Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Evaluating Feedback Tools in Introductory Programming Classes

Ruan Victor Bertoldo Reis de Amorim

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Educação em Ciência da Computação

Gustavo Soares (Orientador)

Melina Mongiovi (Orientadora)

Campina Grande, Paraíba, Brasil

A524e Amorim, Ruan Victor Bertoldo Reis de.
Evaluating feedback tools in introductory programming classes / Ruan Victor Bertoldo Reis de Amorim. – Campina Grande, 2019.
45 f. : il. color.

Dissertação (Mestrado em Ciência Computação) – Universidade Federal de Campina Grande, Centro Centro de Engenharia Elétrica e Informática, 2019.
"Orientação: Prof. Dr. Gustavo Araújo Soares, Profa. Dra. Melina Mongiovi Cunha Lima Sabino".
Referências.

1. Computação – Estudo e Ensino. 2. Programação de Computadores. 3. Ferramentas de Feedback. 4. Educação. I. Soares, Gustavo Araújo. II. Sabino, Melina Mongiovi Cunha Lima. III. Título.

CDU 004.42(07)(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

"EVALUATING FEEDBACK TOOLS IN INTRODUCTORY PROGRAMMING CLASSES"

RUAN VICTOR BERTOLDO REIS DE AMORIM

DISSERTAÇÃO APROVADA EM 19/08/2019

CAMPINA GRANDE - PB

Resumo

Aprender a programar é um desafio enfrentado pelos alunos na maioria dos cursos de introdução à programação. Por este motivo, diversas ferramentas têm sido propostas com o propósito de ajudar os alunos a superar dificuldades conceituais durante o seu aprendizado. Existem ferramentas que utilizam algoritmos de agrupamento e técnicas de reparo de programas para gerar feedback personalizado para os alunos. Em contraste, alguns professores optam por apresentar aos alunos alguma ferramenta de visualização de programas com o intuito de ajudá-los a entender a execução dinâmica de um código-fonte. Estas ferramentas são utilizadas para auxiliar alunos na obtenção de soluções para problemas de programação. No entanto, devido à limitações nas avaliações, ainda não está claro o quão efetivo é o feedback fornecido por elas. Neste estudo, analisamos a eficácia de duas ferramentas, uma de geração de dicas personalizadas e outra de visualização de programas. Para tanto, realizamos um estudo de usuários em que os alunos, auxiliados por essas ferramentas, implementaram soluções para três problemas de programação. Nossos resultados mostram que dicas personalizadas podem reduzir significativamente o esforço do aluno para obter soluções corretas. Além disso, dicas personalizadas podem fornecer aos alunos uma compreensão da solução de problemas semelhante ao uso de casos de teste. Em contrapartida, os alunos que usaram a ferramenta de visualização de programas obtiveram desempenho inferior comparado ao uso de outras abordagens.

Abstract

Learning to program is a challenge faced by students in most introductory programming courses. Recently, several tools have been proposed in order to provide guidance and help students overcome conceptual difficulties in programming education. Some tools leverage clustering algorithms and program repair techniques to automatically generate personalized hints for students' incorrect programs. In contrast, some teachers choose to present students with program visualization tools to help them understand the dynamic execution of a source code. These tools are used to help students get correct solutions for programming assignments. However, due to limitations in assessments, it is still unclear how effective the feedback provided by these tools is. In this study, we analyzed the effectiveness of a tool for generating personalized hints and a tool for visualizing programs. To do so, we conducted a user study in which students, assisted by these tools, implemented solutions for three programming problems. Our results show that personalized hints can significantly reduce student's effort to get correct solutions. In addition, personalized hints can provide students with an understanding of problem solving similar to when using test cases. On the other hand, students who used the program visualization tool got lower performance than using other approaches.

Agradecimentos

Eu agradeço aos meus orientadores, Gustavo Soares e Melina Mongiovi, por toda orientação durante os dois anos de mestrado na UFCG. Esta dissertação é fruto de um belo trabalho em equipe. Eu agradeço aos meus pais, Adriana Bertoldo e Antonio Amorim, que sempre me deram todo o suporte necessário para alcançar os meus sonhos. Sem eles ao meu lado nada na minha vida seria possível. Também agradeço às minhas irmãs, Luana Bertoldo e Anita Amorim, pela amizade e compreensão. Em especial, agradeço à Ana Luiza Martins pela paciência, carinho e companheirismo. Meus dias são mais leves e felizes ao lado dela. Eu agradeço aos professores e funcionários do DSC/COPIN pelo comprometimento e esclarecimentos prestados ao longo desses anos dentro da UFCG. Finalmente, agradeço a CAPES pelo suporte dado ao meu trabalho.

Contents

List of Figures

List of Tables

List of Source Codes

Chapter 1

Introduction

Learning to program is a challenge faced by students in most introductory programming courses [22]. In online and face-to-face classroom, students need to put the acquired knowledge into practice through practical programming assignments. To assist in these activities, teachers need to provide guidance and assistance, especially to novice learners who are getting their first programming experiences and need to overcome conceptual difficulties [4].

Feedback from teachers can help students get unstuck and correct their misconceptions [5], [13]. However, personalized attention does not scale easily, especially in massive programming classrooms [6], [9]. One of the most common practices used by teachers to provide feedback at scale is to present the student with a test-case suite. In this way, students can run their programs against test cases and receive reports from failing tests. However, it can be difficult for a novice programmer to map failed test results back to a specific fault in their code.

Recently, several tools have been proposed to support programming education [11], [37], [34], [14], [28], [18], [7], [24], [12]. These tools use different approaches to generate, scale and personalize feedback to help teachers and students in programming classes. For example, CLARA [11] can automatically repair incorrect programs, indicate the location of bugs (e.g., line number), and provide an exactly textual description of required changes. The approach used by CLARA consists of clustering the existing solutions for a given assignment; selecting a target program from each cluster; and executing a trace-based repair procedure to repair new incorrect attempts. The PYTHON TUTOR [12] allows users to step forwards and backwards through execution to visualize the run-time state of a program's data structures.

The approach used by this tool is to analyze an input program under the supervision of the standard Python debugger module (bdb), which stops execution after every executed line and records the program's run-time state. Using the PYTHON TUTOR, students can debug their programs and, as a result, they can fix bugs and get correct solutions for programming assignments. These types of tools are often used in programming education and can be useful in reducing the teacher's effort to provide feedback.

1.1 Problem

Feedback tools are widely used in introductory programming classes to assist students in their assignments [10]. However, it is still unclear how effective these tools are, especially when the user is a novice programmer. The reason for the lack of clarity on this subject is due to the limitations in evaluations of these tools. The most common limitations found in papers are related to: (1) the lack of user studies, especially with beginners, e.g., [24], [31]; (2) fail to get insight into learning improvement or skill acquisition, e.g., [11], [34]; (3) the lack of comparative studies with other existing tools, e.g., [14], [20]; and (4) focus only on evaluating the tool's performance in generating hints, but does not evaluate its usefulness, e.g., [7], [28]. Therefore, it is necessary to investigate to what extent the feedback approaches provided by these tools can be effective.

1.2 Solution

In this study, we evaluated the effectiveness of CLARA and PYTHON TUTOR in assisting novice programmers in problem solving. Our goal is to analyze whether using these tools students can solve programming assignments better than when using only test-case suites. We selected CLARA for our evaluation because it is a state-of-the-art tool for automatic hints generation. This tool can generate a large number of repairs without any manual intervention, can perform complicated repairs, can be used in an interactive teaching setting, and generates good quality repairs in a large percentage of cases. On the other hand, we selected PYTHON TUTOR because it is widely used in programming classes to visualize program execution. It is a free, open-source, web-based tool following in the long tradition of program visualizations for Computer Science education. Moreover, it is the only Python program visualization tool that runs within a web browser without any required software or plugin installation.

In our evaluation, we recruited 42 undergraduate students and asked them to implement Python solutions for three classic problems. For each problem, students were able to use CLARA or PYTHON TUTOR, and a test-case suite to assist them in the resolution process. Subsequently, in order to evaluate the impact of the tools on the student's understanding of how to solve the problems, we proposed a post-test in which students should review four solutions for a specific problem and indicate whether each solution is correct or not. Specifically, we analyzed the effectiveness of these tools with respect to three aspects: (i) whether the tool leads students to faster results when compared to test cases alone; (ii) what is the impact of the tool on the student's understanding of how to solve the problem; (iii) whether the tool is more useful than using test cases alone.

