# Comparing Cognitive Load Among Undergraduate Students Programming in Python and the Visual Language Algot

Sverrir Thorgeirsson*
sverrir.thorgeirsson@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Theo B. Weidmann*
tweidmann@ethz.ch
ETH Zurich
Zurich, Switzerland

Karl-Heinz Weidmann
karl-heinz.weidmann@fhv.at
University of Applied Sciences Vorarlberg
Dornbirn, Austria

Zhendong Su
zhendong.su@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

## Abstract

**This paper examines whether undergraduate students perform better and experience lower cognitive load when programming in Algot, a visual programming language that supports programming by demonstration, than in the textual programming language Python. We recruited 38 first-semester computer science university students who had received prior instruction in the programming language Python but were unfamiliar with Algot. Participants reviewed a 12-minute video tutorial about Algot and performed the same programming tasks in Python and Algot. We graded student submissions, estimated cognitive load through physiological measures and a validated post-test survey, and evaluated free-form feedback. Our results indicated that students experienced lower negative (extraneous and intrinsic) and higher positive (germane) cognitive load when programming in Algot. Additionally, students programming in Algot scored an average grade of 5.8 out of 10, compared to an average grade of 3.4 when using Python for the same tasks, and according to the free-form feedback, Algot is perceived as well-designed and easy to learn.**

***CCS Concepts:*** **• Human-centered computing → Empirical studies in HCI**; **• Social and professional topics → Computer science education**; **• Software and its engineering → General programming languages**.

***Keywords:*** visual programming, programming by demonstration, undergraduate education, algorithms and data structures, cognitive load

## 1 Introduction

The perception of computer programming as a challenging activity has led researchers and practitioners to propose visual languages to simplify both the learning and instruction of programming. In particular, block-based languages like Scratch [21] are favored by some educators as they address the challenge of syntax errors and offer a clear, visual overview of what the language can achieve [33]. However, these languages do not necessarily address the need for a direct, tangible connection between how a program is composed and its effects when it is run. For beginners and experienced programmers alike, it has been suggested that this disconnect can pose a significant challenge [30].

Algot [29, 32] is a visual programming language that takes a different approach than block-based languages. Rather than arranging and filling in templates of imperative code, the Algot environment supports a novel programming by demonstration paradigm where the user can create programs by directly manipulating a close approximation of the program state and by posing related, natural-language queries. This way, Algot aims to simplify the learning process by making program meanings clearer; when defining a program in Algot, it mirrors the execution process, reflecting ideas from Bret Victor's 2012 paper on learnable programming [31].

In an earlier paper, we speculated that the Algot programming paradigm may reduce students' cognitive load in comparison to textual programming [32]. While another study indicates that Algot may have some effectiveness in education [28], until now, the hypothesis has Algot lowers students' cognitive load not been subjected to empirical evaluation at the tertiary level. In this paper, we present the first study on Algot among first-semester university students pursuing a degree in computer science. After administering a pre-study self-evaluation survey of the participants' programming skills, we conducted a controlled experiment in which the participants completed the same task in Algot and the textual programming language Python, a common choice [20, 22] for first-semester programming courses. While the participants completed the tasks, we measured their cognitive load via their physiological responses (eye fixations and galvanic skin response) followed by a validated post-test

*Sverrir Thorgeirsson and Theo B. Weidmann are co-primary authors.

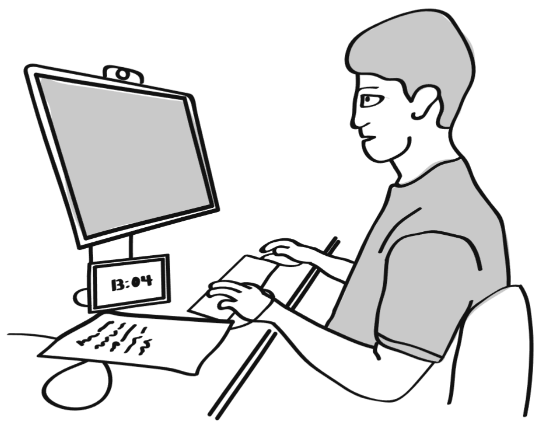


**Figure 1.** A sketch of the testing environment used in the study, showing the eye-tracking equipment (top of monitor), test instructions, the electrode cable extending from the participant's left hand, and a digital clock for participants to keep track of time.

survey on cognitive load intended for computer science [14]. We also graded the quality of their solutions and collected their free-form feedback on Algot and the study design. We sought to answer the following research questions:

**RQ1** What is the difference in cognitive load between Algot and Python when solving introductory programming tasks?
- **H0** Students experience no significant difference in cognitive load when programming in Algot.
- **H1** Students experience lower cognitive load when programming in Algot.
- **H2** Students experience lower cognitive load when programming in Python.

