Automated Correction for Trace Tables in a CS1 Course

Fernando Teubl, Francisco Zampirolli

[1]Federal University of ABC (UFABC)
Av. dos Estados, 5001 – Santo André – 09210-580 – SP – Brazil

{fernando.teubl,fzampirolli}@ufabc.edu.br

Abstract. *This work presents a remote, asynchronous, parametric, and automatically corrected evaluation based on the Trace Table method for an introductory programming course. Moodle was used as the learning platform, VPL as the evaluation module/interface, and MCTest as the test generator. The Trace Table method allows for the analysis of students' knowledge in pseudocode, without relying on any specific programming language. The answers were formatted in a table, and the evaluations were automatically corrected during the student's problem-solving process. Two evaluations were applied throughout the course: an intermediate and a final. The course had 309 students and the results indicated a high acceptance rate, with over 80% approval of the method.*

1. Introduction

In recent years, the number of courses offered in the remote modality has increased considerably, especially during and after the COVID-19 pandemic [Combéfis 2022]. The Information Processing (CS1) course at UFABC aims to teach programming to students and is offered both in-person and remotely. The CS1 course is an introductory discipline in the Bachelor of Computer Science and covers the following topics: (1) Conditional Structures; (2) Repetition Structures; (3) Vectors and Matrices; and (4) Functions and Modularization. The course is divided into two parts: Theory and Practice. The theory is presented in pseudocode, while the practice adopts formal programming languages such as Python or Java.

In [Zampirolli et al. 2021b], work was presented for the same programming course that made the programming language used in the practical part more flexible, allowing the student to choose the language that best suited their needs or interests in each activity or evaluation. This course was offered in remote modality during the COVID-19 pandemic. The practical evaluations consisted of the student creating an algorithm with the chosen programming language in order to produce output according to a specific statement and input parameters so that no two tests were identical. The method of parameterized questions was presented in [Zampirolli et al. 2019].

During the theoretical part of the same course, evaluations involved writing an essay question and submitting it via Moodle. The questions were manually evaluated by the teacher after the end of the test, a process that could be extremely time-consuming in the case of large classes with hundreds of students.

This article proposes a mode of evaluation with automatic correction for the theoretical part of the CS1 course in remote modality. Considering that the goal of the theory is to teach programming structures, the use of a specific language is not recommended, and pseudocode is adopted instead. A Trace Table type question was used, which simulates each execution line manually in a table. The Trace Table evaluation allows for the assessment of the student's understanding of programming structures and their behavior.

This paper presents Section 2 on automatic evaluation in the theoretical part of the CS1 course. Section 3 presents our proposal. Section 4 presents the experiments conducted, and Section 5 presents the results. Finally, Section 6 presents the conclusions.

2. Background

In practical components of computer science courses, such as introductory programming classes like CS1, activities with automatic correction are commonly used [Combéfis 2022]. However, as discussed in [Paiva et al. 2022], automatic correction is more challenging to implement in the theoretical component. These activities are part of the smart learning content, electronically available [Brusilovsky et al. 2014].

To address this gap, [Zavala and Mendoza 2017] proposes that students should learn to read and manipulate code before writing it. They suggest three phases of learning to write programs: code comprehension, code manipulation, and code writing. However, most CS1 courses prioritize code writing and often neglect to assess students' code comprehension and manipulation skills. To remedy this, [Mendoza and Zavala 2018] implemented a two-week intervention strategy focused on code comprehension, with reading, tracing, and modifying existing programs, using Automatic Item Generation (AIG). The intervention showed benefits, and the authors conclude that providing students with opportunities to practice code comprehension and manipulation is crucial for students' overall programming skill development. Our article focuses on the first phase of learning, that is, on carrying out activities to develop the skills of understanding pseudocode, also using AIG.

In line with improving code comprehension, another useful technique is Tracing Tables, which are used to track variable changes during program execution. The paper [Risha and Brusilovsky 2020] introduces a tool that automatically generates Trace Tables from source code. The authors explore the potential of this tool by discussing future opportunities for adapting, providing feedback, and learning specifications. They report a pilot integration of the tool into an existing system, demonstrating its interoperability with a real-world use case. The tool was used by 115 students, and over 500 Trace Tables were completed. However, the authors did not report any feedback from the students, did not use pseudocode, and also did not use the tool to evaluate the students, as done in our proposal.

3. Method

The evaluations were conducted remotely and asynchronously, with questions generated and submitted through three different tools: Moodle, Virtual Programming Lab (VPL) [Rodríguez del Pino et al. 2010], and MCTest. MCTest was responsible for selecting, parameterizing, and distributing tests among students, ensuring that each student received a unique set of algorithm statements and input parameters to reduce plagiarism. Responses were submitted in a textual table format using VPL's editor, which is integrated into Moodle, and automatic correction provided each student with a grade and feedback for each verification execution. To ensure consistent behavior, both systems - MCTest and VPL - must contain a copy of the same code generator and grader, as they operate independently.