1.3 Evaluation

Our results show that, when considering getting correct solutions faster, CLARA can significantly reduce the student's effort, in number of attempts, compared to PYTHON TUTOR and TEST CASES. In the post-test results, we did not find a significant difference comparing the performance of students who used CLARA (score of 2.4 on average) with those who used only test cases (score of 2.3 on average). This may mean that students who used these tools understood problem solving at the same level. However, students who used PYTHON TUTOR got lower post-test performance than using CLARA or TEST CASES (score of 1.8 on average). This highlights a difficulty for novice programmers in performing debugging activities. Finally, students scored CLARA (score of 5.9 on average) as more useful than test cases (score of 5.3 on average) to fix bugs in their programs. They mentioned that finding bugs using only test cases is difficult, but using CLARA, they can figure out where the bugs are and immediately reflect on the hints provided.

1.4 Contribution

In summary, this master thesis proposes an evaluation of feedback tools in introductory programming classes. We conducted user studies with novice programming students to evaluate the effectiveness of a tool for generating personalized hints and a program visualization tool. There is a lack of studies evaluating and comparing different feedback approaches. This work contributes with quantitative and qualitative results of a controlled experiment with 42 undergraduate students where we compare CLARA, PYTHON TUTOR, and TEST CASES [26]. Our results show that more specific feedback can benefit beginner students' programming learning. However, TEST CASES should not be excluded. On the other hand, teachers should be careful when providing beginning students with debugging tools such as PYTHON TUTOR. These results contribute to studies on the effect of feedback tools in programming education.

1.5 Organization

The following chapters are organized as follows. In Chapter 2, we present the background needed to better understand this work. We describe the concepts related to formative feedback and present the feedback tools covered in this study. Chapter 3 describes in detail the experiment conducted with students from introductory programming classes. We present the methodology used, the results obtained, discussions and threats to validity. Finally, we present the works related to ours (Chapter 4), the concluding remarks (Chapter 5), and the appendices (A and B).

Chapter 2

Background

In this chapter, we present an overview of the concepts needed to better understand this work. Section 2.1 presents the definition of formative feedback and how it is classified. Next, Section 2.2 describes in more detail the feedback tools covered in our study.

2.1 Formative Feedback

Shute et al. [30] define and classify feedback through an extensive literature review on this subject. In their work, formative feedback is defined as information communicated to the learner that is intended to modify his or her thinking or behavior for the purpose of improving learning. In the educational context, feedback is considered crucial for improving the acquisition of knowledge and skills. The following describes the types of feedback and when they should be used.

Directive and Facilitative Feedback

There are two main functions of feedback: directive and facilitative. Directive feedback is the one which tells the student what needs to be fixed or revised. Such feedback tends to be more specific compared to facilitative feedback, which provides comments and suggestions to help guide students in their own revision and conceptualization. Novices or struggling students need support and explicit guidance during the learning process. For these students, more directive feedback is recommended. However, for high-achieving learners, facilitative feedback is more appropriate. High-achieving or more motivated students benefit from feedback that challenges them, such as hints, cues, and prompts.

Specific Feedback

Feedback specificity is defined as the level of information presented in feedback messages. In other words, specific (or elaborated) feedback provides information about particular responses or behaviors beyond their accuracy and tends to be more directive than facilitative. Several researchers have reported that feedback is significantly more effective when it provides details of how to improve the answer rather than just indicating whether the student's work is correct or not. For students with low learning orientation, more specific feedback is recommended. If students are oriented more toward performance (trying to please others) and less toward learning (trying to achieve an academic goal), specific and goal-oriented feedback should be provided.

Verification and Elaboration Feedback

Effective feedback provides the learner with two types of information: verification and elaboration. Verification is defined as the simple judgment of whether an answer is correct, and elaboration is the informational aspect of the message, providing relevant cues to guide the learner toward a correct answer. Researchers appear to be converging toward the view that effective feedback should include elements of both verification and elaboration. High-achieving students learn more efficiently if permitted to proceed at their own pace. For these students, verification feedback may be sufficient. This type of feedback provides the level of information most helpful in this endeavor. However, for low-achieving learners, the correct response and some kind of elaboration feedback is recommended. Low-achieving students should be given a concrete and directive form of feedback support.

Scaffolding Feedback

Scaffolding enables learners to do more advanced activities and to engage in more advanced thinking and problem solving than they could without such help. Eventually, high-level functions are gradually turned over to the students as the teacher (or computer system) removes the scaffolding and fades away. Scaffolding feedback should be considered for low-achieving learners. Providing early support and structure for low-achieving students (or those with low self-efficacy) can improve learning and performance.

Delayed and Immediate Feedback

The timing of feedback literature concerns whether feedback should be delivered immediately or delayed. *Immediately* may be defined as right after a student has responded to an item or problem or, in the case of summative feedback, right after a quiz or test has been completed. *Delayed* is usually defined relative to immediate, and such feedback may occur minutes, hours, weeks, or longer after the completion of some task or test. High-achieving students may construe a moderate or difficult task as relatively easy and hence benefit by delayed feedback. However, for low-achieving learners, more immediate feedback is recommended. These students need the support of immediate feedback in learning new tasks they may find difficult.

2.2 Feedback Tools

The following describes the feedback tools covered in our study. Section 2.2.1 describes how test cases suites are used as feedback. Next, we present CLARA, an automated feedback generation tool (Section 2.2.2), and PYTHON TUTOR, an interactive visual debugging tool (Section 2.2.3). Table 2.1 summarizes the tools evaluated in this study and their feedback characteristics.

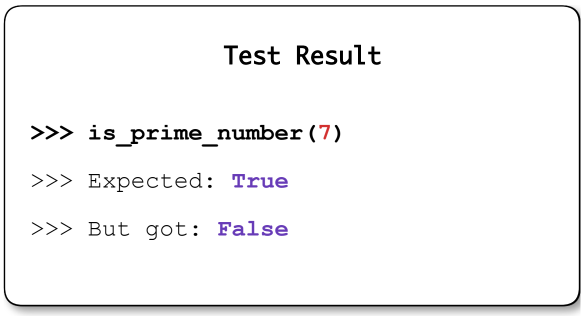
Table 2.1: Summary of tools by type of feedback provided by them.

TOOLS AND THEIR FEEDBACK CHARACTERISTICS				
	FUNCTION	**SPECIFICITY**	**INFORMATION**	**TIMING**
TEST CASES	facilitative	low	verification	immediate
CLARA	directive	high	elaboration	immediate
PYTHON TUTOR	facilitative	low	elaboration	immediate

2.2.1 Test Case Suites

It is one of the most common practices used by teachers to provide feedback in programming classes. Generally, the teacher provides a set of test cases that describes the expected behavior of student programs for a given assignment. The student program is run on a set of test cases and the failing test cases are reported back to the student. Thus, students can check whether their programs are returning the expected results. However, the feedback of failing test cases is not ideal; especially for beginner programmers who find it difficult to map the failing test cases to faults in their code. This is reflected by the number of students who post their submissions on the discussion boards to seek help from instructors and other students after struggling for hours to correct the mistakes themselves [31].

The feedback generated by test cases consists of indicating to which inputs a given program does not return the expected result. For example, consider a student program that should indicate whether a number is prime or not, however, the program does not return the expected result when the input value equals seven. Feedback based on test cases indicates which input value caused the test failure, as well as the result obtained and the expected result. Figure 2.1 presents an example of feedback based on the failure of a test case. This example shows that the *is_prime_number* function did not return the expected result when the input value was equal to seven. In addition, it indicates that the expected result was *True*, but the result obtained was *False*.



```
Test Result

>>> is_prime_number(7)
>>> Expected: True
>>> But got: False
```

Figure 2.1: Example of a test case result.

Feedback based on TEST CASES is classified as **facilitative**, **non-specific**, **verification**, and **immediate**.