**RQ2** What are the perceived advantages and challenges of using Algot for programming, as reported by students, compared to a traditional programming language like Python?
- **H0** Students report mixed results or no significant differences between Algot and Python.
- **H1** Students mostly report advantages of using Algot, such as ease of use or reduced likelihood of errors.
- **H2** Students mostly report disadvantages of using Algot, such as limitations in language features or the need for additional learning resources.

We used a mixed methods design incorporating quantitative methods to answer RQ1 and qualitative data analysis to analyze the free-form feedback to answer RQ2.

## 2 Background

### 2.1 Cognitive Load Theory

Cognitive load theory is a theoretical framework that attempts to explain how the cognitive system processes information and how it affects learning. Originating primarily through the work of John Sweller in the 1980s [17, 25], the framework expanded and evolved significantly in the 1990s [18] and has since become a major theory in the fields of instructional design [7] and multimedia learning [13]. Cognitive load theory posits that working memory has limited capacity, and that the amount of cognitive resources available for processing new information is limited [26]. When cognitive resources are exceeded, learning becomes less efficient and may even be impeded. In other words, learners can be overwhelmed by the demands of a task, making it difficult for them to process and retain new information.

Sweller [27] characterizes three types of cognitive load: intrinsic, extraneous, and germane. Intrinsic cognitive load refers to the inherent difficulty of the material being learned and is determined by the complexity of the topic and the learner's prior knowledge of the subject. Extraneous cognitive load refers to the mental effort required to process information that is not essential for learning. It is caused by irrelevant information or poorly designed instructional materials. Germane cognitive load refers to the mental effort required to integrate new information into existing knowledge and to construct meaningful representations of the learned material. Consequently, effective instruction should aim to manage extraneous load, optimize intrinsic load, and maximize germane load to facilitate learning.

In response to conflicting empirical studies on cognitive load, Sweller suggested in 2019 that germane cognitive load does not contribute to the total cognitive load, but instead redistributes working memory resources from extraneous activities [26]. This addresses some perceived conceptual flaws with the original model such as its "tautological character" or unfalsifiability [3]. A 2022 review of cognitive load theory in CS education [4] calls the former model "old CLT".

Some instruments that have been found to correlate with cognitive load are electroencephalography (EEG) [2], functional near-infrared spectroscopy (fNIRS) [6] and heart rate variability (HRV) [24]. Eye-tracking is also commonly used, for example in visual computing [34]; some common measures are pupil diameter and microsaccades [9]. Eye fixation is also commonly used, although to a lesser degree in studies on computer programming [8]. Additionally, galvanic skin response has been found to be a good indicator of cognitive load with high granularity [23]. Surveys on cognitive load such as the Paas scale [19] are also commonly used, either on their own or in combination with physiological measures. An 11-item Likert-scale survey on self-reported cognitive load [14], adopted from an earlier validated instrument [11] but intended specifically for CS education research, has been

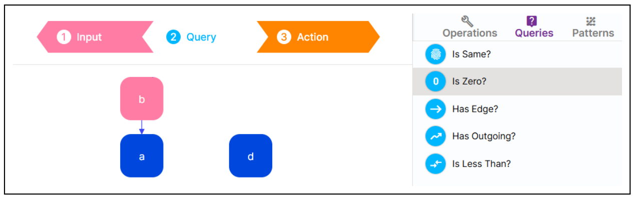


**Figure 2.** A screenshot of operation demonstration view in *Algot*. The user has set two input variables *a* and *d* and used pattern matching to find a parent node of *a*.

commonly used [5, 15]. As with the original version, this instrument is intended to measure the three aforementioned types of cognitive load: intrinsic, extraneous, and germane.