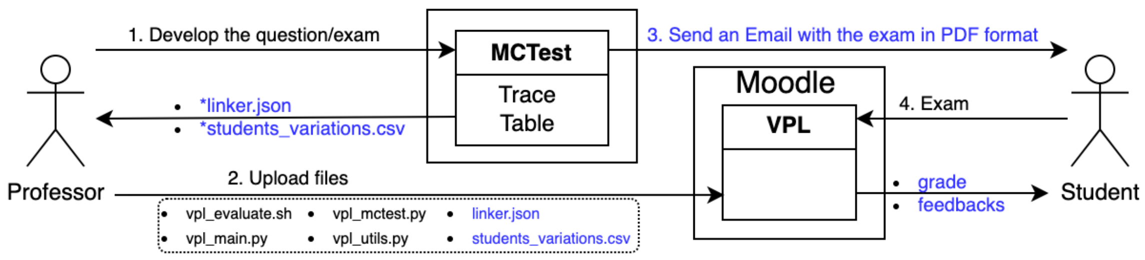


Fig. 1. Overview: Order of Steps and Automatic Files/Information in Blue.

The entire code for question parameterization and grading was developed in Python. An overview of the entire process is presented in the next section, with specific details provided in the following sections.

3.1. Overview

Figure 1 summarizes the entire process. The first step is for the Professor to develop the question on the MCTest platform. Then, the Professor uploads the Python files (Section 3.2) and the two files exported by MCTest (Section 3.3) to VPL. All the files exported to VPL are available at `http://professor.ufabc.edu.br/~fernando.teubl/files/sbie2023`. Once these two steps are completed, the Professor requests the MCTest to send the exam to the students, who access the VPL platform to take the exam and receive feedback with their grade.

3.2. Creating a Parametric Question

The questions were created using MCTest. The creation of a parametric question consists of providing a parameterized description and its respective answer, and then randomly selecting the questions, as presented in [Zampirolli et al. 2019]. The aim of this article is to create theoretical questions with automatic grading for an CS1 course. A Trace Table question consists of simulating line-by-line execution of an algorithm, annotating the values of the variables. Therefore, we opted to use Trace Table to focus solely on understanding the structures of a programming language, without depending on a specific language or programming environment.

Considering that the questions are parametric, there are several possible answers to the same question, depending on the randomly generated parameters. Manual creation of answer keys for each generated question is not feasible, so the answer key must also be automatically generated for each question variation. An auxiliary Python class called `TraceTable` was developed for generating answer keys and performing corrections of the questions. This class features the following methods:

constructor(`function`) Receives a Python function `function` that represents the algorithm, either as text or as an object.

source() Returns the function provided in the constructor as pseudocode. The student will only have access to the algorithm in pseudocode, not to the original in Python. The conversion is performed using several regular expression rules.

make(`args`) Creates a matrix that represents the Trace Table answer for the case where the `function` receives the arguments `args`. The values of the Trace Table are obtained with the help of the PyDev.Debugger library (`pydevd`), which uses `pydevd_tracing` to analyze the function's code line by line, storing the values in a table. The return is a matrix with the values of the Trace Table and the value returned by the respective function.

show() Prints the Trace Table in text mode. It must be called after `make` method and is used only for manual checking.

correct(`answer`) Returns the score of the Trace Table according to the answers provided by the student through the `answer` argument. In addition to the score, a textual feedback indicating possible errors is returned. The score is calculated by comparing the answer key with the student's answer and assigning a score linearly according to the matches.

The initial step in creating a question is to develop one or more algorithms in Python to be used in Trace Table. For each student, one algorithm will be randomly selected, and the more algorithms proposed by the teacher, the greater the diversity of the tests. Ideally, all algorithms should have equivalent complexities. An example of an algorithm is presented in Table 1.

Tab. 1. Comparison of Python code and pseudocode generated.

```
def Algorithm(v, m):
  a = [ 0, 0, 0, 0, 0 ]
  for i in range(0, len(m)):
    for j in range(0, len(m[i])):
      if a[v[i]] > m[i][j]:
        a[v[i]] -= m[i][j]
      else:
        a[v[i]] += m[i][j]
  return a
```

```
Function Algorithm(v, m)
| a ← [ 0, 0, 0, 0, 0 ]
| For i from 0 to m.length (not included), do:
| | For j from 0 to m[i].length (not included), do:
| | | If a[i] > m[v[i]][j], then:
| | | | a[i] ← a[i] - m[v[i]][j]
| | | Else:
| | | | a[i] ← a[i] + m[v[i]][j]
| Return a
```

In addition to randomly selecting the algorithm, a set of values is also randomly chosen as input arguments so that each test has a unique output, even when the same algorithm is repeated. It is important to note that the student does not receive the Python code, as shown in the left column of Table 1, but rather a pseudocode generated by the `TraceTable` class, as shown in the right column of Table 1.