2.2.2 CLARA

It is a fully automated program repair tool for introductory programming assignments. The key idea of CLARA's approach is to use the *wisdom of the crowd*: It uses the existing correct student solutions to repair the new incorrect student attempts. This tool explores the fact that MOOC courses already have tens of thousands of existing student attempts; this was already noticed by Drummond et al. [8].

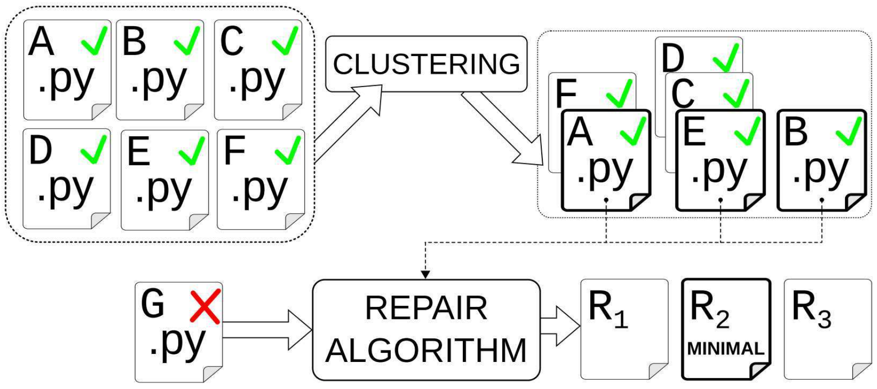


Figure 2.2: High-level overview of CLARA's approach.

Figure 2.2 gives a high-level overview of CLARA's approach: (A) For a given programming assignment, it automatically clusters the correct student solutions (**A-F** in the figure), based on a notion of *dynamic equivalence*; (B) Given an incorrect student attempt (**G** in the figure) CLARA runs a repair algorithm against all clusters, and then selects a minimal repair (**R2** in the figure) from the generated repair candidates (**R1-R3** in the figure). The repair algorithm uses expressions from multiple correct solutions to generate a repair.

Intuitively, the clustering algorithm groups together similar correct solutions. The repair algorithm can be seen as a generalization of the clustering approach of correct solutions to incorrect attempts. The key motivation behind this approach is as follows: to help the student, with an incorrect attempt, CLARA's approach finds the set of most similar correct solutions, written by other students, and generates the smallest modifications that get the student to a correct solution.

As a result, CLARA can synthesize repairs to the student's incorrect programs, indicate the location of bugs (e.g., line number), and provide an exact textual description of the required changes. For example, consider the Source Code 2.1 as an incorrect attempt for the

Sum of Squares problem:

Source Code 2.1: Example of an incorrect implementation for the *Sum of Squares* problem.

```
1 def sum_of_squares(n):
2   total = 1
3   for i in range(n):
4     total = total + i*2
5   return total
```

In this example, CLARA is able to identify three bugs in the given program as input. The first bug can be found in the assignment of the variable *total* at line 2. The second bug is in the parameters of the *range* iterative expression at line 3. Finally, the third bug refers to updating the *total* variable at line 4. As feedback, CLARA synthesizes the following repairs that describe exactly what changes are needed:

- *In assignment at line 2, change* ***total = 1*** *to* ***total = 0****.*
- *In iterator expression at line 3, change* ***range(n)*** *to* ***range(1, n+1)****.*
- *In assignment at line 4, change* ***total = total + i*2*** *to* ***total = total + i**2****.*

In evaluation conducted by Gulwani et al. [11], CLARA was able to repair 97% of student attempts, in 3.2s on average (on 12,973 correct and 4,293 incorrect student attempts); the authors studied the quality of the generated repairs by manual inspection and found that 81% of the generated repairs are of *good-quality* and the size of the generated repair matches the size of the required changes to the student's program.

Feedback provided by CLARA is classified as **directive**, **specific**, **elaboration** and **immediate**.

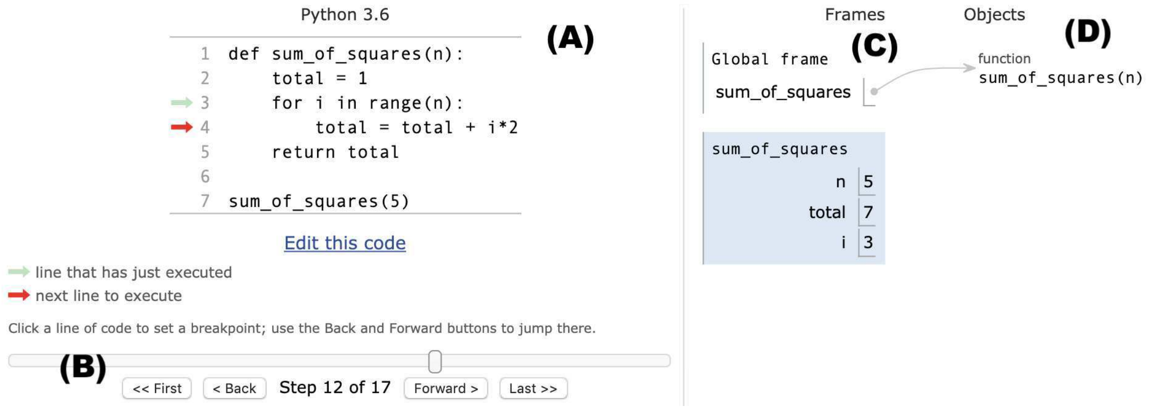
2.2.3 Online Python Tutor

It is a web-based program visualization tool for Python. This tool allows students to step forwards and backwards through execution to view the runtime state of a program's data structures. Students can use these features to debug their programs. As a consequence, they can fix bugs, and get correct solutions to programming assignments.

The PYTHON TUTOR backend takes the source code of a Python program as input and produces an *execution trace* as output. The backend executes the input program under supervision of the standard Python debugger module (bdb), which stops execution after every executed line and records the program's runtime state. The trace is an ordered list of execution points, where each point contains the state right before a line of code is about to execute, including: (1) The line number of the line that is about to execute; (2) the instruction type (ordinary single step, exception, function call, or function return); (3) a map of global variable names to their current values at this execution point; (4) an ordered list of stack frames, where each frame contains a map of local variable names to current values; (5) the current state of the heap; and (6) the program's output up to this execution point.

The following describes each labeled component of the PYTHON TUTOR interface (shown in Figure 2.3) and its respective functionality:

- **(A)** The source code display shows the program that is being visualized. A red arrow in the left margin points to the next line to be executed (line 4 in this example). A light green arrow points to the line that has just executed, which helps users track non-contiguous control flow (e.g., function calls).
- **(B)** A slider bar and text indicate the current execution point being visualized (in this example, step 12 of 17). Each point represents a single executed line. The user can click on or drag the mouse over the slider bar to jump to a particular point or use the VCR-style navigation buttons to step forwards and backwards over executed lines.
- **(C)** The frames pane shows global variables and stack frames at the current execution point, with the stack growing downward. Each frame shows the function name and a list of local variables. Each variable's value is an arrow that points to a heap object.
- **(D)** The objects pane shows visual representations of Python objects and their pointer references to one another. Online Python Tutor renders all compound data types necessary for teaching CS1, including list, tuple, set, dict, class, object instance, and closures.



```
def sum_of_squares(n):
    total = 1
    for i in range(n):
        total = total + i*2
    return total

sum_of_squares(5)
```

Figure 2.3: PYTHON TUTOR Graphical User Interface (GUI).

Feedback provided by PYTHON TUTOR is classified as **facilitative**, **non-specific**, **elaboration** and **immediate**.

Chapter 3

Evaluating Feedback Tools in Introductory Programming Classes

In this chapter, we explain in detail our study design. First, we describe our goals (Section 3.1), the background of the participants (Section 3.2), and the method used in our experiments (Section 3.3). Then, Sections 3.4, 3.5 and 3.6 respectively describe the programming problems covered, the post-test and post-survey applied to participants. Finally, we present our results (Section 3.7), a discussion about them (Section 3.8), and some threats to validity (Section 3.9).