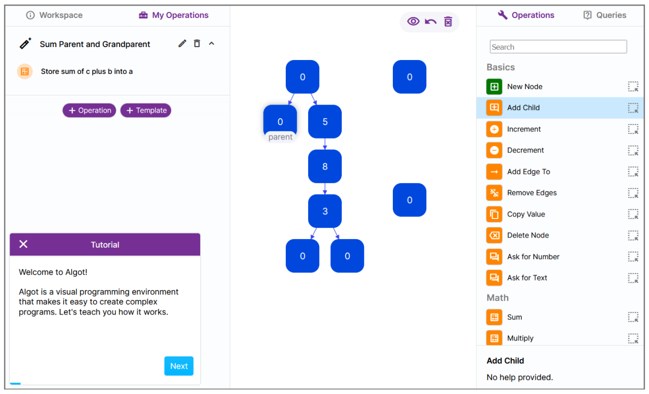
### 2.2 Algot

Algot is a visual programming language originally designed for education [29, 32]. The central research insight driving the development of Algot is its ability to overcome the syntax-semantics barrier of programming through direct manipulation of the program state. By enabling learners to interact with the program state visually, Algot may allow them to focus on understanding core algorithmic concepts rather than wrestling with syntax. This approach distinguishes Algot from other programming languages and is intended to foster a deeper learning experience.

In Algot, the program state is a graph. The use of a graph to represent state is a natural choice, as the state in most other languages form graphs too: In object-oriented programming, as propagated by Python, for example, objects form the object graph, where objects are regarded as nodes, while the references between them form edges. In Algot, this *state graph* is always visible in the *state view* (see center area in Fig. 2). The state view allows the programmer to apply *operations* and *queries*:

- Operations are applied to nodes and modify the state graph, for example by deleting nodes, adding edges or changing the value of a node. An operation applied in the state view takes effect immediately. Certain base operations are available in the system by default. Additionally, the user can import libraries of operations created by other users.
- Queries are natural-language binary questions shown about a selection of nodes such as, “Is the value of this node zero?” or “Is the value of node *A* less than the value of node *B*?”.

Programs are implemented in Algot by defining new operations. An operation can be implemented via demonstration at any time in the left sidebar of the application (see Fig. 3). When doing so, the programmer enters the demonstration view (see Fig. 2). Operations in Algot are defined in three stages:



Welcome to Algot!

Algot is a visual programming environment that makes it easy to create complex programs. Let's teach you how it works.

No help provided.

**Figure 3.** A screenshot showing the latest version of the main Algot user interface, showing the state view (center) and some of the available base operations (right).

1. Input stage: The input nodes of the operation are specified. In this stage, the programmer can use builtin pattern matching to find any nodes in the connected component of any input node.
2. Query stage: The programmer can run queries on the modeled input nodes. Fig. 2 shows some of the available queries that the programmer can ask about the input nodes, such as whether a has a lower value than b, or whether the node d has any outgoing edges.
3. Action stage: The programmer manipulates the nodes from the input stage using existing operations (base operations, imported operations or user-created operations). Each action taken can be predicated on the query results from the query stage.

For instance, to define an operation in Algot that computes a pre-defined number of Fibonacci numbers and outputs them as a linked list, the user might:

(a) define an operation `Fib` which takes two inputs, k and `idx`, and use pattern matching to find the parent of k, that is m (see Fig. 2);
(b) then apply the query `Is Zero?` on `idx`, and on a negative result, append a child node c to k and call the base operation Sum on m, k and c;
(c) and finally, decrement `idx`, call the query `Is Zero?` on `idx` again, and on a negative result, call `Fib` on c and `idx`.

This operation could then be used be used in a wrapper function which computes `Fib` on the two first Fibonacci numbers 0 and 1, or could be used to generate a list with other starting values.

We refer the reader to our earlier paper [32] for a more detailed description of the Algot semantics.

## 3 Method

After receiving ethics approval, the participants were recruited from a first-semester computer science (CS) course for CS majors at the University of Applied Science Sciences, Vorarlberg, Austria, where the study was conducted.. All 40 students enrolled in the class were invited to participate. Students received no financial compensation for their participation, but received extra course credit.

Students who agreed to participate were asked to indicate their familiarity with a few textual programming languages in order to help us determine which language the control group should use. After reviewing the results, we chose Python because it scored highly and its syntax is believed to be suitable for students [20]. All participants had been taught basic Python concepts in lectures from their degree program, for example lists, loops, conditionals, functions and parameters. To calibrate the difficulty of the Python exercises used and the ideal duration of the experiment, we conducted a pilot experiment with a professional software developer as a volunteer two months before the study was conducted.