After the random selection of the algorithm and input argument values, the question is formatted in PDF and sent individually to each student automatically through MCTest.

3.3. Integration with Moodle's VPL

When creating an evaluation using the MCTest, the system generates some files that can be exported to VPL, enabling integration between both systems. Below are the main files generated for importing into the Moodle VPL:

`linker.json` Contains a list of questions and their respective variations in JSON format. Each variation contains several attributes, with the main ones being: (a) **variant**, which has a unique integer value that identifies the variation; (b) **description**, which has the statement of the question in LaTeX and the Python code, such as presented in [Zampirolli et al. 2019]; and (c) **cases**, which has the inputs and their respective outputs of the test cases used in the automatic correction procedure of the VPL;

`students_variation.csv` A table in CSV format containing the name of the student and their assigned variation.

If the number of generated variations is equal to or greater than the number of students, MCTest associates the questions so that no two students have the same variation. In this article, the number of variations generated in each evaluation was the same as the number of students, so there was only one test per student.

The VPL originally requires a list of input cases with their respective expected outputs for correction. Although this information is provided by MCTest, the correction process is inefficient for the Trace Table approach used in this article. Since the Trace Table output is provided in a file in the `txt` format, it is necessary to interpret the response file to evaluate it. Therefore, the original VPL codes were replaced by new codes specific to this evaluation, which altered the default behavior of the VPL. The subsections below describe the changes made.

3.3.1. Evaluation Codes

The default file `vpl_evaluate.sh` was modified by the script below:

```
1  #!/bin/bash
2  . common_script.sh
3  get_source_files txt py
4  cat vpl_environment.sh > vpl_execution
5  python3 vpl_main.py > vpl_out
6  cat << EOS >> vpl_execution
7  cat vpl_out
8  EOS
9  chmod a+x vpl_execution
```

Line 2 includes the default VPL file `common_script.sh`, which contains several auxiliary functions. Line 3 defines to include the `.txt` files containing the student's response, noting that the evaluation is for Trace Table, not implementation. Line 4 creates the `vpl_execution` script with the default VPL environment variables. This script is originally responsible for running the evaluation but was modified to only print the evaluation result. The evaluation is actually performed in advance on line 5 through the `vpl_main.py` script, as described in Section 3.3.2. The evaluation result is stored in the `vpl_out` file, so when the VPL calls the `vpl_execution` script to perform the evaluation, it has already been performed, and it only needs to print the contents of the `vpl_out` file, as shown in line 7. This anticipation of the evaluation was necessary so that the correction process had full access to all files and environment variables. The VPL removes some files and environment variables when `vpl_execution` is executed for correction protection and isolation. For example, the student's program could access and modify the grade variable. Since the student's response is a text file, there is no risk of the student injecting malicious code. Line 9 finalizes the script by adding execution permission to `vpl_execution`.

3.3.2. Correction Codes

As presented in the previous subsection, the default VPL correction code was modified, so that all evaluation, score, and output text must be implemented. The `vpl_main.py` code was developed in Python and is responsible for all evaluation. The code is presented below.

```
import vpl_utils, vpl_MCTest, random, json
import numpy as np

mc = vpl_MCTest.MCTest()
qs = mc.getQuestions()

with open("qs['file'].txt", mode ='r') as file:
    TT = vpl_utils.TraceTable(qs['description'][0]['func'])
    TT.make(json.loads(qs['cases'][0]['input'][0]))
    score, feedback = TM.correct(file.read())

vpl_utils.terminate(score, {
```

```
13        'Question description' : qs['description'][0]['text'],
14        'pseudocode' : TT.source(),
15        'Score' : f"You got {int(100 * score)}%\n",
16        'Feedback' : feedback })
```

Line 4 uses an auxiliary class called `MCTest`, which is implemented in Python in `vpl_mctest.py`. This class provides information about the data generated by MCTest, specifically, by reading the `linker.json` and `students_variations.csv` files. Both files have already been described at the beginning of Section 3.3.