3.1 Study Definition

The goal of our experiment is to evaluate the effectiveness of CLARA and PYTHON TUTOR in assisting novice programmers in problem solving. We are interested in analyzing whether using these tools students can solve programming assignments better than when using only TEST CASES. For this purpose, we address the following research questions:

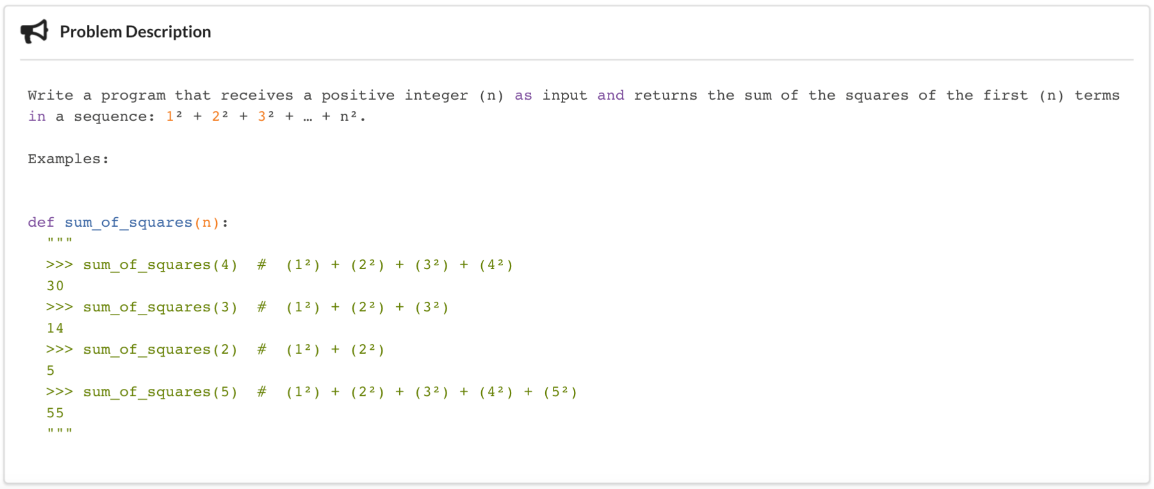
- **RQ1:** *Do students using* CLARA *or* PYTHON TUTOR *solve problems faster than using only* TEST CASES*?* We are interested in analyzing whether CLARA or PYTHON TUTOR can reduce the number of attempts needed to get a correct solution. This can be an indicator that the student is actually being helped by the tool.

- **RQ2:** *What is the impact of using the tools in terms of understanding problem solving when compared to using* TEST CASES *alone?* We are interested in analyzing whether CLARA or PYTHON TUTOR can impact the student's understanding of how to solve the problem. It is important to investigate whether the tool is harming or benefiting student learning.

- **RQ3:** *Do students find* CLARA *or* PYTHON TUTOR *more useful to fix bugs than* TEST CASES*?* Given the purposes of each tool, we are interested in finding out which approach students find most useful in bug fixing.

3.2 Participants

We recruited 42 undergraduate students from UFCG and IFPB introductory programming classes. All participants are from Computer Science or Engineering courses and are getting their first programming experiences with the Python language. They have knowledge in assigning variables, mathematical and boolean expressions, conditional structures (if / elif / else) and iteration using loops (for / while). To participate in this study students signed a informed consent form (Appendix A: Informed Consent Form). This research was also approved by the UFCG research ethics committee (Appendix B: Consubstantiated Opinion of the Research Ethics Committee).


Problem Description

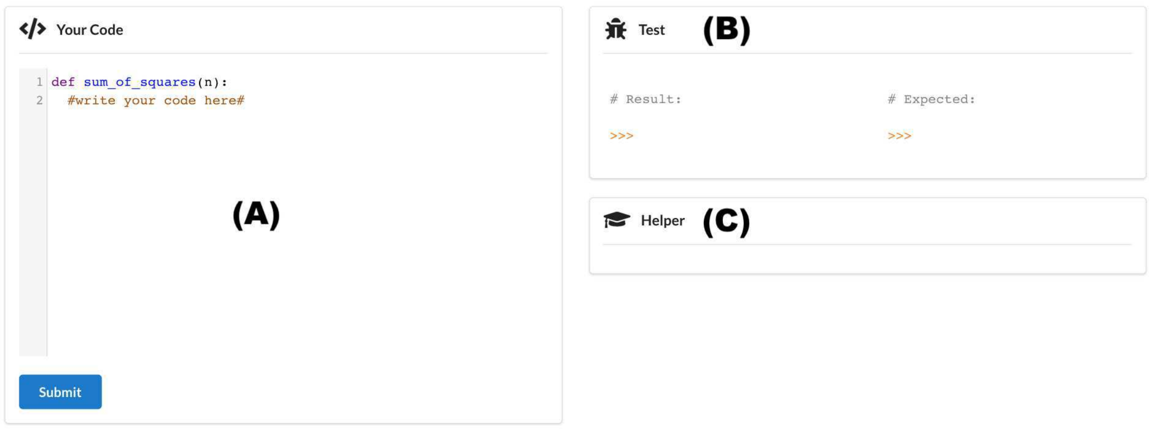
```
Write a program that receives a positive integer (n) as input and returns the sum of the squares of the first (n) terms
in a sequence: 1² + 2² + 3² + … + n².

Examples:


def sum_of_squares(n):
  """
  >>> sum_of_squares(4)  #  (1²) + (2²) + (3²) + (4²)
  30
  >>> sum_of_squares(3)  #  (1²) + (2²) + (3²)
  14
  >>> sum_of_squares(2)  #  (1²) + (2²)
  5
  >>> sum_of_squares(5)  #  (1²) + (2²) + (3²) + (4²) + (5²)
  55
  """
```

Figure 3.1: Problem description panel in our integration platform.

In order to enable interaction with the tools, we developed an integration platform where participants can write their programs and get feedback for incorrect attempts. Our platform includes a panel to describe the programming assignment (Figure 3.1), a text box to write the program code (Figure 3.2 - A), a section to show test case results (Figure 3.2 - B), and a section to present the feedback provided by the tools (Figure 3.2 - C).



```
def sum_of_squares(n):
  #write your code here#
```

```
# Result:          # Expected:
>>>                >>>
```

Figure 3.2: User interaction panels in our integration platform.

3.3 Method

At the start of each study session, we gave each participant an 8-minute video tutorial on TEST CASES, CLARA, and PYTHON TUTOR. The tutorial describes the use of each tool and test cases, and presents a concrete example of solving a factorial problem. Students were able to follow the tutorial video and also implementing their own factorial solutions. This assignment was not considered in our analysis. We expected that after the tutorial, students were able to understand and interpret the feedback provided by the tools and test cases.

Next, we asked students to implement solutions for three programming problems: *Sum of Squares*, *Prime Numbers*, and *Fibonacci*. We evaluated only three programming problems because we had a time limitation in the study sections. Participants were able to choose the order in which problems would be solved, however, we recommend that they begin with the problem that they find easiest. To provide better leveling of the students, we do not allow the implementation of recursive algorithms on the problem solutions. For each problem, participants were asked to implement a solution using one of the following conditions:

- ***Condition 1*** - They could use only the TEST CASES as assistant;
- ***Condition 2*** - They could use CLARA and TEST CASES as assistants;
- ***Condition 3*** - They could use PYTHON TUTOR and TEST CASES as assistants.

All conditions were randomly assigned to the problems. In addition, the same condition can not be attributed to different problems of the same participant.

We presented the participants with a description of each problem and asked them to solve everything within class time (two hours). Whenever the participant submits an incorrect attempt, the assigned tool will provide some feedback to help with the solution. Participants were able to submit as many times as needed until a correct solution was reached. We also did not allow them to run their programs on other platforms beyond our integration platform. To verify that a solution is correct, our platform runs the code against a test suite related to the problem.

Once the participants got a correct solution, they should do a post-test related to the problem solved. The post-test consists of four solutions to the same problem. The participant needs to review the solutions and indicate whether each solution is correct or not. Finally, we conducted a post-survey where participants could rate which tools they found most useful for fixing bugs.

3.4 Programming Problems

The programming problems used in this study were selected from exercise repositories of introductory programming courses. We chose to select problems that are common in programming assignment lists that have submission data available (correct and incorrect programs). These problems explore programming concepts such as variable assignment, mathematical expressions, conditional commands, and loops. The following describes each programming problem proposed to participants in our implementation challenge.

