The European Proceedings of
Social and Behavioural Sciences
EpSBS
www.europeanproceedings.com
e-ISSN: 2357-1330

DOI: 10.15405/epsbs.2021.02.02.127

NININS 2020

International Scientific Forum «National Interest, National Identity and National Security»

WAYS TO WRITE ALGORITHMS AND THEIR EXECUTION TRACES FOR TEACHING PROGRAMMING

Oleg A. Sychev (a)*, Mikhail E. Denisov (b), Anton V. Anikin (c)
*Corresponding author

(a) Volgograd State Technical University, 28, Lenin Ave, Volgograd, Russia, oasychev@gmail.com
(b) Volgograd State Technical University, 28, Lenin Ave, Volgograd, Russia, hello-world-now@ya.ru
(c) Volgograd State Technical University, 28, Lenin Ave, Volgograd, Russia, anton@anikin.name

Abstract

Learning to read source code is an important part of developing programming skills. Using pseudo-code and program traces for this allows creating exercises developing comprehension of basic algorithmic concepts like control structures, expressions, functions calls. However, such assignments require reading long traces and comparing them with the algorithm, identifying the control structures involved. Automatic generation of algorithms for assessments poses a particular problem because such algorithms lack an identified goal. While this is useful for learning to predict program traces without relying on the intuitive understanding of the algorithm, it doesn't allow using meaningful labels for the control structures comprising the algorithm's tree. This paper presents a study of students' preferences between different ways of writing algorithms and program traces. Eighty-one first-year computer science students completed a survey comparing eight ways of writing algorithms and four ways of writing program traces. The students preferred using expressions to identify control structures over short phrases or numbers. They had no clear preferences for the way the program traces were formulated. So a system, generating algorithms for assessments should also have a module generating expressions. These results are limited to university students; the preferences of other groups of learners should be studied separately.

Keywords: Algorithm, e-learning, program trace

https://doi.org/10.15405/epsbs.2021.02.02.127
Corresponding Author: Oleg A. Sychev
Selection and peer-review under responsibility of the Organizing Committee of the conference
eISSN: 2357-1330

1. Introduction

Reading source code is an important part of a programmer's work which takes up to half of their working time (Kashima et al., 2015). Plenty of various tools assisting code understanding is developed, see (Kadar et al., 2016; Perez & Abreu, 2016), but they address the needs of experienced developers while learning to read and understand program code is one of the important stages of computer science education. Learning to understand code is difficult for introductory-courses students because it requires simultaneous learning of programming concepts, ways representing algorithms, and programming-language syntax. This problem has been addressed in many works, including (Gulec et al., 2019; Jawale et al., 2019; Karvelas, 2019; Kramer et al., 2019; Montagner et al., 2019; Swidan & Hermans, 2019; Zafar & Farooq, 2019). Paper (Malik, 2019) states that 50 % of students in the introductory programming course are "not good enough", while 63 % of students were rated as "medium". In spite of the limited significance of these results, they confirm the difficulty of learning to program and the need to improve the educational process.

In our opinion, a significant part of this difficulty is caused by the insufficient time spent on the assessments on the comprehension level of Bloom's taxonomy (Sychev et al., 2020). While it's easy to create quizzes and assignments for the remembering (knowledge) and application levels with automatic grading, the comprehension level often requires personal discussion between a student and a teacher. So a significant amount of novice programmers begin to write their first programs without developing strong comprehension first, which causes a high entry threshold. For programming, comprehension involves understanding what each construction does and how the algorithm will be executed while the application level is tied to the syntax of a particular programming language. Many automated quizzing systems operate at the knowledge level (simple quizzes) and at the application level (problem-solving). Some of them serve not only for evaluation but also to assist in learning. For example, (Sychev & Mamontov, 2018) describes a question type that provides advanced feedback, detecting mistakes like typos and missing, extraneous, and misplaced tokens, and giving hints on how they should be fixed. While good for learning the programming-language syntax, such systems do little to help develop the understanding of how a program executes.

A step-by-step log of commands executed by a program is called a program trace (or just a trace or tracing). The importance of the traces has been frequently mentioned earlier to both developing program code (Abualese et al., 2017; Fittkau et al., 2015; Feng et al., 2018; Kashima et al., 2015; Kadar et al., 2016; Khoury et al., 2017; Khoury et al., 2019; Perez & Abreu, 2016) and programming education (Aggarwal et al., 2018; Paaßen et al., 2016; Srinivasan et al., 2016; Terada, 2005).