The study took place in a usability laboratory with one participant tested at a time. Participants were first given 15 minutes to review a 12-minute video tutorial describing (i) how to use a simplified version of *Code Expert*, an online development environment developed and used at ETH Zurich which is suitable for programming in Python, and (ii) the basics of how to program in Algot. All participants were given the same tutorial and they were free to pause or rewind it. Both Code Expert and Algot are web-based tools specifically designed for use in education environments, and thus allow for a fair comparison.

We did not expect the participants in the study to be familiar with either Algot or Code Expert. Participants only knew which language they would be using once they arrived at the test location. To limit acquiescence bias in the post-test survey, students were not told which language or environment was the focus of the study.

To achieve more robust results, we adopted a within-subjects test design where each participant was asked to complete two tasks, once in Python and once in Algot. Participants were given 15 minutes to solve each task. We used counterbalancing to control for order effects. Both tasks were on manipulating a graph, a topic that the participants had learned in an earlier course on algorithms and data structures.

The tasks were:

1. Define an operation that takes as an input a single node *a*, appends two child nodes to *a* and assigns them the same value as *a*.
2. Define an operation that takes as an input two nodes *a* and *b* such that *a* has a child *c*. If *a* does not have the value 0, change *c*'s parent to *b*.

The chosen tasks are simple but incorporate important concepts in introductory CS, such as functions and function calls, parameters and arguments, variable assignment, lists,

```
def add_child_nodes(
            node):
  b = add_child(node)
  c = add_child(node)
  copy_value(node, b)
  copy_value(node, c)
```

```
def move_child_nodes(a,
            b):
  c = get_children(a)[0
             ]
  if not is_zero(a):
    remove_edges(c)
    add_edge_to(b, c)
```

**Figure 4.** Reference solutions for the tasks in Python. All structures and concepts used in this solution were studied by students in their courses.

and conditional statements. On a high level, the tasks require some skills with decomposition, pattern recognition and abstraction.

Participants were provided with a template for both languages and six functions for accessing and manipulating the graph: *is_zero*, *get_children*, *remove_edges*, *add_child*, *copy_value* and *add_edge_to*. Documentation for these functions was provided at the top of the template. The reference solution for Python is seen in Fig. 4 and required students to write four lines of code for each task. The available functions and documentation closely resembles the base operations available in Algot. The expected solutions in both languages thus proceed almost identically.

We evaluated the quality of the participants' solutions and their responses on the 11-item survey on cognitive load [14]. We asked students to complete the survey for both the Algot and Python task settings. Lastly, we invited participants to provide free-form feedback on the study or either of the code environments. We evaluated the responses using qualitative coding to identify key themes and to possibly gain further insight into our results.

To measure the participants' physiological responses while they completed the assignments, we measured their galvanic skin response and eye movements. These measures were selected to provide objective, physiological indicators of cognitive load to complement the other data collected. For eye movements, we chose to focus on fixation rather than microsaccades or pupil dilation. Accurate microsaccade measurements would have required head-stabilized eye tracking with a chin rest, but we opted for head-free eyetracking (see Fig. 1) since this is a more natural and realistic setting. Furthermore, comparing pupil dilations across the two tasks would have required a controlled, consistent luminance level between the two tasks.

To measure fixations, we used a Tobii eye tracker with a sampling frequency of 60 Hz. The eye tracker uses the Tobii Fixation Filter algorithm to process and analyze the fixation data, a robust algorithm that accurately identifies fixations by considering factors such as velocity threshold, minimum fixation duration, and noise reduction [16]. For our data analysis, we used the Tobii validity marker to ensure the reliability of the fixation data. The validity marker helps in identifying and filtering out data points with low validity, such as those affected by blinks, off-screen gaze, or other artifacts that could compromise data quality. For fixation, we only considered time points with the validity code 0 out of 4, i.e., where the data is considered highly trustworthy. We note that given the distinct visual presentations of the two programming environments, the eye-tracking comparisons should be interpreted with caution.

### 3.1 Evaluation of Task Solutions

To evaluate the accuracy of the solutions that participants produced in both Algot and Python, we manually reviewed screen recordings of all tests. For both tasks, we selected four grading criteria that closely correspond to the steps that must be performed to solve the task and are detailed in Table 2. Each criterion is either fulfilled or unfulfilled, represented by 1 or 0 points, respectively. When reviewing the screen recordings, we graded the best solution each participant produced at any point even if they later changed the solution. Points were also awarded in Python if undefined variables `a`, `b` or `c` were used and the intention was clear.