Sum of Squares

Write a program that receives a positive integer n as input and returns the sum of the squares of the first n terms in a sequence: $1^2 + 2^2 + 3^2 + ... + n^2$.

Examples:

- *sum_of_squares(1)* = $1^2 = 1$
- *sum_of_squares(3)* = $1^2 + 2^2 + 3^2 = 14$
- *sum_of_squares(5)* = $1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 55$

Source Code 3.1: Example of solution for *Sum of Squares* problem using **for** iteration loop.

```python
def sum_of_squares(n):
  total = 0
  for i in range(1, n+1):
    total = total + i**2
  return total
```

Source Code 3.2: Example of solution for *Sum of Squares* problem using **while** iteration loop.

```python
def sum_of_squares(n):
  total = 0
  while n >= 1:
    total = total + (n**2)
    n = n - 1
  return total
```

Prime Numbers

Write a program that receives a positive integer *n* as input and returns *True* if *n* is a prime number, or *False* otherwise.

Examples:

- *is_prime_number(1) = False*
- *is_prime_number(7) = True*
- *is_prime_number(10) = False*

Source Code 3.3: Example of solution for *Prime Numbers* problem using **for** iteration loop.

```python
def is_prime_number(n):
  count = 0
  for i in range(1, n+1):
    if n % i == 0:
      count += 1
  return count == 2
```

Source Code 3.4: Example of solution for *Prime Numbers* problem using **while** iteration loop.

```python
def is_prime_number(n):
  i = 2
  while n > i :
    if n % i == 0:
      return False
    i += 1
  return True and n != 1
```

Fibonacci

Write a program that receives a positive integer *n* as input and returns the *n*th element of the Fibonacci sequence $(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...)$.

Since some students may not know how the Fibonacci sequence is produced, we provided the following definition in the problem description: *"The Fibonacci sequence is a set of numbers that starts with a one or a zero, followed by a one, and proceeds based on the rule that each number is equal to the sum of the preceding two numbers"*. In addition, we found that the Fibonacci problem can be very difficult for some students. Therefore, we provided the following suggestion on how to get started: *"Declare in your program two variables: current = 0 and next_ = 1. From these two variables it is possible to calculate the following elements of the sequence"*.

Examples:

- *fibonnaci(0)* = $[0] = 0$
- *fibonnaci(2)* = $[0, 1, 1] = 1$
- *fibonnaci(6)* = $[0, 1, 1, 2, 3, 5, 8] = 8$
- *fibonnaci(7)* = $[0, 1, 1, 2, 3, 5, 8, 13] = 13$

Source Code 3.5: Example of solution for *Fibonacci* problem using **for** iteration loop.

```python
def fibonacci(n):
  current = 0
  next_ = 1
  for i in range(n):
    temp = current
    current = next_
    next_ = temp + next_
  return current
```

Source Code 3.6: Example of solution for *Fibonacci* problem using **while** iteration loop.

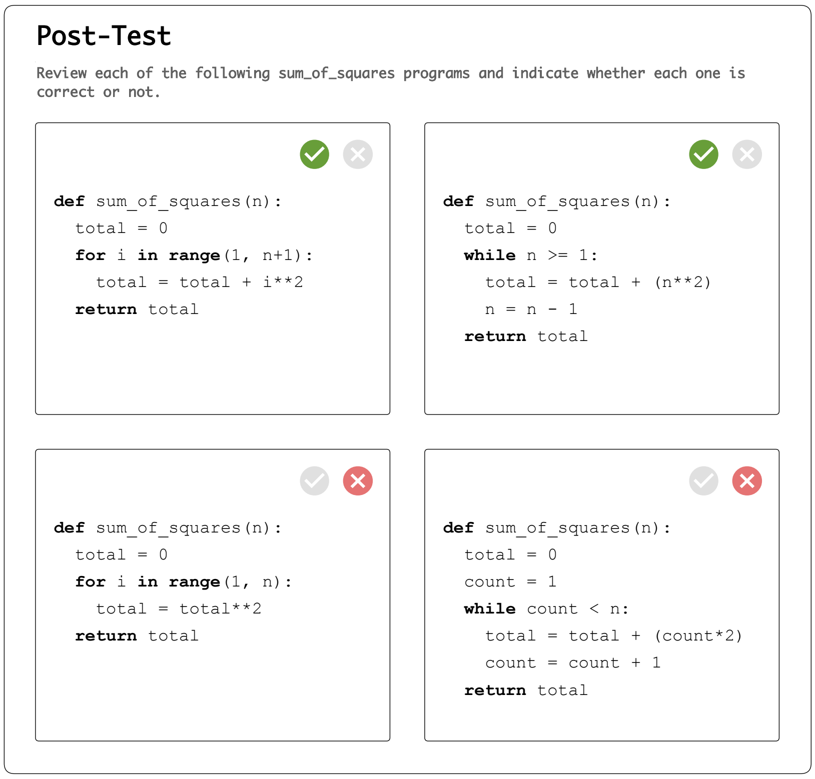
```python
def fibonacci(n):
  current = 0
  next_ = 1
  while (n > 0):
    temp = current
    current = next_
    next_ = temp + next_
    n = n - 1
  return current
```

3.5 Post-test

We proposed a post-test to measure the impact of tools on students' understanding of how to solve the problems. The post-test is presented immediately after the student has reached a correct solution to a problem. For each problem, we collected a set of correct and incorrect programs, and then we created a data source from them. The programs were obtained from a pilot study session and also from storage bases of programming assignments.

A post-test for a specific problem contains four different solutions for it, which may be correct or incorrect solutions. The proportion of correct/incorrect and selection of solutions are randomly defined by our platform. It is not possible that the same solutions are presented in the same post-test. The participant needs to review the solutions and indicate whether each solution is correct or not (as shown in Figure 3.3). It was not necessary to repair the incorrect programs. Since we are interested in measuring student‘s understanding based only on their knowledge, we did not allow participants to run the programs presented in the post-test.

For each correct statement in the post-test, one point is accumulated in the participant’s score on the problem addressed. As a result of the post-test, a range score of zero to four is generated to represent the student’s understanding of how to solve a given problem. This score is associated with the test cases or tool that the participant used to help solve the problem.


Post-Test

Review each of the following sum_of_squares programs and indicate whether each one is correct or not.

✓
```
def sum_of_squares(n):
  total = 0
  for i in range(1, n+1):
    total = total + i**2
  return total
```

✓
```
def sum_of_squares(n):
  total = 0
  while n >= 1:
    total = total + (n**2)
    n = n - 1
  return total
```

✗
```
def sum_of_squares(n):
  total = 0
  for i in range(1, n):
    total = total**2
  return total
```

✗
```
def sum_of_squares(n):
  total = 0
  count = 1
  while count < n:
    total = total + (count*2)
    count = count + 1
  return total
```

Figure 3.3: Post-test example for the *Sum of Squares* problem.

We analyzed the post-test scores as an indicator of student’s understanding of how to solve the problems addressed. This metric is based on one of Bloom’s taxonomy levels, which is *comprehension*. Bloom’s taxonomy [2] is a set of three hierarchical models used to classify educational learning objectives into levels of complexity and specificity. The level of *comprehension* involves demonstrating an understanding of facts and ideas by organizing, comparing, translating, interpreting, giving descriptions, and stating the main ideas.

3.6 Post-survey

Once students complete the post-test they are directed to a survey related to the problem solved and the tool used. In this survey the following question is presented to the participant: "How much did the [tool] used help you fix the bugs found in your program?". The *[tool]* pattern in the question was replaced by the name of the tool that the participant used, such as TEST CASES, CLARA, or PYTHON TUTOR. Students answered our question using a 7-point Likert scale, where one point means that the tool did not help and seven points means that the tool was indispensable for bug fixing. The scores from students' answers were associated with the tools used to solve problems. Subsequently, we analyzed the scores as an indicator of the usefulness of the tool in bug fixing.