Usually, the student learns to understand the program by changing the code and observing changes in program output. Thus they extract implicit information about the sequence of acts (the trace) executed by the program while running and compares it with the program code in their mind. Most development environments allow tracing of code, but that requires student's actions to perform, and a few students do it as the grade is often given for the program that gives the correct output, not for the trace. A novice programmer often omits the understanding phase and just tries to change their program in hopes of getting the desired output: this strategy works while the programming assignments are simple. Using

https://doi.org/10.15405/epsbs.2021.02.02.127
Corresponding Author: Oleg A. Sychev
Selection and peer-review under responsibility of the Organizing Committee of the conference
eISSN: 2357-1330

reasoning by analogy is also common for beginners. However, if a student fails, it is impossible to determine without additional data whether it is a comprehension error (wrong use of control structures) or application error (mistake in the programming-language syntax). A teacher has to ask questions in order to understand the true cause of an error and give the correct explanation. This process limits the usefulness of automatic quizzing and testing systems for introductory programming courses. It so limits the number of assignments a student can perform as the teacher's time is limited.

For reducing the impact of the level of application, we aim to exclude the use of a particular programming language and execution of the program, showing only its output. Instead, we represent an algorithm as a unified pseudo-code that includes a minimum of details, focusing on the essence of the task, and show the program behaviour as a text trace. Thus, the dynamic properties of the algorithm – the previously hidden key element of understanding, with which the vast majority of students experience problems – come to the fore.

In the context of a learning management system (LMS), tasks with explanations that help the learning can be implemented by automatically generating and checking test tasks. We plan to use a generator of random algorithms and their traces to create unique tasks for each student. However, a random algorithm is meaningless: it doesn't achieve any conceivable goal. For learning the concepts of algorithm and control structures, the absence of the purpose of the program is an advantage (Qi & Fossati, 2020), because it prevents students from assuming anything based on everyday experience and analogy, forcing them to use programming knowledge only. The downside of this approach is that the program traces are long (Whitington & Ridge, 2017) and, contain excess details (Trumper et al., 2013) and require careful reading. There have been many attempts to simplify the perception of traces through various representations. Parer (Abualese et al., 2017) attempts to simplify the trace using a generalizing transformation, "unwinding". In (Whitington & Ridge, 2017) a subset of program traces (like a calculation of one-liner pure functions composition) is discussed including a line-by-line textual visualization of functional computations as mathematical expressions with substitutions of function formulas and sub-expressions convolution into intermediate results. An interesting approach, producing attractive graphics and handling really large call traces, is considered in (Trumper et al., 2013). The authors try to compare the traces of the program visually so that semantically different ways of program execution would look easily distinguishable, while similar call traces wouldn't. The article (Almadhun et al., 2019) describes a pseudo-code and activity-diagram generator (reverse engineering from Visual Basic) for teaching high school students. Their pseudo-code, according to the screenshot in their article, is just code cleared from VB directives. However, this doesn't allow to check their comprehension before they write programming code A gaming form of training in the form of a web application is described in work (Beckwith & Ahmed, 2018), where a visual interactive Binary Search Tree is used as a basis for visualization, and pseudo-code is provided as a comment of the currently displayed action. The article emphasizes self-study at an individual pace. PCIL – PseudoCode Interpreted Language is introduced in (Malone et al., 2009). It is intended for direct visualization of the algorithm and formulation of various pedagogical tasks over an algorithm. Some features of the proposed language may be adopted in further research and education as good practices. These include i) indentation of nested blocks of code; ii) sentence-like rows of some actions (that overcomes the usual poor readability of such constructs in popular programming languages); iii) explicit endings of control structures to close the nesting level; iv) familiar form of frequent actions like assignments and arithmetics. The properties of a good algorithm visualization are discussed in the article (Saraiya et al., 2004). The number of stable properties (i.e. positively correlated with the effect) was found to be quite small. As the most confident, the ability to directly control the pace of visualization (which is relevant for the animated view) is highlighted. As for the other properties of the algorithm visualization considered by the authors, they were not considered essential in general. So more research is needed to find the best way to write pseudo-code and program traces.

2. Problem Statement

The automatic generation of programs allows developing a system to teach programming skills on Comprehension level of Bloom's taxonomy. However, automatically generated programs lack an intuitive goal, but students still need some way to identify the control structures and find relevant trace rows. Given that program traces in exercises can have a significant size, the way to designate individual control structures should be as convenient as possible. So our research was aimed at studying which ways to identify control structures and write program trace work better for students.

3. Research Questions

3.1. Question #1

What is the most convenient way of identifying control structures in an algorithm and its trace when the algorithm lacks an intuitive goal?

3.2. Question #2

Which way to write trace elements is more convenient?

