On the left, the user can find the short description of the available statements.

Finally, the teacher can explain how the code will be processed, and what statements AlgoTaurus can use. All the following points can be explained separately.

- AlgoTaurus will not be directed interactively by the user, but a list of statements should be sent to her, which she can execute independently of the user.

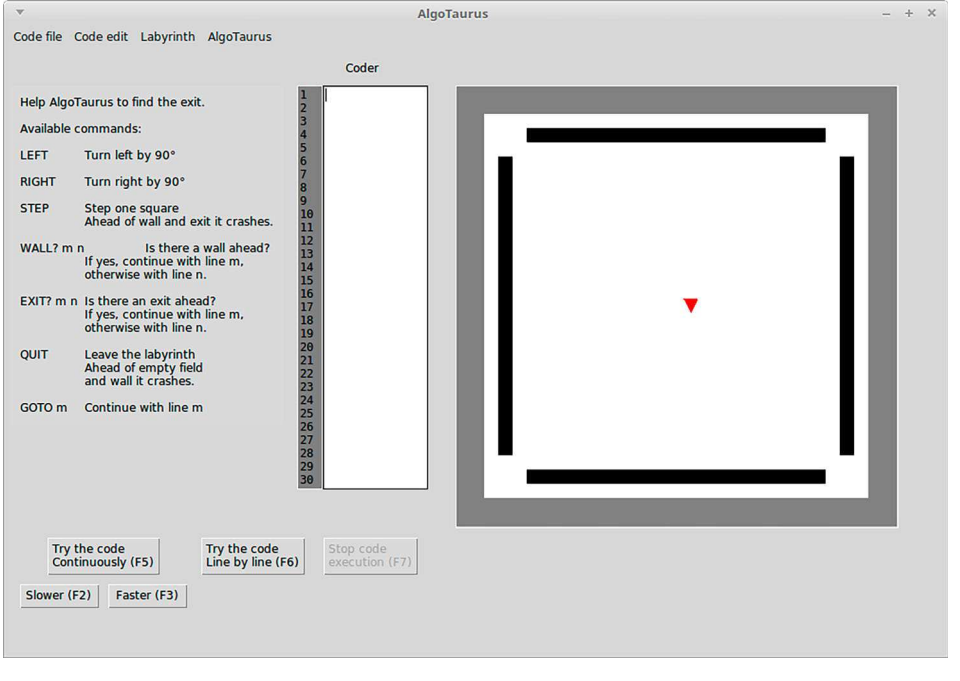

Figure 2. Four walls version of the Labyrinth.

- She understands only a few statements/commands, and no other commands can be sent to her.
- A line in the code can include only a single statement.
- The list of statements will be executed line by line, starting from the first line.
- There are some statements that can direct her to continue not on the next line, but on some other line.
- No other help can be sent to her.
- The commands AlgoTaurus can understand:

Statement	Explanation
LEFT/RIGHT	The robot turns by 90 degree (she only turns, but does not step)
STEP	Steps one square ahead (to the direction of the arrow). She can only step into an empty field, otherwise AlgoTaurus will crash.
QUIT	It starts the engines to leave the labyrinth. If it is used not in front of an exit field, AlgoTaurus will be destroyed.
GOTO *m* (where *m* is a number of a line in the code)	Usually the statements are executed one after the other. After GOTO, the execution will not be continued on the next line, but on the line given after GOTO.
WALL?/EXIT? *m n* (where *m* and *n* are numbers of a line in the code)	The statements check if there is a wall or exit field in front of AlgoTaurus. If there is a wall/exit ahead, then the code will be continued on line m, otherwise on line *n*.

- After a GOTO or WALL/EXIT command, when executing the new line, AlgoTaurus will not go back to the line after the GOTO/WALL/EXIT line, but continues after the new line she has executed.
- If the statement can include a number or numbers (GOTO/WALL/EXIT) use a space after the keyword, and between the numbers.
- Students can write the code in the middle panel, and the code can be tested continuously (and execution speed can be set faster and slower), or it can be tested line by line. Teacher may demonstrate this even with a simple code as 1: LEFT, 2: GOTO 1, and probably explaining this simple code, too.
- While executing the code, one can track what line is executed at the moment (little sign at the beginning of that line).
- The code can be edited again only if the code execution has stopped. Starting code execution again will draw a new labyrinth.