3.7 Results

Overall, 42 undergraduate student from introductory programming classes participated in this study. In total, all participants produced 876 submissions. Our integration platform was able to provide feedback for 387 incorrect submissions. However, among other submissions, 9 failed in generation, 114 were correct and 366 had syntax errors. The tools evaluated in this study are not able to produce feedback when a submission contains syntax errors. In this case, we present the student with the compiler error message. However, students seem to have difficulty understanding what is wrong with their code through these messages. Becker et al. [1] propose to improve feedback through the use of enhanced compiler error messages.

For purposes of analysis, we considered only student solutions that received some feedback in at least one of the submissions. For this reason, we discarded data from 42 student solutions. In total, we analyzed 72 student solutions, of which 23 were obtained using only TEST CASES (TC), 21 were obtained working with CLARA (CL), and 28 were obtained by interacting with PYTHON TUTOR (PT). To better understand the effect of the tools on student performance, we segmented our data into datasets, as shown in Table 3.1.

Table 3.1: Datasets segmented by combining the *Sum of Squares* (SS), *Prime Numbers* (PN), and *Fibonacci* (FIB) problems.

DATASETS					
N°	PROBLEMS COMBINATION	TC	CL	PT	TOTAL
# 1	SS	7	4	9	20
# 2	PN	12	7	10	29
# 3	FIB	4	10	9	23
# 4	SS + PN	19	11	19	49
# 5	SS + FIB	11	14	18	43
# 6	PN + FIB	16	17	19	52
# 7	SS + PN + FIB	23	21	28	72
		NUMBER OF SOLUTIONS			

3.7.1 Overall analysis

This analysis refers to dataset #7, which contains all 72 students' solutions for the programming problems addressed in this study. Table 3.2 shows an overview of the results from our overall analysis. First, we analyzed how many submissions students made using each tool. In this analysis, students who used only TEST CASES required an average of 8.3 attempts to get a correct solution. This number decreased when they used CLARA (4.05) or PYTHON TUTOR (6.68). These differences are statistically significant when comparing CLARA with TEST CASES (Z = -2.5, $p < 0.02$) and also when comparing CLARA with PYTHON TUTOR (Z = -2.65, $p < 0.01$) by Wilcoxon-Mann-Whitney test.

The results of the post-test were analyzed as an indicator of the student's understanding of how to solve the problems addressed. Our overall analysis shows that there is no significant difference when comparing the post-test scores of students who used TEST CASES (2.39) with students who used CLARA (2.43). However, we observed statistically significant differences in scores obtained using PYTHON TUTOR (1.79), when compared to TEST

CASES (Z = -1.88, p < 0.05) and also when compared to CLARA (Z = 2.01, p < 0.05) by Wilcoxon-Mann-Whitney test.

Finally, in our post-survey, participants were asked how much each tool was useful for fixing bugs in their programs. Students scored the usefulness of TEST CASES for bug fixes with an average of 5.3. The CLARA and PYTHON TUTOR average scores were 5.9 and 5.4, respectively. There is a statistically significant difference when comparing TEST CASES with CLARA (Z = 2.10, p < 0.04) by the by Wilcoxon-Mann-Whitney test.

Table 3.2: Results of the overall analysis (dataset #7).

OVERALL ANALYSIS						
	TC	**CL**	**PT**	**TC - CL**	**TC - PT**	**CL - PT**
NUMBER OF ATTEMPTS	8.3 (7.2)	4.0 (2.5)	6.7 (5.3)	0.012	0.510	0.008
POST-TEST SCORES	2.3 (1.2)	2.4 (1.1)	1.8 (0.8)	0.903	0.049	0.043
UTILITY IN BUG FIXES	5.3 (1.4)	5.9 (1.8)	5.4 (1.7)	0.035	0.650	0.171
	AVERAGE (SD)			P-VALUE		

3.7.2 Individual analysis

In this analysis, we analyzed the tools considering each programming problem individually, which refers to datasets #1, #2, and #3. Through this analysis, we observed the effect of the tools in each problem addressed. In total, the dataset from the *Sum of Squares* problem (#1) contains 20 students' solutions; the dataset from the *Prime Numbers* problem (#2) contains 29 students' solutions; and the dataset from the *Fibonacci* problem (#3) contains 29 students' solutions. In the following, we describe the results found for each of these problems.

Sum of Squares (dataset #1): Table 3.3 shows an overview of the results obtained from this dataset. In particular, most participants found this problem easy. Students who used only TEST CASES required an average of 3.4 attempts to get a correct solution, when those using CLARA and PYTHON TUTOR required averages of 3.0 and 5.3 attempts, respectively. These differences are not statistically significant. In the post-test analysis, students using TEST CASES got an average score of 2.3, while those using CLARA and PYTHON TUTOR scored averages of 2.7 and 1.9, respectively. There is no statistically significant differences when comparing the post-test scores. Finally, in this dataset, we also did not find significant differences when analyzing the answers of our post-survey.

Table 3.3: Results of the analysis of the *Sum of Squares* problem (dataset #1).

INDIVIDUAL ANALYSIS: SUM OF SQUARES						
	TC	**CL**	**PT**	**TC - CL**	**TC - PT**	**CL - PT**
NUMBER OF ATTEMPTS	3.4 (1.9)	3.0 (1.4)	5.3 (2.5)	0.763	0.067	0.071
POST-TEST SCORES	2.3 (1.1)	2.7 (0.9)	1.9 (0.8)	0.492	0.469	0.139
UTILITY IN BUG FIXES	6.2 (0.7)	6.7 (0.5)	6.0 (1.1)	0.194	0.900	0.236
	AVERAGE (SD)			P-VALUE		

Prime Numbers (dataset #2): Table 3.4 shows an overview of the results obtained from this dataset. Students who used only TEST CASES required an average of 9.2 attempts to get a correct solution, while those using CLARA and PYTHON TUTOR required averages of 3.7 and 7.6 attempts, respectively. There is a statistically significant difference when comparing TEST CASES with CLARA (Z = -2.59, $p < 0.01$) by the by Wilcoxon-Mann-Whitney test. In the post-test analysis, students using TEST CASES got an average score of 2.6, while those using CLARA and PYTHON TUTOR scored averages of 2.4 and 1.5, respectively. These differences are statistically significant when comparing TEST CASES with PYTHON TUTOR (Z = -2.21, $p < 0.03$) by Wilcoxon-Mann-Whitney test. Finally, in this dataset, we did not find significant differences when analyzing the answers of our post-survey.

Table 3.4: Results of the analysis of the *Prime Numbers* problem (dataset #2).

INDIVIDUAL ANALYSIS: PRIME NUMBERS						
	TC	**CL**	**PT**	**TC - CL**	**TC - PT**	**CL - PT**
NUMBER OF ATTEMPTS	9.2 (7.3)	3.7 (1.6)	7.6 (7.3)	0.009	0.196	0.071
POST-TEST SCORES	2.6 (1.2)	2.4 (1.3)	1.5 (0.7)	0.793	0.026	0.104
UTILITY IN BUG FIXES	5.2 (1.2)	5.0 (1.8)	5.0 (1.6)	0.871	0.725	0.952
	AVERAGE (SD)			P-VALUE		

Fibonacci (dataset #3): Table 3.5 shows an overview of the results obtained from this dataset. In particular, most participants found this problem difficult. Students who used only TEST CASES required an average of 14.2 attempts to get a correct solution, while those using CLARA and PYTHON TUTOR required averages of 4.7 and 7.0 attempts, respectively. We observed a statistically significant difference when comparing TEST CASES with CLARA (Z = -2.00, $p < 0.05$) by the by Wilcoxon-Mann-Whitney test. In the post-test analysis, students using TEST CASES got an average score of 2.0, while those using CLARA and PYTHON TUTOR got averages of 2.3 and 2.0, respectively. These differences are not statistically significant. Finally, students scored the usefulness of TEST CASES for bug fixes with an average of 4.0. The CLARA and PYTHON TUTOR average scores were 6.2 and 5.0, respectively. There is a statistically significant difference when comparing TEST CASES with CLARA (Z = 2.45, $p < 0.02$) by the by Wilcoxon-Mann-Whitney test.

Table 3.5: Results of the analysis of the *Fibonacci* problem (dataset #3).