4. Purpose of the Study

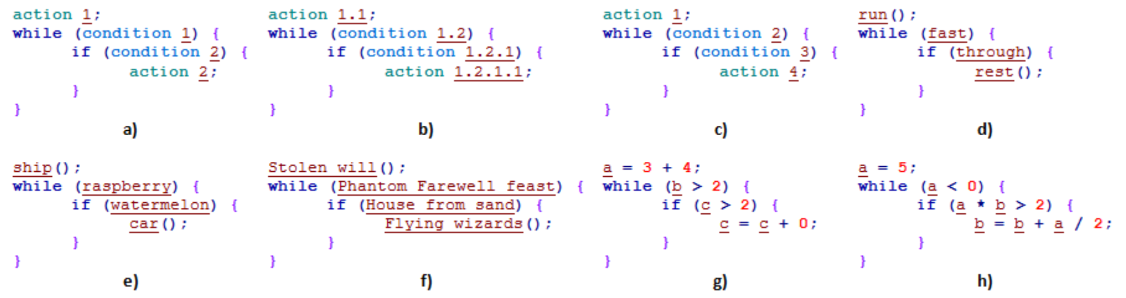
The purpose of this work is to find out which forms of identifying control structures in the algorithm and trace sentences are most convenient for 1st-grade computer science students for a randomly-generated algorithm.

5. Research Methods

5.1. Survey

To find out how students perceive algorithms and their traces, we created a survey showing different ways of writing them. The survey has four steps, each requesting to rank alternatives shown on the page. The first three steps showed three different control-structure trees with corresponding traces written in eight ways each, asking the students to rank these alternatives.

https://doi.org/10.15405/epsbs.2021.02.02.127
Corresponding Author: Oleg A. Sychev
Selection and peer-review under responsibility of the Organizing Committee of the conference
eISSN: 2357-1330



```
action 1;
while (condition 1) {
        if (condition 2) {
                action 2;
        }
}
```
a)

```
action 1.1;
while (condition 1.2) {
        if (condition 1.2.1) {
                action 1.2.1.1;
        }
}
```
b)

```
action 1;
while (condition 2) {
        if (condition 3) {
                action 4;
        }
}
```
c)

```
run();
while (fast) {
        if (through) {
                rest();
        }
}
```
d)

```
ship();
while (raspberry) {
        if (watermelon) {
                car();
        }
}
```
e)

```
Stolen will();
while (Phantom Farewell feast) {
        if (House from sand) {
                Flying wizards();
        }
}
```
f)

```
a = 3 + 4;
while (b > 2) {
        if (c > 2) {
                c = c + 0;
        }
}
```
g)

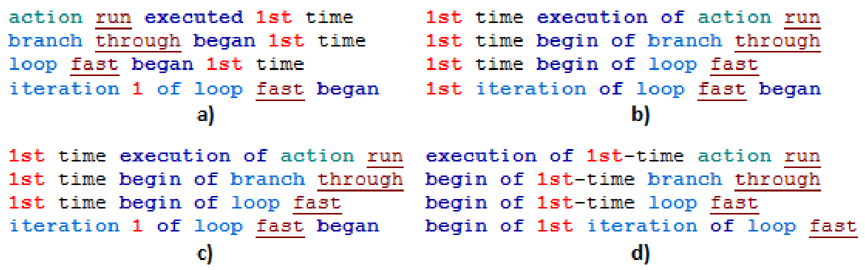
```
a = 5;
while (a < 0) {
        if (a * b > 2) {
                b = b + a / 2;
        }
}
```
h)

Figure 1. Identifying styles for control structures a)-h): shortened examples of algorithms

The alternatives differed in the way the control structures and their elements were identified. Figure 01 shows eight alternative ways to write an algorithm (shortened algorithms translated to English while the survey was in Russian).

a) Numbering by entity type (simple actions, loops, conditions have their enumeration each);
b) Hierarchical through-numbering representing control-structures nesting;
c) Unified through-numbering (each entity gets a consecutive number);
d) Single-word by the part of speech: a random verb for action, a random adjective for a condition;
e) Single-word by an entity: a random vehicle for action, a random plant for a condition;
f) Random phrase labelling;
g) Programming language expressions including single-character identifiers, integer constants, and binary operators;
h) The same as above but the pseudo-code is more syntactically correct (no uninitialized variable is accessed); the identifiers are the same.

The fourth step of the survey shows four variants of the trace representation. In Russian, changing word order of a sentence changes only the emphasis. However, it often leaves the meaning intact, so different variants of the same sentences were supposed to find the way that is most convenient to the students. This process is especially crucial because the traces tend to be significantly longer than algorithms. Figure 02 shows four alternative ways to write traces, rendered in English as close as possible to their Russian equivalents.



```
action run executed 1st time
branch through began 1st time
loop fast began 1st time
iteration 1 of loop fast began
```
a)

```
1st time execution of action run
1st time begin of branch through
1st time begin of loop fast
1st iteration of loop fast began
```
b)

```
1st time execution of action run
1st time begin of branch through
1st time begin of loop fast
iteration 1 of loop fast began
```
c)

```
execution of 1st-time action run
begin of 1st-time branch through
begin of 1st-time loop fast
begin of 1st iteration of loop fast
```
d)

Figure 2. Trace word ordering styles a)-d): examples of trace lines

The structure of a trace row includes three parts: the subject (action or control structure), the type of action (beginning, ending, or execution), and the time it appears over the trace. The trace styles had a

https://doi.org/10.15405/epsbs.2021.02.02.127
Corresponding Author: Oleg A. Sychev
Selection and peer-review under responsibility of the Organizing Committee of the conference
eISSN: 2357-1330

different order of these three parts, allowing students to concentrate first on different things. In order to ensure that the chosen style would help reading long traces, a complex 31-line trace was used for the survey.