Note that achieving full points does not strictly imply that the solution is correct. For example, even if the edge from $a$ to $c$ in Task 2 was removed unconditionally, we awarded points.

## 4 Results

Out of the 40 students invited to participate in the study, 38 consented to do so. According to our demographic survey, 13 participants were female and 25 were male. The participants ranged in age from 18 to 36 with a median age of 21.

As reported below, we used Bayesian analysis to interpret some of the data. All the Bayes factors reported have estimation errors below 0.1%.

### 4.1 Self-Reported Cognitive Load

**Table 1.** The results from the Cognitive Load Component Survey showing the mean and standard deviation (SD)

| Cognitive Load | Algot Mean | SD | Python Mean | SD |
|---|---|---|---|---|
| Intrinsic Load | 2.56 | 2.18 | 3.55 | 2.21 |
| Extraneous Load | 1.45 | 1.51 | 2.50 | 2.28 |
| Germane Load | 5.37 | 2.19 | 3.46 | 2.41 |

Descriptive statistics from the Cognitive Load Component Survey can be seen in Table 1. Using the statistical software JASP [12], we computed the Bayesian Paired Samples Student's t-test to test our hypothesis, that students experienced (i) lower intrinsic and extraneous cognitive load, and (ii) higher germane cognitive load when programming the tasks in Algot than when programming in Python. This is what we refer to as the alternative hypothesis; the null hypothesis is the mutually exclusive claim. The tests returned $BF_{10} = 15.4$ for intrinsic load, $BF_{10} = 28.8$ for extraneous load, and $BF_{10} = 2249$ for germane load. $BF_{10}$ values above 10 indicate strong evidence in favor of the alternative hypothesis.

We also computed a classical Paired Samples t-test for our hypothesis on the intrinsic, extraneous and germane cognitive load, returning the p-values 0.002, 0.001, and $p < 0.001$, respectively, meaning that under the framework of frequentist statistics, we could reject the null hypothesis.

### 4.2 Textual Feedback

Of the 38 students who participated, 34 submitted textual feedback. We used software for qualitative data analysis, MAXQDA[10], to color code the responses. After reading through the responses to get a general sense of the data, we identified four feedback categories: *Study Design*, *User Interface (UI)*, *Algot*, *Python*:

1. 14 participants submitted feedback or opinions on the study design. Five participants suggested that the presentation of the provided Python functions could have been improved, such as with syntax highlighting and a more clear indication of their return value. Three participants expressed that the computer laboratory setting fell short compared to their usual programming environments due to an unfamiliar keyboard or lack of access to the internet or a debugger. Three participants would have liked a longer or a more detailed tutorial. Three participants left only positive comments, e.g., "Everything you have to know about the study was perfectly introduced and explained. As I tried the study, I always knew in which phase I was currently."
2. 13 participants had UI-related suggestions or comments on parts they found confusing. All of these comments were on Algot. The comments were diverse and without a strong common theme. Examples: Two participants found it too easy to accidentally delete nodes, one participant was unsure about the meaning of certain words used in the interface such as "append", and one student was unsure how to delete nodes.
3. 10 participants left general comments on Algot. All of these comments were positive remarks on the system or the language, such as "I liked the simplicity of Algot," "It was easier to solve the exercise in Algot, at least for someone with little knowledge," "Algot was much easier to use than Python, "The Algot program was very simple to understand", "I really like Algot. It reminds me of Scratch," "I liked Algot. I could see (younger) children use it, to make their first steps into programming" and "Algot: it tempts a person to try

it out because it is very nicely formatted & designed (color, structure, descriptions, etc)."

4. 12 students left comments that mentioned Python. Most of these comments had to do with the study design. None of the comments were positive. One student remarked "In Python, I didnâĂŹt understand what was going on, which nodes are already there, if my code actually did anything or what I had to change in my code."

In general, the free-form data suggests that students found it easier solving the tasks in Algot than Python, which aligns with the results from the cognitive load survey instrument. Some comments from participants related to the validity of the study and to which extent the results can be generalized are discussed further in Section 6.