INDIVIDUAL ANALYSIS: FIBONACCI						
	TC	**CL**	**PT**	**TC - CL**	**TC - PT**	**CL - PT**
NUMBER OF ATTEMPTS	14.2 (8.4)	4.7 (3.2)	7.0 (4.9)	0.045	0.134	0.213
POST-TEST SCORES	2.0 (1.4)	2.3 (1.2)	2.0 (0.9)	0.606	0.807	0.609
UTILITY IN BUG FIXES	4.0 (2.0)	6.2 (2.0)	5.0 (2.3)	0.014	0.385	0.180
	AVERAGE (SD)			P-VALUE		

3.7.3 Paired analysis

In this analysis, we evaluated the tools considering the programming problems in pairs, which refers to datasets #4, #5, and #6. It is important to analyze in pairs to identify which problems the tool effects occur most frequently. Since paired analysis ignores data from one of the problems, we can see how much this problem affects the results found in the overall analysis. In total, the dataset from *Sum of Squares* and *Prime Numbers* problems (#4) contains 49 students' solutions; the dataset from *sum of squares* and *Fibonacci* problems (#5) contains 43 students' solutions; and the dataset from *Prime numbers* and *Fibonacci* problem

(#6) contains 52 students' solutions. In the following, we describe the results found for each of these datasets.

Sum of Squares and Prime Numbers (dataset #4): Table 3.6 shows an overview of the results obtained from this dataset. Students who used only TEST CASES required an average of 7.0 attempts to get a correct solution, when those using CLARA and PYTHON TUTOR required averages of 3.4 and 6.5 attempts, respectively. These differences are statistically significant when comparing CLARA with TEST CASES ($Z = -2.03$, $p < 0.05$) and also when comparing CLARA with PYTHON TUTOR ($Z = -2.54$, $p < 0.01$) by Wilcoxon-Mann-Whitney test. In the post-test analysis, students using TEST CASES got an average score of 2.4, while those using CLARA and PYTHON TUTOR scored averages of 2.5 and 1.6, respectively. There is no statistically significant difference when comparing the post-test scores of TEST CASES and CLARA. However, we observed statistically significant differences in scores obtained using PYTHON TUTOR, when compared to TEST CASES ($Z = -2.23$, $p < 0.03$) and also when compared to CLARA ($Z = 2.11$, $p < 0.04$) by Wilcoxon-Mann-Whitney test. Finally, in this dataset, we did not find significant differences when analyzing the answers of our post-survey.

Table 3.6: Results of paired analysis of the *Sum of Squares* and *Prime Numbers* problems (dataset #4).

PAIRED ANALYSIS: SUM OF SQUARES AND PRIME NUMBERS						
	TC	**CL**	**PT**	**TC - CL**	**TC - PT**	**CL - PT**
NUMBER OF ATTEMPTS	7.0 (6.5)	3.4 (1.5)	6.5 (5.5)	0.042	0.837	0.010
POST-TEST SCORES	2.4 (1.1)	2.5 (1.1)	1.6 (0.7)	0.876	0.025	0.034
UTILITY IN BUG FIXES	5.6 (1.1)	5.6 (1.6)	5.5 (1.4)	0.591	0.953	0.731
	AVERAGE (SD)			P-VALUE		

Sum of Squares and Fibonacci (dataset #5): Table 3.7 shows an overview of the results obtained from this dataset. Students who used only TEST CASES required an average of 7.3 attempts to get a correct solution, while those using CLARA and PYTHON TUTOR required averages of 4.2 and 6.1 attempts, respectively. These differences are not statistically significant. In the post-test analysis, students using TEST CASES got an average score of 2.1, while those using CLARA and PYTHON TUTOR got averages of 2.4 and 1.9, respectively. These differences are not statistically significant. Finally, students scored the usefulness of TEST CASES for bug fixes with an average of 5.3. The CLARA and PYTHON TUTOR average scores were 6.3 and 5.5, respectively. There is a statistically significant difference when comparing TEST CASES with CLARA (Z = 2.59, $p < 0.01$) by the by Wilcoxon-Mann-Whitney test.

Table 3.7: Results of paired analysis of the *Sum of Squares* and *Fibonacci* problems (dataset #5).

PAIRED ANALYSIS: SUM OF SQUARES AND FIBONACCI						
	TC	**CL**	**PT**	**TC - CL**	**TC - PT**	**CL - PT**
NUMBER OF ATTEMPTS	7.3 (7.3)	4.2 (2.8)	6.1 (3.8)	0.353	0.509	0.057
POST-TEST SCORES	2.1 (1.1)	2.4 (1.0)	1.9 (0.8)	0.551	0.687	0.216
UTILITY IN BUG FIXES	5.3 (1.7)	6.3 (1.6)	5.5 (1.7)	0.009	0.512	0.072
	AVERAGE (SD)			P-VALUE		

Prime Numbers and Fibonacci (dataset #6): Table 3.8 shows an overview of the results obtained from this dataset. Students who used only TEST CASES required an average of 10.4 attempts to get a correct solution, while those using CLARA and PYTHON TUTOR required averages of 4.2 and 7.3 attempts, respectively. These differences are statistically significant when comparing CLARA with TEST CASES (Z = -3.19, $p < 0.01$) and also when comparing CLARA with PYTHON TUTOR (Z = -2.13, $p < 0.04$) by Wilcoxon-Mann-Whitney test. In the post-test analysis, students using TEST CASES got an average score of 2.4, while those using CLARA and PYTHON TUTOR got averages of 2.3 and 1.7, respectively. These differences are not statistically significant. Finally, students scored the usefulness of TEST CASES for bug fixes with an average of 4.8. The CLARA and PYTHON TUTOR average scores were 5.6 and 5.0, respectively. There is a statistically significant difference when comparing TEST CASES with CLARA (Z = 1.90, $p < 0.05$) by the by Wilcoxon-Mann-Whitney test.

Table 3.8: Results of paired analysis of the *Prime Numbers* and *Fibonacci* (dataset #6).

PAIRED ANALYSIS: PRIME NUMBERS AND FIBONACCI						
	TC	**CL**	**PT**	**TC - CL**	**TC - PT**	**CL - PT**
NUMBER OF ATTEMPTS	10.4 (7.7)	4.2 (2.6)	7.3 (6.1)	0.001	0.071	0.032
POST-TEST SCORES	2.4 (1.2)	2.3 (1.1)	1.7 (0.8)	0.852	0.079	0.109
UTILITY IN BUG FIXES	4.8 (1.5)	5.6 (1.9)	5.0 (1.9)	0.049	0.721	0.235
	AVERAGE (SD)			P-VALUE		

3.8 Discussion

Through the results we found that CLARA can significantly reduce student's effort, in number of attempts, to get correct solutions (RQ1). This result was observed in our overall analysis and in most other datasets. This was already expected, since CLARA provides specific hints on how to get the correct solutions to programming problems. However, we also expected that the PYTHON TUTOR could reduce the number of attempts to get correct solutions. This was not observed in any of our analyzes. We thought that by debugging the code with PYTHON TUTOR, students would solve problems faster than using TEST CASES. We believe that there are two possible explanations for this result: (i) although we have provided a tutorial on PYTHON TUTOR, students may need more practice with the tool for better results; and (ii) as our study was conducted in introductory programming classes, students may not have enough experience for debugging activities.

In the post-test results, we found that students who used CLARA and those who used only TEST CASES got approximate scores (RQ2). No significant difference was observed when comparing these tools. This may mean that although CLARA provides specific hints on how to correct program bugs, the student's understanding of problem solving is not impaired. We noticed that, most of the time, students were trying to understand why they should apply the hints given by CLARA. This behavior may have led students to better understand how to solve problems, resulting in better performance in our post-test. This result is also supported by recent studies that have found that specific hints, such as those provided by CLARA, may be good for learning [29], [23].

In contrast, students who used PYTHON TUTOR got lower post-test performance than CLARA or TEST CASES. This result was observed in our overall analysis and in some other datasets. This is an unexpected result. We thought that by debugging the code, students would have an better overview of how to solve the problem. This result may be due to the unexpectedness of students with debugging activities. Another possible reason for this result would be that, when debugging their code, students focus only on a particular way of solving a problem. However, our post-test consists of analyzing different solutions to the same problem. Recent studies have found that for more effective pedagogical results using program visualization tools, such as PYTHON TUTOR, students need to be actively engaged with the tool [16], [33].