Four trace styles may be formally defined as follows:

a) <object> – <action> – <nth time>: <object> is always first.
b) <nth time> – <action> – <object>: <nth time> is first for atomic actions; the loop's iteration is enumerated without "time" word.
c) The same as 2 but the format of loop's iteration begin/end row follows the 1st variant's scheme (`iteration 1` instead of `1st iteration`).
d) <action> – <nth time> – <object>: <action> is always first. The emphasis in Russian is put on the <object> part.

Each step of the survey presents the alternatives and asks a student to rank them. The survey audience consisted of 81 first-year computer science students who have completed the course Informatics (the introductory course about algorithms) and learned the basics of the C language.

5.2. Analysis of survey's results

To aggregate assessors' scores, four voting election algorithms were chosen:

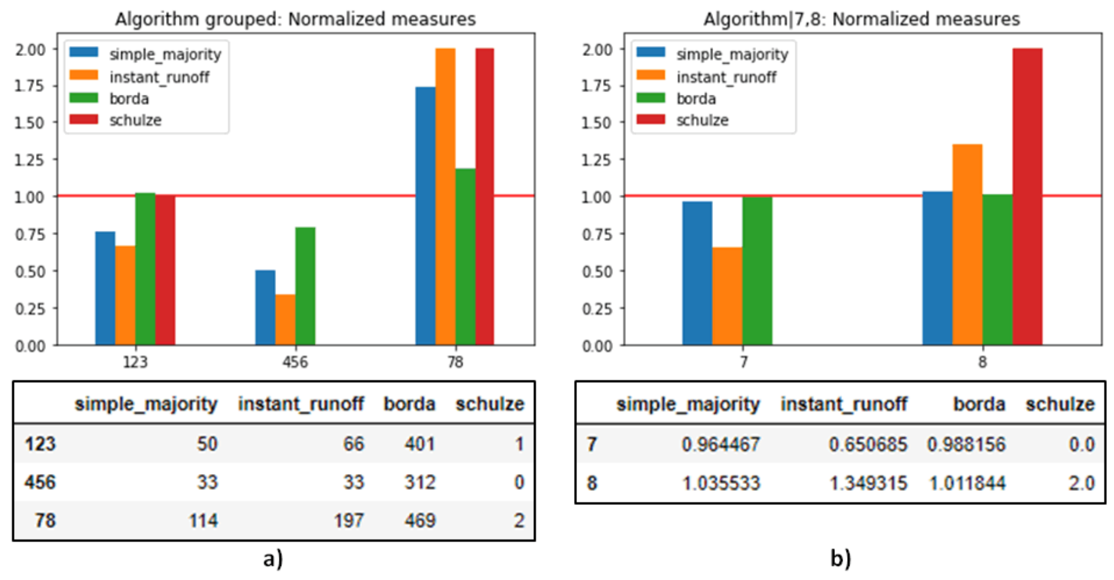
a) A simple majority (a.k.a. Majority judgment) is simply the number of first votes;
b) Instant-runoff voting (a.k.a. ranked-choice voting) (Blom et al., 2015);
c) Borda count method is still actual as recent research (Kondratev et al., 2019; Kurz et al., 2019; P’osfai et al., 2019) shows;
d) Schulze method (a.k.a. Schwartz Sequential dropping, beat path method) (Schulze, 2018).

These methods were chosen because of the research questions, the preferential nature of the data, and the low rate of invalid (non-incremental) responses. Some of the alternative styles that we have offered to assessors are similar, so we grouped them into subsets. This prevents the alternatives from subsets with uneven distribution of votes from winning over the alternatives with more popular subsets with even distribution of votes. At first, we compared the subsets, using the best alternative from the subset for the particular student as the subset's score. Then, the best alternative from the best subset can be selected. We formed three subsets for algorithm styles, using numbers includes the variants (alternatives a,b,c), words and collocations – (alternatives d,e,f), and expressions (alternatives g,h). For trace styles, variants 2 and 3 differ only in iteration row form, so we also group them, leaving variants 1 and 4 as separate subsets.