### 4.3 Task Solutions

We graded each submission on four criteria on a scale from 0 to 1 such that each criterion had equal weight. The results can be seen in Table 2. The average grade on the Algot test component was significantly higher than on the Python component. For completeness, we computed a Bayesian Paired Value T-Test to test the hypothesis that students would perform better on the tasks with Algot than Python, which resulted in $BF_{10} = 6413$.

**Table 2.** Accuracy of the solution

| **Task 1** | **Algot** | **Python** |
|---|---|---|
| Solution adds first child $u$ to $a$ | 0.55 | 0.44 |
| Solution adds second child $v$ to $a$ | 0.55 | 0.44 |
| Solution copies value from $a$ to $u$ | 0.50 | 0.23 |
| Solution copies value from $a$ to $v$ | 0.50 | 0.26 |
| Fully Correct Solution | 50% | 18% |
| **Task 2** | **Algot** | **Python** |
| Solution accesses the child node $c$ | 0.73 | 0.10 |
| Solution checks whether $c$ is 0 and correctly employs it as a predicate | 0.65 | 0.31 |
| Solution adds edge from $b$ to $c$ | 0.52 | 0.36 |
| Solution removes edge from $a$ to $c$ | 0.60 | 0.52 |
| Fully Correct Solution | 47% | 5% |
| **Average Grade** | **Algot** | **Python** |
| Mean | **0.58** | **0.34** |
| Std. Deviation | 0.42 | 0.34 |

### 4.4 Physiological measurements

**4.4.1 Eye fixation.** After preprocessing the data and applying the Tobii Fixation Filter to identify all high-validity time points with gaze events classified as "fixation," we time-normalized the data to calculate the percentage of time each participant spent fixating during the task. Data from three participants had to be excluded since some of the eye-tracking data from the first day of testing was accidentally overwritten. On average, participants fixated longer while solving the Algot task (48.0%, $SD = 0.215$) compared to the Python task (21.5%, $SD = 0.188$). A Bayesian Paired Samples Student's T-Test yielded a $BF_{10}$ value of 0.065, indicating a low probability that our alternative hypothesis is true, i.e., that students would exhibit fewer fixation periods when programming in Algot than Python.

In addition to our pre-planned analysis, we took a granular look at the data and calculated the frequency of participants fixating for more than 0.5 seconds, which suggests deep mental processing [1]. Here, the results were nearly identical (22% for both tasks). When we further increased this threshold to 1 and 2 seconds, respectively, the percentage of time spent fixating decreases for the Algot task in comparison to the Python task (9.2% versus 10.4%) and (1.4% versus 3.1%).

**4.4.2 Galvanic skin response.** After preprocessing the skin conductance data, we calculated three measures for each of the two tasks, Algot and Python: the mean SCL (Skin Conductance Level), which represents the overall level of arousal or general stress; the mean SCR (Skin Conductance Response) amplitude, which indicates the average strength of the individual skin conductance responses; and the SCR frequency, which reflects the proportion of data points associated with skin conductance responses, providing information about the occurrence of these responses during the tasks. The descriptive statistics for these measures can be seen in Table 3.

**Table 3.** Bayesian Paired Samples Student's T-Test for three galvanic skin conductance metrics

| | Algot | Python | $BF_{10}$ % |
|---|---|---|---|
| Mean SCL | 34990 | 35080 | 0.28 |
| Mean SCR Amp. | 0.362 | 0.622 | 7.00 |
| Mean SCR Freq. | 0.133 | 0.158 | 8.57 |

On average, participants exhibited lower skin conductance during the Algot task compared to the Python task. Our Bayesian Paired Samples Student's T-Test provided evidence for differences in SCR amplitude and frequency between the two tasks (7.00 and 8.57), suggesting that participants experienced lower cognitive load during the Algot task. However, the analysis did not show a significant difference in the mean SCL between the tasks.

## 5 Discussion and Threats to Validity

In this study, we sought to answer two research questions. The first research question (RQ1) asked how Algot affects students' cognitive load compared to a textual programming language like Python when solving introductory programming tasks. Our findings suggest that students experience lower cognitive load when programming in Algot, as indicated by the self-report surveys, the participants' performance, and to some extent the physiological data. Specifically, the lower SCR amplitude and frequency values observed for the Algot task compared to the Python task support H1. However, the similar SCL values and the eye fixation data imply that the tasks may differ in certain aspects of cognitive load, but not in others, or that the lack of difference might be due to variability in the physiological data or other uncontrolled external factors.