The whole explanation usually takes only a few minutes (depending on the age and some other circumstances, this could be 5–10 min). After this introduction, the students can start writing and testing the code. Writing and testing can take a lot of time, and according to our experiences, the students find the task appealing, so it is not unusual that they work longer than an hour without interruption.

As a main rule, the teacher should let the students experimenting, and not help them until they are seriously stuck (i.e. they face the same issue repeatedly, and they cannot come up with new idea to solve it for minutes). As another rule, the teacher should not encourage them to plan the code in advance, because for them, at the beginning of understanding code execution and writing, it is more important to see different scenarios, how the robot reacts to the changes of the code, even if that leads to bad syntax or crashes. So experimenting and experiencing issues is a key point of the learning process (see a similar conclusion in Fagin & Merkle, 2003). If students ask any questions, teachers can clarify the issues, unless the question is related to the algorithm to use. In latter case, teacher can encourage students to experiment with the program, and try to find the answer for themselves.

After 10–20–30 min (depending on age, motivation, etc. of the students), if someone is stuck, the teacher can suggest that the student could solve a more simple task first, the Four walls version (it can

be set from the Labyrinth menu). The task is simple enough that if someone understands how a code is executed, then he/she can write the code. Still, it can take some time, until an absolute beginner can find the solution. Some students will try to build a specific list of steps in the Four walls version (e.g. ten STEP statements, then a LEFT, then several STEP statements again, and a QUIT). This version might work sometime, but will not be appropriate all the time, because the starting point of the robot is random, and occasionally will hit the wall or exit fields, etc. In this case, the teacher can encourage the students to create a code that can solve the problem all the time, so they should come up with some more general solution. If the Four walls version is solved, student can switch back to the Depth first version. We do not recommend that students should start with Four walls version first, because the Depth first version introduces some features with which students understand that some general solution should be found, and if they started with the Four walls version, they could misunderstand the task, trying to give specific series of LEFT/RIGHT/STEP commands for a specific plan. Therefore, we recommend that the Four walls version should be used only if the Depth first version is too hard for the student.

If the students cannot solve the Four walls version for a longer time, the teacher can set a specific task for them to solve first. (1) "Write a program that makes the robot turn left twice, then turn right twice, and will make her to do the whole thing again and again." (2) "Write a program that first checks whether there is a wall ahead of her, and if yes, she should turn right, otherwise she should step one square ahead, then the whole thing should be repeated again and again."

If the students are still stuck with the Depth first labyrinth, then the teacher can hint that the students could draw various parts, corners, corridors, etc. of the labyrinth on a paper, and they can think about those configurations systematically, how the robot will behave in those places with the given code.

Another option if some students are still stuck, that they can cooperate to find solutions. They can discuss the possibilities in pairs, and they could create and test the code together in a collaborative problem-solving process.

Usually, there are very large differences between the students how much time is needed to solve the task. Among others, it depends on whether they have learned any computer programming language before, it may depend on the intellectual abilities of the students, and finding a working solution may also depend on luck (sometimes partly by random trial and error procedure they find a solution, but they do not entirely understand how the code works). When some students have solved the task, the teacher can set new tasks, listed in the next section. This might ensure that all students can work continuously in their own pace. If some students cannot solve the task in the class, they can try to solve it at home, as the software is freely available, and solutions can be discussed in the next class.

2.2. Next steps and additional tasks

Solving at least the Four walls version can ensure that the student understood some basic concepts of code execution and code creation. If those goals are fulfilled, the course can advance to additional topics.

AlgoTaurus can also be used to demonstrate some additional key features of computer codes in an easy and entertaining way. Here we list a few tasks and issues that could be solved and discussed, depending on the age of the students and the aim of the course.