6. Findings

The three pages of the survey gave us 197 rankings of eight alternatives to write an algorithm. The last page produced 72 rankings comparing word order in program traces. Each ranking consisted of n integers representing the places assigned to fixed-order alternatives.

https://doi.org/10.15405/epsbs.2021.02.02.127
Corresponding Author: Oleg A. Sychev
Selection and peer-review under responsibility of the Organizing Committee of the conference
eISSN: 2357-1330

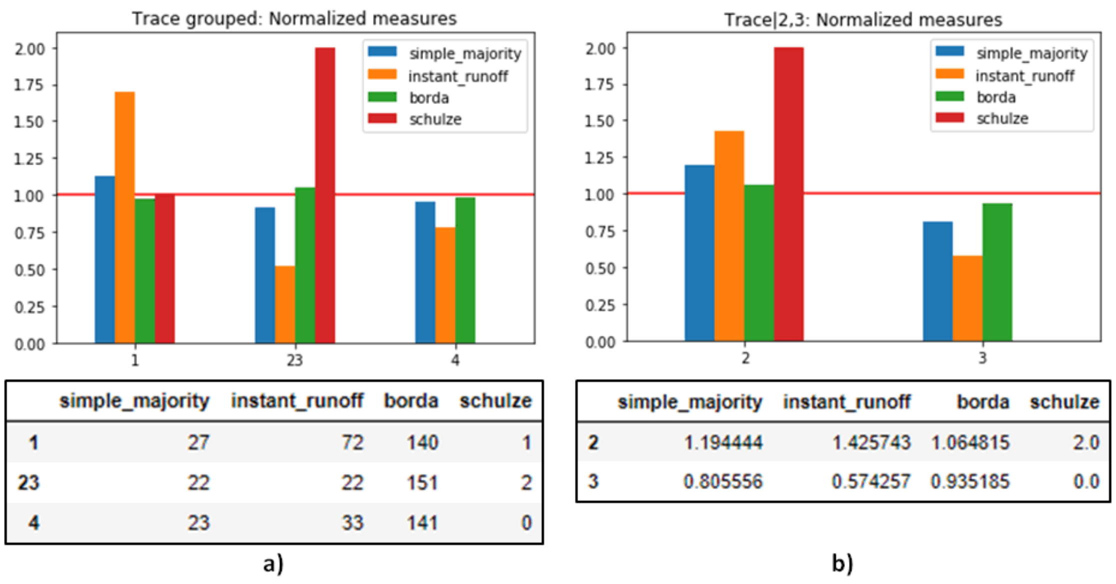


	simple_majority	instant_runoff	borda	schulze
123	50	66	401	1
456	33	33	312	0
78	114	197	469	2

	simple_majority	instant_runoff	borda	schulze
7	0.964467	0.650685	0.988156	0.0
8	1.035533	1.349315	1.011844	2.0

Figure 3. Identifying styles for control structures : a) results for subsets (123: styles a,b,c – numbers, 456: styles d,e,f – words, 78: styles g,h – expressions), b) results for styles in the best subset

Aggregated survey results for the subsets of the ways to write algorithms are shown in figure 03 and visualized as bar charts for each calculation method. Similarly, the results concerning the word order in program trace pictured in figure 04. For making the charts better readable, the visualized values were normalized so that the average value for each metric is 1. So the red horizontal line indicates the value of 1 to see the superiority easily.



	simple_majority	instant_runoff	borda	schulze
1	27	72	140	1
23	22	22	151	2
4	23	33	141	0

	simple_majority	instant_runoff	borda	schulze
2	1.194444	1.425743	1.064815	2.0
3	0.805556	0.574257	0.935185	0.0

Figure 4. Trace word ordering styles: a) results for subsets (1: trace style a – object first, 23: trace styles b and c – nth time first, 4: trace style d – action first), b) results for the styles in subset 23.

Figure 03 shows that subset 78 (expressions) has a clear advantage over the other subsets, receiving maximum value from every method (figure 05). Comparing the alternatives inside this group, we can see that the 8th style is preferred over the 7th. So we can conclude that most students prefer to use