The second research question (RQ2) aimed to explore the perceived advantages and challenges of using Algot for programming, as reported by students, compared to a traditional programming language like Python. The free-form feedback on the self-report surveys indicates that participants found the Algot task easier, more engaging, and less demanding than the Python task, suggesting that the hypothesis H1 ("students mostly report advantages of using Algot, such as ease of use or reduced likelihood of errors") is correct.

Our hypothesis is that the difference in results can be attributed to Algot having a short feedback loop and a visible program state, which are features considered helpful for beginners [31]. As a prime example, participants struggled in the rubric "Solution accesses the child node *c*" for Task 2 in Python, asking them to access a parent-child relation. This same subtask became easier in Algot due to a visual state. Additionally, some mistakes participants made in Python, such as passing arguments in the wrong order to *add_edge_to*, can be detected visually in Algot.

However, our results are limited by the scope of the study such as the task selection and educational level of the participants, and some challenges might surface as learners progress to more complex programming tasks. In future studies, it would be beneficial to delve deeper into long-term learning effects, potential pitfalls, and how Algot scales with more advanced programming concepts. Note that the tasks were not chosen with the aim of being easier to complete in one language than another, but rather that they could be completed by first-semester CS students in the limited time allocated under a thorough, within-subjects study design where one individual is tested at a time using physiological measurements.

Other threats to validity of the study are the ecological validity, as the controlled laboratory environment might not reflect the settings in which students typically program, which may have affected their performance and cognitive load; a few students noted explicitly that this affected their performance. Additionally, for the internal validity, we note that two minor experimental mishaps occurred during the study: The testing environment terminated too early for the first participant, and another participant encountered an unstable internet connection during the Python tasks, but still achieved a full score. We took this into account when analyzing the physiological data.

## 6 Conclusion

This study aimed to compare the cognitive load experienced by participants while completing tasks in two different programming languages, Algot and Python. Our findings suggest that, on average, participants experienced lower cognitive load in the Algot tasks as indicated by the self-report surveys, the participants' performance, and to some extent the captured physiological data. We presented a hypothesis to explain these effects.

## References

[1] Amine Abbad-Andaloussi, Thierry Sorg, and Barbara Weber. 2022. Estimating developers' cognitive load at a fine-grained level using eye-tracking measures. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 111–121.

[2] Pavlo Antonenko, Fred Paas, Roland Grabner, and Tamara Van Gog. 2010. Using electroencephalography to measure cognitive load. *Educational psychology review* 22 (2010), 425–438.

[3] Ton De Jong. 2010. Cognitive load theory, educational research, and instructional design: Some food for thought. *Instructional science* 38, 2 (2010), 105–134.

[4] Rodrigo Duran, Albina Zavgorodniaia, and Juha Sorva. 2022. Cognitive Load Theory in Computing Education Research: A Review. *ACM Transactions on Computing Education (TOCE)* 22, 4 (2022), 1–27.

[5] Barbara J Ericson, Lauren E Margulieux, and Jochen Rick. 2017. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling international conference on computing education research*. 20–29.

[6] Frank A Fishburn, Megan E Norr, Andrei V Medvedev, and Chandan J Vaidya. 2014. Sensitivity of fNIRS to cognitive state and load. *Frontiers in human neuroscience* 8 (2014), 76.

[7] Paul Ginns and Jimmie Leppink. 2019. Special issue on cognitive load theory. *Educational Psychology Review* 31 (2019), 255–259.

[8] Lucian José Gonçales, Kleinner Farias, and Bruno C da Silva. 2021. Measuring the cognitive load of software developers: An extended Systematic Mapping Study. *Information and Software Technology* 136 (2021), 106563.

[9] Krzysztof Krejtz, Andrew T Duchowski, Anna Niedzielska, Cezary Biele, and Izabela Krejtz. 2018. Eye tracking cognitive load using pupil diameter and microsaccades with fixed gaze. *PloS one* 13, 9 (2018), e0203629.

[10] Udo Kuckartz and Stefan Rädiker. 2019. *Analyzing qualitative data with MAXQDA*. Springer.

[11] Jimmie Leppink, Fred Paas, Cees PM Van der Vleuten, Tamara Van Gog, and Jeroen JG Van Merriënboer. 2013. Development of an instrument for measuring different types of cognitive load. *Behavior research methods* 45 (2013), 1058–1072.