- A few students can show their working code, and the class can discuss them. "How does the different codes work? Do they work the same way? What modification can be considered as a different implementation of the same algorithm?" With these discussions it can be also demonstrated why code reading is difficult, and why comments, and other features of computer languages are useful to ease code reading. In many cases, code reading is hard, and students often have difficulty to describe the code in high-level version (e.g. even university students find it hard to explicitly state that in some solutions the robot works as if she

would follow the wall on one of her sides). The task also shows that the same task can be solved in various ways, either with the same basic algorithm or with some differing algorithms.
- Teachers can show non-working versions, and the students could fix them. The problems could be simple syntactic issues, or preferably incorrect algorithms or implementations.
- "How can you ensure that the code will work in all possible labyrinths that are generated?" After the students check a few possibilities, many times they come up with the idea that the correctness depends on the labyrinth type. "What are the conditions for the code running correctly?" This can demonstrate that codes are often prepared only for a set of specific features of the environment, or in other words, the code relies on the features of the environment.
- "Can you create a code that could solve both the Four walls and the Depth first versions?" This task is related to the previous one, and the same topics can be discussed in a different context.
- "Try to create a code that includes fewer lines. Sometimes to create a shorter code, it is not enough to modify the previous one, but an entirely different approach should be used." Usually, the task is more exciting and challenging if the teacher does not tell what the shortest possible solution is or how many lines it includes.
- "Try to create a code where the physical movement of the robot is minimized." "Try to create a code where the number of lines executed are minimized." Together with the previous task, different aspects of optimization can be discussed and demonstrated.

Another option is to continue computer programming learning with other educational languages. One of the present authors routinely use is Scratch to introduce other elements of computer programming (Maloney et al., 2004; *Scratch*, n.d.), e.g. loops, variables, arrays, and so on, before using a computer language that is used also by professionals. (Scratch can be used successfully even with adults, although the colorful interface might suggest that it is a children-only educational environment.) One might note that maybe Scratch or other similar environments could be used even without AlgoTaurus. However, there are a few cases where AlgoTaurus could serve a more efficient start than Scratch, and educators may consider which tool to use or how to combine them. First, because AlgoTaurus uses a more simple language than Scratch for a more specific complex problem, and because code execution and program states can be tracked more easily, a lot of computational issues could be taught more efficiently with AlgoTaurus after a very short instruction. Second, using several, differently working languages may help students to have a deeper understanding of the computational concepts, by seeing both the similarities and the differences between the environments, so they can generalize and abstract the relevant concepts. Similarly, there is a debate whether educational tools should be graphical or text-based or combined (Weintrop & Wilensky, 2018). Using a text-based environment, such as AlgoTaurus, and a graphical tool, like Scratch, may be another source of deeper understanding of computational concepts. (Importantly, because AlgoTaurus has a very limited language, one of the main issue of the complex syntax of text-based environments is less relevant here.) Third, optimization is an important aspect of computer programming, and AlgoTaurus can demonstrate this issue inherently after a short instruction and with important hands-on experience, because AlgoTaurus includes a simple to understand task with a specific aim. (See some additional recommendations in the next Section discussing the experiences with pilot teaching sessions.)

3. Experiences with AlgoTaurus based on pilot teaching sessions

In this section we summarize our main experiences in several teaching sessions. Many of our experiences are also included in section describing the features of AlgoTaurus and in the proposed class plan, because those suggestions were based on the problems and successful solutions we have met with.

Several types of classes were tested by the present authors and by other teachers both in the capital and in some smaller towns in Hungary. In primary schools several classes were tested: a

group of 10–12 years old children in Budapest (10 boys, 1 girl), a group of 8 years old children in a smaller town (8 boys, 4 girls), a group of 10 years old children in a smaller town (15 students), and a group of 11 years old children in a smaller town (9 boys and 5 girls). All of these students participated in groups of extracurricular activity. In higher education, AlgoTaurus was tested with cognitive psychology master students (several groups, usually with 12–18 persons; some of these groups used the Labirint software – see section What is AlgoTaurus?), primary school teacher students and preschool teacher students (two groups with 9 and 10 students). In all of these groups a lesson lasted for 45 or 90 min.

All of our reported findings are based on the observation of the students in the classroom (e.g. interaction of the students, the questions they ask about the task, their emotional reactions), their performance (whether they could solve the task or some versions of the task), and informal interviews with the students about the task.