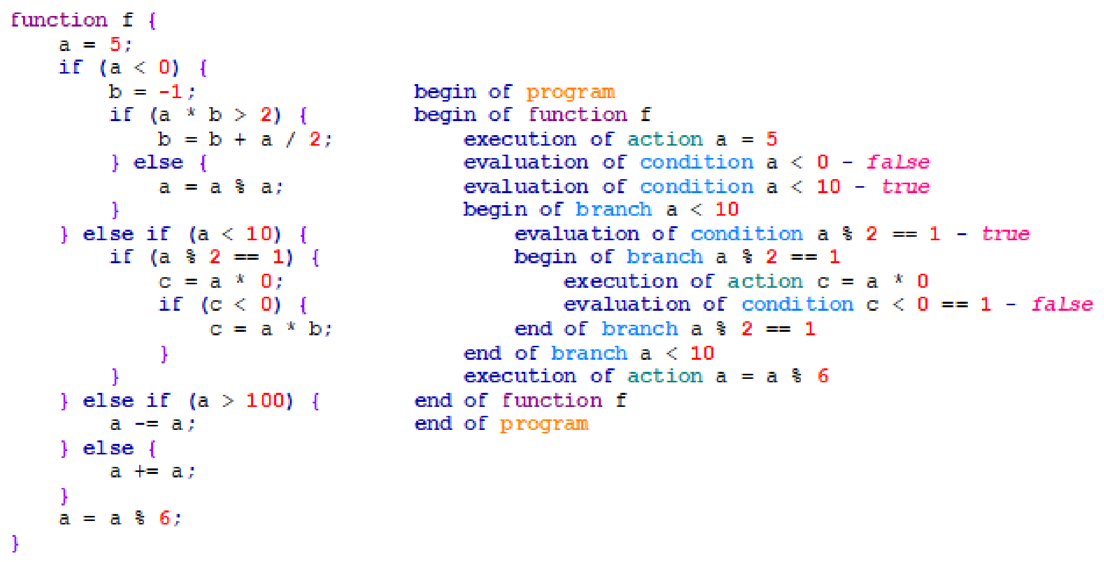
https://doi.org/10.15405/epsbs.2021.02.02.127
Corresponding Author: Oleg A. Sychev
Selection and peer-review under responsibility of the Organizing Committee of the conference
eISSN: 2357-1330

expressions to identify control structures, bringing pseudo-code closer to the programming-language code. The students chose a well-known but complicated way to identify control structures over simpler alternatives. The selected style of the algorithm has disadvantages. In real algorithms, the same expressions are often used in different operators, which is unacceptable for the assignments where each program trace elements must identify the relevant part of the algorithm. Generating expressions and maintaining a relatively small amount of variables that are initialized before use is a more complex task for the algorithm generator than numbering control structures or using a limited vocabulary of phrases to identify them. However, when reading long traces, students' convenience is important.

According to figure 04 (a), there is no consistency in the indications of different calculation methods. Each alternative is the worst according to the results of at least one of the methods. The likely reason for this is the lack of consensus among respondents about which order is better. The students stated that the word order in program trace is not important for reading it. Inside the subset 23, the respondents strongly preferred alternative 2, which was more consistent than the other. This supports the conclusion that uniformity is important for the students.

7. Conclusion

In this paper, we compared different ways to write algorithms and an algorithm's traces for introductory programming courses. The survey of first-year computer science students showed that even for automatically-generated algorithms without an intuitive goal written in pseudo-code, they prefer to identify control structures by expressions, not by numbers or small phrases. The comparison of four ways of writing program traces was inconclusive; this shows that the students don't have any clear preference between the order of words in the program trace. In all cases, the participants chose more uniform ways of writing algorithms and program traces, minimizing the amount of the different types of objects they are dealing with.



```
function f {
    a = 5;
    if (a < 0) {
        b = -1;
        if (a * b > 2) {
            b = b + a / 2;
        } else {
            a = a % a;
        }
    } else if (a < 10) {
        if (a % 2 == 1) {
            c = a * 0;
            if (c < 0) {
                c = a * b;
            }
        }
    } else if (a > 100) {
        a -= a;
    } else {
        a += a;
    }
    a = a % 6;
}
```

```
begin of program
begin of function f
    execution of action a = 5
    evaluation of condition a < 0 - false
    evaluation of condition a < 10 - true
    begin of branch a < 10
        evaluation of condition a % 2 == 1 - true
        begin of branch a % 2 == 1
            execution of action c = a * 0
            evaluation of condition c < 0 == 1 - false
        end of branch a % 2 == 1
    end of branch a < 10
    execution of action a = a % 6
end of function f
end of program
```

Figure 5. Identifying style #8: complete survey example of algorithm and execution trace.

https://doi.org/10.15405/epsbs.2021.02.02.127
Corresponding Author: Oleg A. Sychev
Selection and peer-review under responsibility of the Organizing Committee of the conference
eISSN: 2357-1330

These results will be used in further research on creating a system to generate algorithmic questions and grade them automatically. The study shows that generating just control-structure trees is insufficient: the generator should also generate expressions to identify the control structures. However, these results were obtained for university students. The other categories of introductory-programming learners may have other preferences, which also needs further study.

Acknowledgments

The reported study was funded by RFBR, project number 20-07-00764.