Line 5 obtains the specific question for the current student. The student's name is provided in a VPL environment variable, which must correspond to their respective variation in `students_variations.csv`. The specific student's question information is obtained from `linker.json`. There are two important parameters: `description` and `cases`, which represent the algorithm (in Line 8) and the input arguments of the algorithm (in Line 9), respectively, represented by a vector. Many keys in `linker.json` are lists, and therefore, there are `[0]` in Lines 8, 9, and 13.

Line 7 reads the user's response in the `Q1.txt` file of Trace Table. Line 8 instantiates a `TraceTable` class (already described in Section 3.2) with the respective algorithm drawn for the student. The `TraceTable` class is implemented in an auxiliary library called `vpl_utils`, which contains other useful functions as will be presented later.

Lines 9 and 10 create the answer key and correct the question, respectively. As a result, two variables are generated: (1) `score` with the student's score; and (2) `feedback`, with a textual return of the evaluation.

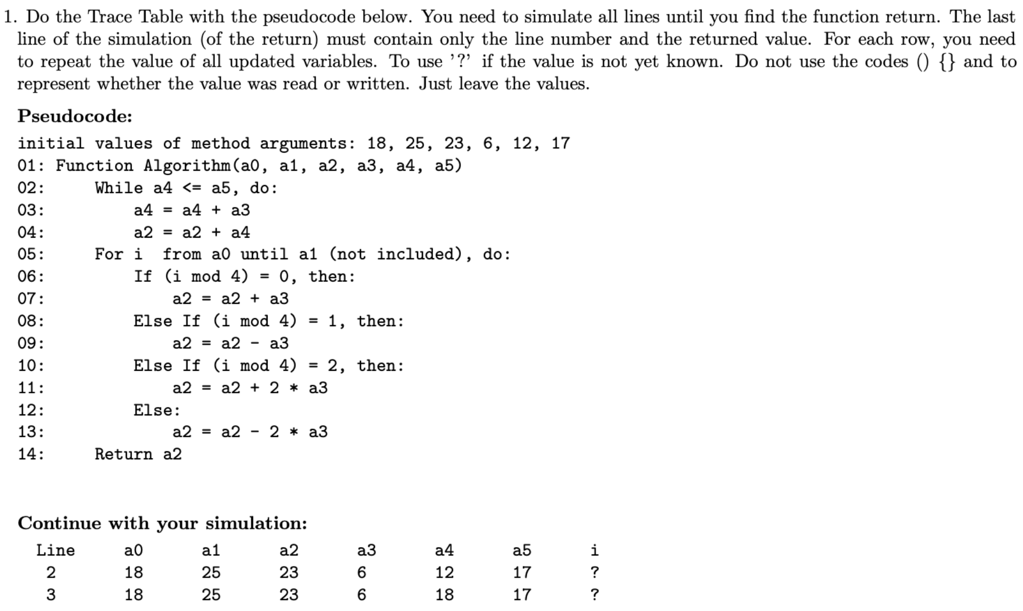
Finally, Line 12 (and later) ends the evaluation by generating the content that will be displayed to the student after the evaluation in the VPL interface. The `terminate` function, implemented in `vpl_utils.py`, receives the score (variable `score`) and a dictionary with the data that will be displayed to the student, with the key being the item name and the value being the description. This information is processed in a format compatible with the expected interpretation by VPL. Note that the statement and pseudocode are repeated (Line 13). Thus, if the student has not received the test by email, in PDF format by the `MCTest` system, he or she can still take the test. The score in percentage and the textual return of the evaluation are also presented (Lines 15 and 16).

4. Evaluation

Two exams were given to a class of 352 enrolled students, with 309 active students and 281 graduates. The students were divided into 12 classes, originally with 30 students each. The first exam, held in the middle of the academic period, covered simple conditional and repetition structures. The second exam, held at the end of the academic period, added vector and matrix manipulation. The following subsections will describe each exam in more detail.

4.1. First Exam

The first exam consisted of creating a Trace Table for an algorithm with simple conditional and repetition structures. Integer-type input arguments were drawn, and the student had to simulate step by step the code execution. Figure 2 shows an example of a question sent to the student via email in PDF format. The structure of the Trace Table, as well as the first two lines, were provided to the student.



1. Do the Trace Table with the pseudocode below. You need to simulate all lines until you find the function return. The last line of the simulation (of the return) must contain only the line number and the returned value. For each row, you need to repeat the value of all updated variables. To use '?' if the value is not yet known. Do not use the codes () {} and to represent whether the value was read or written. Just leave the values.