One important finding is that larger part of the university students and children as young as 10 years of age could solve the task successfully, while the 8-years old children could only solve typically the four walls problem. The first solutions usually could come even after 20–30 min after starting to work on the problem, but most student need more time. It is not unusual that even after 90 min university students (supposedly with an above average intelligence) cannot solve the problem, and for children even more time can be expected to find a solution. Usually students who had former computer programming experience, find the solutions first.

It is important to highlight it again, that the main aim of the program is that students should understand code execution and code writing, and this can be achieved even if they cannot solve the Depth first version; one sign of conceptual understanding of basic programming principles is if they can solve the Four walls problem. Most student could solve the Four walls task, with some obvious exception where motivational or intellectual problems precluded a successful solution.

The task seemed to be enjoyable both for primary school children and for young adult university students. Basically, most students found the task interesting, they worked persistently even longer than 60 min without willing to have a break. In some of the groups, AlgoTaurus was one of the most interesting tasks among other class activities in general, with the fewest interruptions while solving the task, with the longest sustained attention and with a work with the highest concentration.

While usually we found that children can use AlgoTaurus successfully, among the test classes we found three cases when the task was not solved successfully or when the students were not motivated. First, as we mentioned, for 8-years old children the tasks proved to be too difficult, and typically only the Four walls version could be solved. Second, among the university students, kindergarten teacher students found the task too difficult and even irritating. However, it is not clear why they failed to solve the task. This might be the result of some special attitude, or one even can consider some intellectual selection among those students. Third, in a set of primary school classes children between 8 and 12 years of age found the task too difficult and frustrating, although in that case we found some issues with the teacher, e.g. the teacher reported technical problems installing AlgoTaurus and some conceptual issues about the teaching methods could not be resolved before the classes (while in some other schools, children with similar age could successfully use the software), so in that case it is more probably the attitude or teaching method of the teacher that might have influenced the performance. These failures may initially frame the limitations of AlgoTaurus use.

We have not tested the task with a larger sample of students living with learning difficulties or with lower intelligence, therefore, we do not know what would be the cognitive precondition to use AlgoTaurus successfully.

We have also tested some of the advanced tasks (correctness of the code, optimization problems, etc.) with university students. Large part of the students understood why code correctness or optimization is critical, and how it changes the approaches with which codes are created and how the codes can be tested, or how the environment is critical in those issues. Seemingly, even some quickly utilizable and simple examples in the framework of AlgoTaurus can introduce advanced

concepts about computer codes. These examples were also useful later in the course when these topics were discussed in the context of some general computer language and scripts.

Finally, we were able to compare the use of the former Labirint and our AlgoTaurus solution in some cognitive psychology master students groups: Earlier groups used Labirint and recent groups used AlgoTaurus (see some differences of the two software in section What is AlgoTaurus?). We found that students could use of AlgoTaurus more straightforwardly. Usually, using AlgoTaurus they hardly had any questions about the use of the software and about how a code is executed. Instead almost all of their questions were about the specific algorithm that could solve the task. Overall, the changes in AlgoTaurus compared to Labirint helped the users to create, run and test the code more seamlessly.

Overall, we found that if the attitude of the teacher was appropriate, most children older than 10-years old and young adults find the task inspiring, they are able to learn the bases of code execution and code writing, and AlgoTaurus can be used efficiently as an introductory tool in computer programming classes. These results are in line with the efficiency of some more complex mini-languages applied in universities (Brusilovsky et al., 1997).

3.1. Special case study: AlgoTaurus in cognitive psychology major classes

Computer programming could be essential for cognitive psychologists for several reasons: some experiments are run with experiment control software (e.g. Peirce, 2007), data could be analyzed either with scripts or more interactively with appropriate libraries or modules (Krajcsi, 2018; Perez & Granger, 2007), models can be simulated, and computer programming can form a new perspective how to create theories or models. For these reasons, computer programming is part of the cognitive psychology and cognitive science education at Eötvös Loránd University. Large part of the students do not have any computer programming experience before the course.