References

Abualese, H., Sumary, P., Al-Rousan, T., & Al-Mousa, M. R. (2017). A trace simplification framework. *Proc. 8th Int. Conf. on Inform. Technol. (ICIT 2017)* (pp. 464–468). https://doi.org/10.1109/ICITECH.2017.8080043

Aggarwal, A., Touretzky, D. S., & Gardner-McCune, C. (2018). Demonstrating the ability of elementary school students to reason about programs. *Proc. of the 49th ACM Techn. Symp. on Computer Sci. Ed. (SIGCSE 2018)*, *2018-Janua* (pp. 735–740). https://doi.org/10.1145/3159450.3159488

Almadhun, S., Toycan, M., & Adalier, A. (2019). VB2ALGO: An educational reverse engineering tool to enhance high school students' learning capacities in programming. *Revista de Cercetare Si Interventie Sociala*, *67* (December), 67–87. https://doi.org/10.33788/rcis.67.5

Beckwith, B., & Ahmed, D. (2018). Gamification of undergraduate computer science data structures. *Proc. Int. Conf. on Computat. Sci. and Computat. Intellig. (CSCI-2018)* (pp. 648–650). https://doi.org/10.1109/CSCI46756.2018.00129

Blom, M. L., Stuckey, P. J., Teague, V., & Tidhar, R. (2015). Efficient Computation of Exact {IRV} Margins. *CoRR*, *abs/1508.0*. http://arxiv.org/abs/1508.04885

Feng, Y., Dreef, K., Jones, J. A., & Van Deursen, A. (2018). Hierarchical abstraction of execution traces for program comprehension. *Proc. Int. Conf. on Software Engineer.* (pp. 86–96). https://doi.org/10.1145/3196321.3196343

Fittkau, F., Finke, S., Hasselbring, W., & Waller, J. (2015). Comparing Trace Visualizations for Program Comprehension through Controlled Experiments. *IEEE Int. Conf. on Program Comprehension*, *2015-Augus* (pp. 266–276). https://doi.org/10.1109/ICPC.2015.37

Gulec, U., Yilmaz, M., Yalcin, A. D., O'Connor, R. V., & Clarke, P. M. (2019). CENGO: A Web-Based Serious Game to Increase the Programming Knowledge Levels of Computer Engineering Students. *Communicat. in Comp. and Inform. Sci.*, *1060*, 237–248. https://doi.org/10.1007/978-3-030-28005-5_18

Jawale, M. A., Pawar, A. B., & Kyatanavar, D. N. (2019). Smart python coding through voice recognition. *Int. J. of Innovat. Technol. and Explor. Engineer.*, *8*(10), 3283–3285. https://doi.org/10.35940/ijitee.J1207.0881019

Kadar, R., Syed-Mohamad, S. M., & Abdul Rashid, N. (2016). Semantic-based extraction approach for generating source code summary towards program comprehension. *9th Malaysian Software Engineer. Conf. (MySEC-2015)* (pp. 129–134). https://doi.org/10.1109/MySEC.2015.7475208

Karvelas, I. (2019). Investigating novice programmers' interaction with programming environments. *Annual Conf. on Innovat. and Technol. in Computer Sci. Ed. (ITiCSE)* (pp. 336–337). https://doi.org/10.1145/3304221.3325596

Kashima, Y., Ishio, T., Etsuda, S., & Inoue, K. (2015). Variable data-flow graph for lightweight program slicing and visualization. *IEICE Transact. on Inform. and Syst.*, *E98D*(6), 1194–1205. https://doi.org/10.1587/transinf.2014EDP7395

Khoury, R., Hamou-Lhadj, A., Rahim, M. I., Halle, S., & Petrillo, F. (2019). Triade a three-factor trace segmentation method to support program comprehension. *Proc. IEEE 30th Int. Symp. on Software*

Reliab. Engineer. Workshops (ISSREW-2019) (pp. 406–413). https://doi.org/10.1109/ISSREW.2019.00103

Khoury, R., Shi, L., & Hamou-Lhadj, A. (2017). Key elements extraction and traces comprehension using gestalt theory and the helmholtz principle. *Proc. IEEE Int. Conf. on Software Maintenance and Evolut. (ICSME-2016)* (pp. 478–482). https://doi.org/10.1109/ICSME.2016.24

Kondratev, A. Y., Ianovski, E., & Nesterov, A. S. (2019). How should we score athletes and candidates: geometric scoring rules. *CoRR*, *abs/1907.0*. http://arxiv.org/abs/1907.05082

Kramer, M., Barkmin, M., & Brinda, T. (2019). Identifying predictors for code highlighting skills a regressional analysis of knowledge, syntax abilities and highlighting skills. *Annual Conf. on Innovat. and Technol. in Computer Sci. Ed. (ITiCSE)* (pp. 367–373). https://doi.org/10.1145/3304221.3319745

Kurz, S., Mayer, A., & Napel, S. (2019). Influence in Weighted Committees. *ArXiv*, *abs/1912.1*.