[12] Jonathon Love, Ravi Selker, Maarten Marsman, Tahira Jamil, Damian Dropmann, Josine Verhagen, Alexander Ly, Quentin F Gronau, Martin Šmíra, Sacha Epskamp, et al. 2019. JASP: Graphical statistical software for common statistical designs. *Journal of Statistical Software* 88 (2019), 1–17.

[13] Richard Mayer and Richard E Mayer. 2014. *The Cambridge handbook of multimedia learning* (second ed.). Cambridge university press.

[14] Briana B Morrison, Brian Dorn, and Mark Guzdial. 2014. Measuring cognitive load in introductory CS: adaptation of an instrument. In *Proceedings of the tenth annual conference on International computing education research*. ACM, 131–138.

[15] Briana B Morrison, Lauren E Margulieux, and Mark Guzdial. 2015. Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the eleventh annual international conference on international computing education research*. 21–29.

[16] Anneli Olsen. 2012. The Tobii I-VT fixation filter. *Tobii Technology* 21 (2012), 4–19.

[17] Elizabeth Owen and John Sweller. 1985. What do students learn while solving mathematics problems? *Journal of educational psychology* 77, 3 (1985), 272.

[18] Fred Paas, Alexander Renkl, and John Sweller. 2003. Cognitive load theory and instructional design: Recent developments. *Educational psychologist* 38, 1 (2003), 1–4.

[19] Fred GWC Paas. 1992. Training strategies for attaining transfer of problem-solving skill in statistics: a cognitive-load approach. *Journal of educational psychology* 84, 4 (1992), 429.

[20] Atanas Radenski. 2006. "Python first": A lab-based digital introduction to computer science. *ACM SIGCSE Bulletin* 38, 3 (2006), 197–201.

[21] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

[22] Christine Shannon. 2003. Another breadth-first approach to CSI using Python. *ACM SIGCSE Bulletin* 35, 1 (2003), 248–251.

[23] Yu Shi, Natalie Ruiz, Ronnie Taib, Eric Choi, and Fang Chen. 2007. Galvanic skin response (GSR) as an index of cognitive load. In *CHI'07 extended abstracts on Human factors in computing systems*. 2651–2656.

[24] Soroosh Solhjoo, Mark C Haigney, Elexis McBee, Jeroen JG van Merrienboer, Lambert Schuwirth, Anthony R Artino, Alexis Battista, Temple A Ratcliffe, Howard D Lee, and Steven J Durning. 2019. Heart rate and heart rate variability correlate with clinical reasoning performance and self-reported measures of cognitive load. *Scientific reports* 9, 1 (2019), 1–9.

[25] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive science* 12, 2 (1988), 257–285.

[26] John Sweller, Jeroen JG van Merriënboer, and Fred Paas. 2019. Cognitive architecture and instructional design: 20 years later. *Educational Psychology Review* 31 (2019), 261–292.

[27] John Sweller, Jeroen JG Van Merrienboer, and Fred GWC Paas. 1998. Cognitive architecture and instructional design. *Educational psychology review* (1998), 251–296.

[28] Sverrir Thorgeirsson, Lennart Lais, Theo Weidmann, and Zhendong Su. 2024. Recursion in Secondary Computer Science Education: A Comparative Study of Visual Programming Approaches. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education (SIGCSE 2024)*. Portland, Oregon. In Press.

[29] Sverrir Thorgeirsson and Zhendong Su. 2021. Algot: an educational programming language with human-intuitive visual syntax. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–5.

[30] Bret Victor. 2012. Inventing on principle. In *Invited talk at the Canadian University Software Engineering Conference (CUSEC)*, Vol. 5.

[31] Bret Victor. 2012. Learnable programming: Designing a programming system for understanding programs. *URL: http://worrydream.com/LearnableProgramming* (2012).

[32] Theo B Weidmann, Sverrir Thorgeirsson, and Zhendong Su. 2022. Bridging the Syntax-Semantics Gap of Programming. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 80–94.

[33] David Weintrop. 2019. Block-based programming in computer science education. *Commun. ACM* 62, 8 (2019), 22–25.

[34] Johannes Zagermann, Ulrike Pfeil, and Harald Reiterer. 2016. Measuring cognitive load using eye tracking technology in visual computing. In *Proceedings of the sixth workshop on beyond time and errors on novel evaluation methods for visualization*. 78–85.