Different educational tools have been tested for this class, and AlgoTaurus was found to be the most efficient tool for an introduction. Different features contributed to this choice. First, all advantages discussed above apply here: the task can be introduced very shortly, students understand the task immediately, and they are very motivated to solve the puzzle and to experiment with their proposed solutions, while they learn implicitly (i.e. without any explicit explanation) the basic concepts of computer programming. Second, there are some features of the tool that can demonstrate some additional concepts that are essential for cognitive psychologists. Cognitive psychology supposes that the brain is a computational tool, and animal and human brains run some sort of codes (for an introductory explanation see, for example, Eysenck & Keane, 2005). In that framework, cognitive psychology can be considered as a reverse engineering task: scientist are trying to recover the code the humans or animals run (Pinker, 1999). In this framework, AlgoTaurus can help to demonstrate several features of computer codes, that are also essential for cognitive psychologists, such as: the same algorithm can be implemented in several ways; even if the source code or algorithm is available, it might be complicated to find out what the code is doing, because it is not trivial to see some consequences of code running; code relies on expected environment; relatedly, code correctness depends on the environment; optimization requires code modifications; depending on the specific optimization aim, different type of code modification is required, etc. Importantly, all these concepts and ideas can be demonstrated in a few classes, and students already have some firsthand experience about them, so these concepts are not some intangible abstract information that are hard to anchor.

In that Introduction to computer programming class at cognitive psychology major, AlgoTaurus is used together with another educational programming language, Scratch. Scratch seems to be an appropriate next step for several reasons. For example, when the language is extended, chance of syntax error gets higher, and a visual programming language, such as Scratch is more appropriate at that point, if the class wants to focus on the more general concepts (otherwise, finding syntax problems would take considerable time, which is not an issue in AlgoTaurus, because AlgoTaurus has a

very small language). Also, showing different languages or environments helps students to understand the common type of components (e.g. control flow, conditionals, etc.), which can ease learning any additional new languages. Additionally, because Scratch is more general than AlgoTaurus, and its building blocks are not only usable in a specific microworld, more general tasks can be applied. (But see some additional considerations, why AlgoTaurus is also used first, and why the combination of the two types of tools is more efficient than using only one of them in section Next steps and additional tasks.)

As a final step in that course, Python is taught, because researchers prefer high-level languages that could be used interactively, and in cognitive psychology after the year of popularity of Matlab, Python started to emerge as one of the most popular choice (note that R is also used if the main aim is data analysis). Showing different languages help students to understand some common principles behind languages, and also demonstrates syntactical and conceptual differences between languages. For this reason, Python is not only contrasted with the appropriate language elements of AlgoTaurus and Scratch, but analogue Matlab solutions are also shown.[2]

Overall, AlgoTaurus is a useful component of computer programming education in cognitive psychology education. This case study also illustrates a more general idea: although there are many educational languages and tools to help initial teaching of computer programming, it is still worth to look for new combinations of features, and create new educational tools, because depending on the educational aims and needs, different types of tools could be used as the most efficient solutions.

4. Conclusion

AlgoTaurus is a mini-language microworld programming environment, a remake and extension of the Labirint program. It is a free and open source software, meaning that no additional cost is required to use it, and the software can be extended with some coding knowledge. It can be used to start computer programming language use, and after a very short introduction students can start exploring solutions immediately. We found it to be an efficient tool to introduce the basic elements of computer programming, and it is a useful tool to complete other types of educational solutions, such as Scratch.

It might be useful to recall again that applied educational research is unavoidably limited these days, mostly because basic research has limited models about human learning, or in our case, specifically, about computational thinking. Currently, the most frequently utilized approach is (a) to build upon the intuition and experience of educators, and (b) to extend those observations with more systematic and objective measurements about the efficiency of educational tools and methods. In a similar manner, AlgoTaurus was built upon former experiences and feature selection identified by review studies, however, at the moment it is impossible to find theoretical justification why those solutions may work better than other solutions. While beyond the present pilot study, larger-sample, systematic, objective measurements can investigate the efficiency of AlgoTaurus and the related methods, it is a long way ahead to reach an extensive theoretical model that may account for those results.