Pseudocode:

```
initial values of method arguments: 18, 25, 23, 6, 12, 17
01: Function Algorithm(a0, a1, a2, a3, a4, a5)
02:     While a4 <= a5, do:
03:         a4 = a4 + a3
04:         a2 = a2 + a4
05:     For i  from a0 until a1 (not included), do:
06:         If (i mod 4) = 0, then:
07:             a2 = a2 + a3
08:         Else If (i mod 4) = 1, then:
09:             a2 = a2 - a3
10:         Else If (i mod 4) = 2, then:
11:             a2 = a2 + 2 * a3
12:         Else:
13:             a2 = a2 - 2 * a3
14:     Return a2
```

Continue with your simulation:

```
 Line    a0     a1     a2     a3     a4     a5     i
  2      18     25     23     6      12     17     ?
  3      18     25     23     6      18     17     ?
```

Fig. 2. Example of question from the first exam, sent by email.

Figure 3 shows the VPL interface for the exam response. Note that the student only has access to a text file (Q1.txt, which is a Trace Table). When evaluated, the correction system calculates the score based on the number of lines identical to the answer key. In each exam, the student was presented with a Proposed Score and Comments section, which contains the question description, the algorithm, the percentage of correct answers, and textual feedback indicating if there are any invalid lines or other structural errors in the response.

4.2. Second Exam

The second exam was similar to the first exam. Complex repetition structures and vector and matrix variables were added. Figure 4 shows an example of the question. In this exam, it was decided to send the algorithm in Python instead of pseudocode (see Figure 4). In the VPL interface, we have kept only the pseudocode.

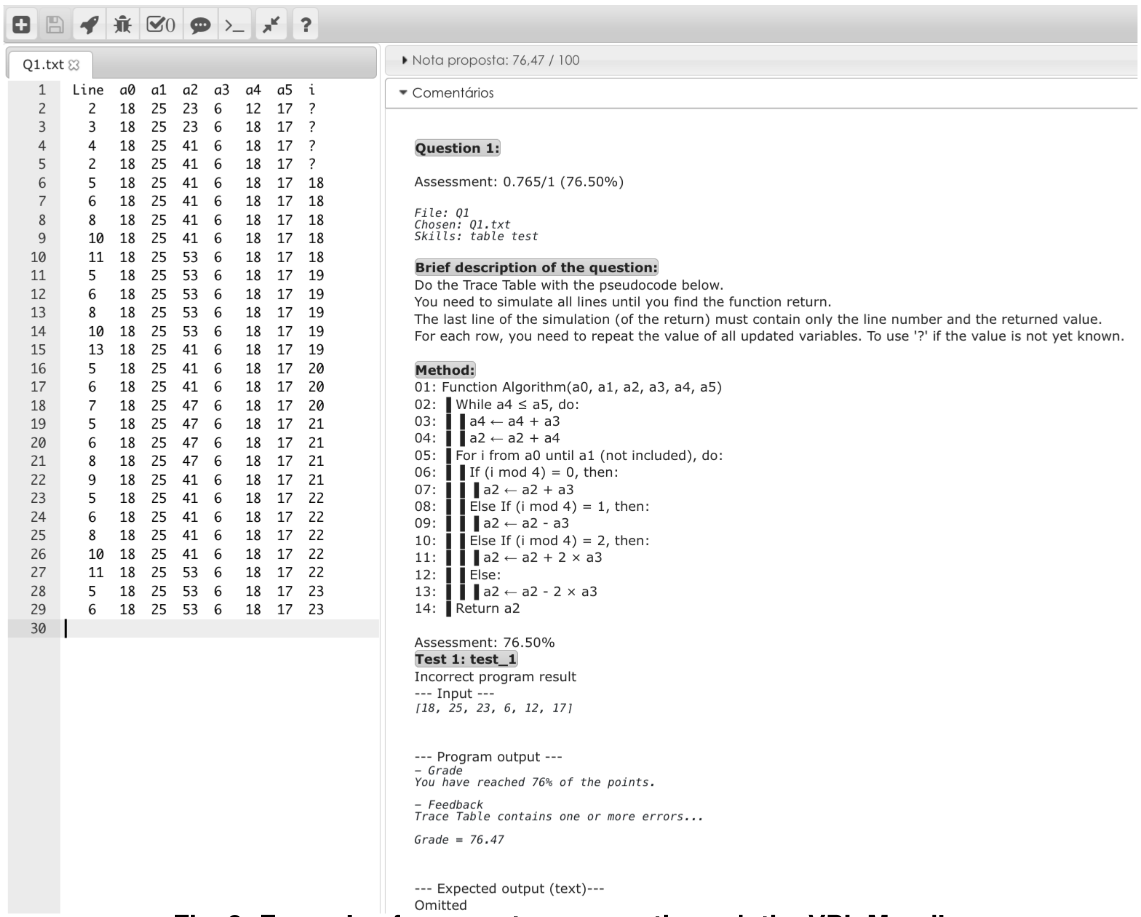
5. Resultants

Out of 309 active students, 281 completed the course by taking all assessments, including the substitute assessment and exam. 271 students completed the first assessment, and 263 completed the second assessment. The following subsection will present the quantitative results of both assessments, while the subsequent subsection will present qualitative results obtained through survey forms.