Malik, S. I. (2019). Assessing the teaching and learning process of an introductory programming course with bloom's taxonomy and Assurance of Learning (AOL). *Int. J. of Inform. and Communicat. Technol. Ed.*, *15*(2), 130–145. https://doi.org/10.4018/IJICTE.2019040108

Malone, B., Atkison, T., Kosa, M., & Hadlock, F. (2009). Pedagogically effective effortless algorithm visualization with a PCIL. *Proc. Frontiers in Ed. Conf. (FIE)*. https://doi.org/10.1109/FIE.2009.5350481

Montagner, I. S., Ferrao, R. C., Marossi, E., & Ayres, F. J. (2019). Teaching C programming in context: A joint effort between the Computer Systems, Embedded Computing and Programming Challenges courses. *Proc. Frontiers in Ed. Conf. (FIE), 2019-Octob*. https://doi.org/10.1109/FIE43999.2019.9028687

P'osfai, M., Braun, N., Beisner, B. A., McCowan, B., & D'Souza, R. M. (2019). *Consensus ranking for multi-objective interventions in multiplex networks*.

Paaßen, B., Jensen, J., & Hammer, B. (2016). Execution traces as a powerful data representation for intelligent tutoring systems for programming. *Proc. of the 9th Int. Conf. on Ed. Data Mining (EDM-2016)* (pp. 183–190). https://www.scopus.com/inward/record.uri?eid=2-s2.0-85058054798&partnerID=40&md5=cac86535d6058c3840058c03b2978fa8

Perez, A., & Abreu, R. (2016). Framing program comprehension as fault localization. *J. of Software: Evolut. and Process*, *28*(10), 840–862. https://doi.org/10.1002/smr.1799

Qi, R., & Fossati, D. (2020). Unlimited trace tutor: Learning code tracing with automatically generated programs. *Annual Conf. on Innovat. and Technol. in Comp. Sci. Ed. (ITiCSE)* (pp. 427–433). https://doi.org/10.1145/3328778.3366939

Saraiya, P., Shaffer, C. A., McCrickard, D. S., & North, C. (2004). Effective features of algorithm visualizations. *Proc. of the SIGCSE Techn. Symp. on Computer Sci. Ed.* (pp. 382–386). https://www.scopus.com/inward/record.uri?eid=2-s2.0-2642570286&partnerID=40&md5= fef7210798bae64ed7a1e7a0e7833227

Schulze, M. (2018). The Schulze Method of Voting. *CoRR*, *abs/1804.0*. http://arxiv.org/abs/1804.02973

Srinivasan, M., Lee, Y., & Yang, J. (2016). Enhancing object-oriented programming comprehension using optimized sequence diagram. *Proc. IEEE 29th Conf. on Software Engineer. Education and Training (CSEEandT 2016)* (pp. 81–85). https://doi.org/10.1109/CSEET.2016.37

Swidan, A., & Hermans, F. (2019). The Effect of Reading Code Aloud on Comprehension: An Empirical Study with School Students. *Proc. of the ACM Conf. on Global Comput. Ed. (CompEd 2019)* (pp. 178–184). https://doi.org/10.1145/3300115.3309504

Sychev, O., Anikin, A., & Prokudin, A. (2020). Automatic grading and hinting in open-ended text questions. *Cognit. Syst. Res.*, *59*, 264–272. https://doi.org/10.1016/j.cogsys.2019.09.025

Sychev, O. A., & Mamontov, D. P. (2018). Automatic error detection and hint generation in the teaching of formal languages syntax using correctwriting question type for moodle LMS. *Proc. of the 3rd Russian-Pacific Conf. on Computer Technol. and Applicat (RPC 2018).* https://doi.org/10.1109/RPC.2018.8482125

Terada, M. (2005). ETV: A program trace player for students. *Proc. of the 10th Annual SIGCSE Conf. on Innovat. and Technol. in Computer Sci. Ed.* (pp. 118–122). https://doi.org/10.1145/1067445.1067480

https://doi.org/10.15405/epsbs.2021.02.02.127
Corresponding Author: Oleg A. Sychev
Selection and peer-review under responsibility of the Organizing Committee of the conference
eISSN: 2357-1330

Trumper, J., Dollner, J., & Telea, A. (2013). Multiscale visual comparison of execution traces. *IEEE Int. Conf. on Program Comprehension* (pp. 53–62). https://doi.org/10.1109/ICPC.2013.6613833

Whitington, J., & Ridge, T. (2017). Visualizing the evaluation of functional programs for debugging. *Open Access Ser. in Inform.*, *56*. https://doi.org/10.4230/OASIcs.SLATE.2017.7

Zafar, S., & Farooq, M. S. (2019). A graphical methodology to promote programming language concepts in novice. *3th Int. Conf. on Innovat. Comput. (ICIC-2019)*. https://doi.org/10.1109/ICIC48496.2019.8966727