Notes

1. Cognitive psychology students learn computer programming to understand computational models more appropriately and to create their own computational models, to analyze data, and to generate stimuli and measure responses in experiments. Many of those students have never learned computer programming before, and one major challenge was to teach the very basics of computer language use.
2. While this has not been a systematic comparison between various classes, some reader might be interested in differences between groups utilizing different introductory tools. We found that using mini-language microworld programming environment before using a production language helps learning programming compared to the production-language-only case (in those classes, Labirint was used before teaching the Presentation language, a C-like language for experiment control). Also, as we described in details, AlgoTaurus use is more informative

for the students than the Labirint use. Finally, adding Scratch to the AlgoTaurus/Labirint and Python combination is an efficient extension, because it creates a more general and abstract knowledge about computer programming concepts, and because it can help the students and teachers to focus on the Python-specific solutions and syntax forms instead of the general algorithms.

Acknowledgement

We thank Ádám Markója for his help to technically improve AlgoTaurus, including making the graphical user interface, building the Windows version and installer, and proposing further improvements and bugfixes. Attila Krajcsi is grateful to Anikó Kónya who made it possible to use Labirint with psychology students many years ago. We thank Éva Kovács and Erika Lakosné Makár for helping in some of the pilot teaching classes.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

This work was supported by Content Pedagogy Research Program of the Hungarian Academy of Sciences: Extended Didactical Grant [grant number ID 471028.].

Notes on contributors

Attila Krajcsi is a cognitive psychologist, leading the NumberWorks lab at Cognitive Psychology Department, ELTE Eötvös Loránd University, Hungary (www.thenumberworks.org). His main research area is numerical cognition, including the topics of basic representation supporting elementary number processing, development of symbolic numerical knowledge in preschoolers and methodological issues in measuring numerical abilities. He is also the developer of an automatic statistical data analysis software, CogStat (www.cogstat.org).

Csaba Csapodi is a teacher of mathematics. He has been working for 13 years in a secondary school in Budapest. Since 2015 he is a teacher at the Mathematics Institut at ELTE Eötvös Loránd University, Hungary. His main research area is the final exams in mathematics.

Eleonóra Stettner has a Master's degree in Mathematics, Physics and Informatics. She got her PhD degree in Geometry at the Budapest University of Technology and Economics. She worked at a secondary grammar school for 25 years, where she taught Mathematics, Physics and Informatics for children between 10 and 18 years. She has been working at Kaposvár University since 2003, teaching Mathematics in BSc, MSc and PhD courses. She is the Head of Department of Mathematics and Informatics. She has publications on the methodology of teaching Mathematics, the application of computer programmes in teaching Mathematics and the connection of Mathematics and Arts. She works as a research coordinator in the International Experience Workshop.

References

Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., & Miller, P. (1997). Mini-languages: A way to learn programming principles. *Education and Information Technologies*, *2*(1), 65–83. doi:10.1023/A:1018636507883

Chen, G., Shen, J., Barth-Cohen, L., Jiang, S., Huang, X., & Eltoukhy, M. (2017). Assessing elementary students' computational thinking in everyday reasoning and robotics programming. *Computers & Education*, *109*, 162–175. doi:10.1016/j.compedu.2017.03.001

Chen, C.-F., Yan, L.-Y., Yang, M.-C., & Lin, J. M.-C. (2005). *Teaching computer programming in elementary schools: A pilot study*. Retrieved from http://rportal.lib.ntnu.edu.tw/handle/77345300/34717

Eysenck, M. W., & Keane, M. T. (2005). *Cognitive psychology: A student's handbook* (5th ed.). New York: Psychology Press.

Fagin, B., & Merkle, L. (2003). *Measuring the effectiveness of robots in teaching computer science*. Proceedings of the 34th SIGCSE technical symposium on computer science education (pp. 307–311). New York, NY: ACM. doi:10.1145/611892.611994

Futschek, G., & Moschitz, J. (2011). Learning algorithmic thinking with tangible objects eases transition to computer programming. In *Informatics in schools. Contributing to 21st century education* (pp. 155–164). Berlin, Heidelberg: Springer. doi:10.1007/978-3-642-24722-4_14

Good, J., & Howland, K. (2017). Programming language, natural language? Supporting the diverse computational activities of novice programmers. *Journal of Visual Languages & Computing*, *39*, 78–92. doi:10.1016/j.jvlc.2016.10.008

Gopnik, A., Meltzoff, A. N., & Kuhl, P. K. (1999). *The scientist in the crib: Minds, brains, and how children learn* (Vol. xv). New York, NY: William Morrow.