5.1. Quantitative Results

Table 2 presents the distribution of grades in the first and second evaluations, as well as the final average grade. Firstly, it can be observed that the grade distribution is similar between the first and second evaluations. This information suggests that the complexity of the exams was proportional between the evaluations. This equivalence can also suggest that the students had the same difficulty in completing the exams, not being affected by any learning curve in the evaluation process, which suggests that the evaluation method does not require prior skills.

Another important report is that about 71% of the students who completed the course achieved the maximum grade, and about 80% achieved the maximum grade in



```
Line  a0  a1  a2  a3  a4  a5  i
  2   18  25  23  6   12  17  ?
  3   18  25  23  6   18  17  ?
  4   18  25  41  6   18  17  ?
  2   18  25  41  6   18  17  ?
  5   18  25  41  6   18  17  18
  6   18  25  41  6   18  17  18
  8   18  25  41  6   18  17  18
  10  18  25  41  6   18  17  18
  11  18  25  53  6   18  17  18
  5   18  25  53  6   18  17  19
  6   18  25  53  6   18  17  19
  8   18  25  53  6   18  17  19
  10  18  25  53  6   18  17  19
  13  18  25  41  6   18  17  19
  5   18  25  41  6   18  17  20
  6   18  25  41  6   18  17  20
  7   18  25  47  6   18  17  20
  5   18  25  47  6   18  17  21
  6   18  25  47  6   18  17  21
  8   18  25  47  6   18  17  21
  9   18  25  41  6   18  17  21
  5   18  25  41  6   18  17  22
  6   18  25  41  6   18  17  22
  8   18  25  41  6   18  17  22
  10  18  25  41  6   18  17  22
  11  18  25  53  6   18  17  22
  5   18  25  53  6   18  17  23
  6   18  25  53  6   18  17  23
```

Question 1:

Assessment: 0.765/1 (76.50%)

File: Q1
Chosen: Q1.txt
Skills: table test

Brief description of the question:
Do the Trace Table with the pseudocode below.
You need to simulate all lines until you find the function return.
The last line of the simulation (of the return) must contain only the line number and the returned value.
For each row, you need to repeat the value of all updated variables. To use '?' if the value is not yet known.

Method:

```
01: Function Algorithm(a0, a1, a2, a3, a4, a5)
02:   While a4 ≤ a5, do:
03:     a4 ← a4 + a3
04:     a2 ← a2 + a4
05:   For i from a0 until a1 (not included), do:
06:     If (i mod 4) = 0, then:
07:       a2 ← a2 + a3
08:     Else If (i mod 4) = 1, then:
09:       a2 ← a2 - a3
10:     Else If (i mod 4) = 2, then:
11:       a2 ← a2 + 2 × a3
12:     Else:
13:       a2 ← a2 - 2 × a3
14:   Return a2
```

Assessment: 76.50%
Test 1: test_1
Incorrect program result
--- Input ---
[18, 25, 23, 6, 12, 17]

```
--- Program output ---
- Grade
You have reached 76% of the points.

- Feedback
Trace Table contains one or more errors...

Grade = 76.47

--- Expected output (text)---
Omitted
```

Fig. 3. Example of a correct response through the VPL Moodle.

1. Create a trace table for the code below. Consider the following input arguments: [3, 4, 2, 1, 0], [[8, 4], [8, 9], [4, 4], [7, 6], [7, 3]]. You must simulate every line of the code until you reach the function's return statement. The last line of the simulation (the return) should contain only the line number and the value returned. For each line, you need to repeat the value of all updated variables. Use '?' if the value is not yet known. For variables of the list type, use the notation '[1, 2, 3]'. Commas are optional. Do not use the codes () {} and to represent whether the value was read or written. Just leave the values.

Python code:

```python
def Algoritmo(v, m):
    a = [ 0, 0, 0, 0, 0 ]
    for i in range(0, len(m)):
        for j in range(0, len(m[i])):
            if a[v[i]] > m[i][j]:
                a[v[i]] -= m[i][j]
            else:
                a[v[i]] += m[i][j]
    return a
```

Continue with your simulation:

```
 Line      a          i       j
  2    [0, 0, 0, 0, 0]   ?       ?
  3    [0, 0, 0, 0, 0]   0       ?
```

Fig. 4. Example of question from the second exam, sent by email.

the course, grade A. Historically, the pass rate for this course has been below 70% on average [Zampirolli et al. 2021a]. This fact suggests four hypotheses: (1) the Trace Table evaluation style or the algorithm was easy; (2) there was a high rate of plagiarism; (3) location and time in answering; and (4) failures in the automatic correction system by providing important hints.