Griffin, S. (2004). Building number sense with number worlds: A mathematics program for young children. *Early Childhood Research Quarterly*, *19*(1), 173–180. doi:10.1016/j.ecresq.2004.01.012

Howland, K., & Good, J. (2015). Learning to communicate computationally with flip: A bi-modal programming language for game creation. *Computers & Education*, *80*, 224–240. doi:10.1016/j.compedu.2014.08.014

Kayama, M., Satoh, M., Kobayashi, K., Kunimune, H., Hashimoto, M., & Otani, M. (2014). Algorithmic thinking learning support system based on student-problem score table analysis. *International Journal of Computer and Communication Engineering*, *3*(2), 134–140.

Krajcsi, A. (2018). *CogStat – An automatic analysis statistical software (version 1.7.0)*. Retrieved from https://www.cogstat.org

Lukyanov, K., & Volkov, S. (1993). *Labirint - Computer program (version 2.0)*.

Maloney, J., Burd, L., Kafai, Y., Rusk, N., Silverman, B., & Resnick, M. (2004). *Scratch: A sneak preview*. Presented at the second international conference on creating, connecting, and collaborating through computing (pp. 104–109), Kyoto.

Marcelino, M. J., Pessoa, T., Vieira, C., Salvador, T., & Mendes, A. J. (2018). Learning computational thinking and scratch at distance. *Computers in Human Behavior*, *80*, 470–477. doi:10.1016/j.chb.2017.09.025

Papastergiou, M. (2009). Digital game-based learning in high school computer science education: Impact on educational effectiveness and student motivation. *Computers & Education*, *52*(1), 1–12. doi:10.1016/j.compedu.2008.06.004

Papert, S. A. (1993). *Mindstorms: Children, computers, and powerful ideas* (2nd ed.). New York, NY: Basic Books.

Peirce, J. W. (2007). Psychopy – psychophysics software in python. *Journal of Neuroscience Methods*, *162*(1–2), 8–13. doi:10.1016/j.jneumeth.2006.11.017

Perez, F., & Granger, B. E. (2007). IPython: A system for interactive scientific computing. *Computing in Science & Engineering*, *9*(3), 21–29. doi:10.1109/MCSE.2007.53

Pinker, S. (1999). *How the mind works* (Later prt. ed.). New York, NY: W. W. Norton & Company.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2), 137–172. doi:10.1076/csed.13.2.137.14200

Román-González, M., Pérez-González, J.-C., & Jiménez-Fernández, C. (2017). Which cognitive abilities underlie computational thinking? Criterion validity of the computational thinking test. *Computers in Human Behavior*, *72*, 678–691. doi:10.1016/j.chb.2016.08.047

Román-González, M., Pérez-González, J.-C., Moreno-León, J., & Robles, G. (2018). Can computational talent be detected? Predictive validity of the computational thinking test. *International Journal of Child-Computer Interaction*. doi:10.1016/j.ijcci.2018.06.004

Scratch. (n.d.). Retrieved from https://scratch.mit.edu/

Tekerek, M., & Altan, T. (2014). The effect of scratch environment on student's achievement in teaching algorithm. *World Journal on Educational Technology*, *6*(2), 132–138.

Voogt, J., Fisser, P., Good, J., Mishra, P., & Yadav, A. (2015). Computational thinking in compulsory education: Towards an agenda for research and practice. *Education and Information Technologies*, *20*(4), 715–728. doi:10.1007/s10639-015-9412-6

Weintrop, D., & Wilensky, U. (2018). How block-based, text-based, and hybrid block/text modalities shape novice programming practices. *International Journal of Child-Computer Interaction*, *17*, 83–92. doi:10.1016/j.ijcci.2018.04.005

Xinogalos, S., Satratzemi, M., & Malliarakis, C. (2017). Microworlds, games, animations, mobile apps, puzzle editors and more: What is important for an introductory programming environment? *Education and Information Technologies*, *22*(1), 145–176. doi:10.1007/s10639-015-9433-1