The first hypothesis is not likely, as the same question style has been used in previous offerings. The second hypothesis regarding plagiarism is also unlikely, considering that each student has a unique set of questions and answers. The third hypothesis is quite likely. The rules of the University for remote evaluation require it to be asynchronous and last for 72 hours. This exaggerated time may have allowed the student to study and try different answers until they achieved perfection.

The last hypothesis occurs due to the unlimited feedback of the evaluation. The student could write one line at a time and evaluate it. When the score was increased, it meant that the line was correct, and the student started the next line. This, in theory, allows for trial and error resolution. After the first evaluation, the number of evaluations performed by the students was evaluated, and about 8 students (3%) evaluated more than 100 times, more than 80 (30%) students evaluated more than 50 times, and 187 students (69%) evaluated more than 30 times. This data suggests that unlimited evaluation can facilitate the full resolution of the question, but not through trial and error.

An alternative to avoid this type of assistance would be to limit the number of evaluation attempts to, for example, 20. This limitation was not imposed in the second evaluation to avoid significant changes compared to the first evaluation in order not to harm the student. However, future work can place an evaluation limiter or decrement the grade according to the number of evaluations. The submission deadline can also be reviewed, as 72 hours for taking a test allows the student to evaluate multiple times until finding the correct answer.

Tab. 2. Student performance table.

Grade	First Evaluation	Second Evaluation	Final Average
100%	$226(83,4\%)$	$232(88,2\%)$	$202(71,9\%)$
$[90\%, 100\%[$	$15(5,5\%)$	$18(6,8\%)$	$26(9,3\%)$
$[70\%, 90\%[$	$8(3,0\%)$	$3(1,1\%)$	$16(5,6\%)$
$[50\%, 70\%[$	$10(3,7\%)$	$2(0,8\%)$	$11(3,9\%)$
$[0\%, 50\%[$	$12(4,4\%)$	$8(3,1\%)$	$26(9,3\%)$
Total	$271(100\%)$	$263(100\%)$	$281(100\%)$

5.2. Qualitative Results

Two forms were sent after the first and second evaluations, respectively. About 23% of the students responded to the first evaluation. The form for the initial evaluation is presented in Table 3 and uses a Likert scale ranging from 1 to 5 to indicate levels of agreement, where 1 represents Totally Disagree, 2 represents Disagree, 3 represents Neutral, 4 represents Agree, and 5 represents Totally Agree. An explanatory video on how to use the evaluation tool was provided prior to the first evaluation. The acceptance of the evaluation method was 86%, and the vast majority (81%) said that plagiarism had been reduced. Regarding the response process, 80% of the students liked the feedback provided during the evaluations, and 74% of the students said that the text response in table format was appropriate.

The second questionnaire had only 11 participants, around 4% of the students. One explanation for the low participation in the second evaluation is that it was requested after the end of the classes. Table 4 shows the students' responses. Despite the low participation of the students, the values were equivalent to those of the first evaluation, with around 80% of acceptance.

In one of the questions, "Easy to understand the question", although the approval rate was 82%, only 27% answered "Totally agree". This questionnaire had a profile of 36.4% of students with the maximum score, 45.5% with intermediate scores, and 18.2% of failures. As the students with the maximum score were 71%, the sampling of the

Tab. 3. Questionnaire of 64 students (23%) about Evaluation 1.

Question	1	2	3	4	5
Moodle+VPL use in EVALUATIONS with automatic correction and immediate feedback is very interesting	3(5%)	0(0%)	6(9%)	10(16%)	45(70%)
The VIDEO explaining the EVALUATION is very good, which improved my understanding	0(0%)	0(0%)	0(0%)	10(16%)	54(84%)
The INDIVIDUAL EVALUATION with RANDOM VERSIONS helps to reduce plagiarism, thus helping in learning	1(2%)	2(3%)	9(14%)	18(28%)	34(53%)
The automatic FEEDBACK of the evaluation helped me understand the problems, although it did not point out exactly where the errors were	3(5%)	8(12%)	2(3%)	9(14%)	42(66%)
The RESPONSE FORMAT (table in text file) was adequate for the proposed question	3(5%)	2(3%)	12(19%)	14(22%)	33(52%)
I liked the EVALUATION, I would like a similar one for EVALUATION 2	2(3%)	5(8%)	6(9%)	12(19%)	39(61%)

Tab. 4. Questionnaire of 11 students (4%) about Evaluation 2.

Question	1	2	3	4	5
Consistent with the subject	0(0%)	0(0%)	2(18%)	2(18%)	7(64%)
Efficient to Evaluate Knowledge	1(9%)	0(0%)	2(18%)	3(27%)	5(45%)
Easy to Understand the Question	0(0%)	2(18%)	0(0%)	6(55%)	3(27%)
Adequate Response Style	0(0%)	1(9%)	1(9%)	4(36%)	5(45%)
Appropriate Difficulty Level	0(0%)	0(0%)	3(27%)	4(36%)	4(36%)
Automatic Correction and Feedback helpful for Learning	0(0%)	0(0%)	2(18%)	4(36%)	5(45%)

responses is from students who had more difficulties and, consequently, there may be a divergence in relation to the overall opinion.

5.3. Threats to validity

There are some threats to validity of our study. Despite the method being applied to many students, it would be necessary to conduct new experiments with both test and control groups to ensure improvements in the first group. Furthermore, the method was applied in a remote context and should also be tested in in-person settings with limited time. It would also be essential to verify whether the students can generate code to present the results from the Trace Table.

6. Conclusion and Future Work

This work presented an innovative method for evaluating students' knowledge in pseudocode through the Trace Table model, demonstrating the feasibility and acceptance of an automatic, asynchronous evaluation in a CS1 course. The use of VPL and MCTest enabled parametric and remotely administered evaluations that provided immediate feedback to students. Two assessments were applied with simple questions involving conditional and repetition structures and vectors and matrices. The results indicated a high approval rate, with over 80% of students achieving an A.

This model can be expanded to upper-level programming courses. The types of questions and answers can be diversified to include code snippets and programming problems in addition to pseudocode. Although the method presented in this article was applied to exams containing only one question, it has already been included in the MCTest version, available on GitHub, and can be used in exams that contain Trace Table questions in languages other than pseudocode (such as Python, Java, etc.) and with other types of questions. This integration has not yet been validated in classes and will be carried out in future work.

Additional studies should be conducted to investigate the impact of this assessment method on student learning and outcomes. Furthermore, the method presented in this article has great potential for use in Massive Open Online Courses (MOOCs) and especially in traditional face-to-face courses with thousands of students.

References

[Brusilovsky et al. 2014] Brusilovsky, P., Edwards, S., Kumar, A., Malmi, L., Benotti, L., Buck, D., Ihantola, P., Prince, R., Sirkiä, T., Sosnovsky, S., Urquiza, J., Vihavainen, A., and Wollowski, M. (2014). Increasing adoption of smart learning content for computer science education. In *ITiCSE-WGR*, pages 31–57.

[Combéfis 2022] Combéfis, S. (2022). Automated code assessment for education: review, classification and perspectives on techniques and tools. *Software*, 1(1):3–30.

[Mendoza and Zavala 2018] Mendoza, B. and Zavala, L. (2018). An intervention strategy to hone students' code under-standing skills. *Journal of Computing Sciences in Colleges*, 33(3):105–114.

[Paiva et al. 2022] Paiva, J. C., Leal, J. P., and Figueira, A. (2022). Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education*, 22(3):1–40.

[Risha and Brusilovsky 2020] Risha, Z. and Brusilovsky, P. (2020). Making it smart: Converting static code into an interactive trace table. In *Proc. of Sixth SPLICE Workshop*.

[Rodríguez del Pino et al. 2010] Rodríguez del Pino, J. C., Rubio Royo, E., and Hernández Figueroa, Z. J. (2010). VPL: laboratorio virtual de programación para moodle. In *Jornadas de Enseñanza Universitaria de la Informática*, pages 429–435.

[Zampirolli et al. 2021a] Zampirolli, F. A., Borovina Josko, J. M., Venero, M. L., Kobayashi, G., Fraga, F. J., Goya, D., and Savegnago, H. R. (2021a). An experience of automated assessment in a large-scale introduction programming course. *Computer Applications in Engineering Education*, 29(5):1284–1299.

[Zampirolli et al. 2019] Zampirolli, F. A., Teubl, F., and Batista, V. (2019). Online generator and corrector of parametric questions in hard copy useful for the elaboration of thousands of individualized exams. In *CSEDU*, pages 352–359.

[Zampirolli et al. 2021b] Zampirolli, F. A., Teubl, F., Kobayashi, G., Neves, R., Rozante, L., and Batista, V. (2021b). Introductory computer science course by adopting many programming languages. In *SBIE*, pages 1118–1127, Porto Alegre, RS, Brasil. SBC.

[Zavala and Mendoza 2017] Zavala, L. and Mendoza, B. (2017). Precursor skills to writing code. *Journal of Computing Science in Colleges*, 32(3):149–156.