Finally, students scored CLARA as more useful than TEST CASES to fix bugs in their programs (RQ3). This result was observed in our overall analysis and in some other datasets, especially in datasets composed of solutions of the *Fibonacci* problem. Students mentioned that finding bugs in their programs using only TEST CASES was difficult, but using CLARA, they could easily find out where the bugs were.

3.9 Threats to Validity

In this section, we discuss possible threats to validity of our study. Our evaluation design sought to minimize the threats discussed whenever possible.

Internal validity

Since the study involves the active participation of humans, it is subject to internal threats. It is possible that the results were affected due to the moment and place where the experiments were conducted. Some of our study sessions happened in the classroom during class time. Participants were not previously advised that they would participate in this experiment. However, we let students know that their participation was not mandatory and that they could participate in the study in a private session. It is important to consider that students were solving programming problems, so it is possible that at some moment they are too tired or bored to perform their activities with involvement.

Participants in this study were recruited from Computer Science and Engineering courses. Although they are all enrolled in introductory programming classes, they may have different motivations and knowledge. Therefore, it is possible that some students were more experienced than others. To minimize this threat, we considered only in our analyzes the solutions that were correct, this ensures that the student was experienced enough to solve the problem. In addition, we discarded solutions from students who did not need any feedback to solve the problems, they were considered more experienced.

Construct validity

This study proposes a post-test to evaluate the student's understanding of the problem solution. However, the post-test score may not fully represent the student's understanding. There are many social aspects that can partially or totally affect the measurement of this construct. In addition, although we have observed a significant effect of the tools on the metrics, it is possible that the results found may not be entirely due to the tools used.

External validity

Participants in this study are representative only for the introductory programming context of our local universities (UFCG and IFPB). We also considered test cases as the baseline in our study because this is adopted in the programming courses of our local universities. However, the subjects and the baseline may not be representative for all educational institutions.

Conclusion validity

In our experiment, only three programming problems were addressed. However, a greater variety of programming problems may be required to generalize the conclusions. Other assignments that address different data structures and programming concepts may yield different results for different tools. Therefore, we may not be able to generalize the results of this experiment to other contexts. For more general results, this study should be replicated in other introductory programming subjects.

Due to time constraints and availability of participants, it was only possible to evaluate the tools for a short period of time. Perhaps the results would be different if students had more time to practice and learn about the tools. This study could draw further conclusions if carried out continuously over a long period of time.

3.10 Answers to the Research Questions

Next, we answer our research questions.

- **RQ1:** Do students using CLARA or PYTHON TUTOR solve problems faster than using only TEST CASES? **ANSWER:** Our results show that students using CLARA can solve problems faster, in number of attempts, than using TEST CASES. However, the same was not observed when students used PYTHON TUTOR.

- **RQ2:** What is the impact of using the tools in terms of understanding problem solving when compared to using TEST CASES alone? **ANSWER:** Students using CLARA or TEST CASES got approximate scores in our post-test. This may mean that they understood problem solving at the same level. However, in some of our analyzes, we found that students scored lower when using PYTHON TUTOR. This may mean that their understanding of problem solving was impaired or limited.

- **RQ3:** Do students find CLARA or PYTHON TUTOR more useful to fix bugs than TEST CASES? **ANSWER:** In some of our analyzes, students scored Clara's utility for bug fixes better than TEST CASES, especially in the problems they found most difficult. However, the same was not observed when students used PYTHON TUTOR.

Chapter 4

Related Work

In this chapter, we present the works related to ours. Section 4.1 describes related works on automated feedback generation. Next, Section 4.2 presents the works on visual and interactive debugging.

4.1 Automated Feedback Generation

Recent years have seen the emergence of automated feedback generation for programming assignments as a new, active research topic. Intelligent Tutoring Systems (ITSs) often supply a sequence of hints that descend from high-level pointers down to specific, bottom-out hints that spell out exactly how to generate the correct solution. For example, in the ANDES PHYSICS TUTORING SYSTEM, hints were delivered in a sequence: pointing, teaching, and bottom-out [36]. These systems have been historically expensive and time-consuming to build because they rely heavily on experts to construct hints.

Researchers have recently demonstrated how program synthesis, program repair techniques, and clustering algorithms can generate personalized and automated feedback for programming assignments (e.g., [18], [7], [27], [28], [31], [37], [11], [24]). For example, AUTOGRADER [31] can identify and fix bugs in incorrect code submissions, and then automatically generate sequences of increasingly specific hints about where the bugs are and what a student needs to change to fix them.

High-level hints that point to relevant class materials or attempt to reteach a concept can be difficult to automatically generate because they require more context or the deep domain knowledge of a teacher. To leverage the teacher's high-level feedback at scale, CODEOPTICON [13] provides a tutoring interface that helps teachers provide synchronous feedback for multiple students at once. Moreover, recent work has also demonstrated how program analysis and synthesis can be used as an aid for a teacher to scale feedback grounded in their deep domain knowledge [9], [14]. While reducing the teacher's effort, these systems still require teachers to manually review and write hints for incorrect student work.

Ihantola et al. [17] present a systematic literature review of the recent development of automatic assessment tools for programming assignments. They provide an overview of newly developed and currently available automated assessment systems. The systems included can be roughly divided into two categories: (1) automatic assessment systems for programming competitions; and (2) automatic assessment systems for (introductory) programming education. They discussed the key features of automatic assessment systems. From these, they pointed out that the differences in how tests are defined, how resubmissions are handled, and how the security is guaranteed were the most significant.

4.1.1 Automated Feedback Tools and Evaluations

Singht et al. [31] propose AUTOGRADER, a program synthesis based automated feedback generation for programming assignments. Their approach is to get a reference solution and an error model consisting of potential corrections to students errors, and search for the minimum number of corrections using a SAT-based program synthesis technique. This technology makes it possible to provide students with a measure of exactly how incorrect a given solution was, as well as feedback about what they did wrong. The researchers evaluated AUTOGRADER on thousands of student solutions on programming problems. They analyzed the effectiveness of AutoGrader in generating appropriate corrections as feedback for incorrect attempts. As a result, AUTOGRADER was able to provide feedback on more than 64% of incorrect solutions.

Gulwani et al. [11] present CLARA (Cluster And RepAir), a fully automated program repair tool for introductory programming assignments. This tool can automatically repair incorrect programs, indicate the location of bugs (e.g., line number), and provide an exact textual description of required changes. Its approach is to cluster the correct programs for a given assignment and select a canonical program from each cluster to form the reference solution set. Then, CLARA runs a trace-based repair procedure w.r.t. each program in the solution set and selects a minimal repair from the repair candidates. The researchers evaluated the number, size and quality of the generated repairs on thousands of incorrect student attempts, and then compared the results with AUTOGRADER using the same data. In addition, they conducted a user study about performance and usefulness of CLARA in an interactive teaching setting. In this study, participants solved six programming programs and answered a question about how useful the feedback provided by the tool was. As a result, CLARA was able to repair 97% of student attempts, in 3.2s on average, while 81% of those are small repairs of good quality. In addition, the tool got the average usefulness grade 3.4 on a scale from 1 to 5.

Wang et al. [37] propose SARFGEN (Search, Align, and Repair for Feedback GENeration), a data-driven program repair framework for generating feedback in introductory programming assignments. SARFGEN leverages the large number of available student solutions to generate instant, minimal, and semantic fixes to incorrect student submissions without any instructor effort. Its approach is to search for reference solutions similar to a given incorrect program, and then align each statement in the incorrect program with a corresponding statement in the reference solutions to identify discrepancies for suggesting changes. Finally, using this information, it can point out the minimum corrections to correct the incorrect program. The researchers analyzed the scalability and efficiency of SARFGEN in generating corrections on thousands of incorrect student attempts, and then compared the results with CLARA. In addition, they conducted a user study to measure the usefulness of the tool in a programming course. They analyzed the number of submissions and time that participants needed to get correct solutions. As a result, SARFGEN was able, within two seconds on average, generate concise, useful feedback for 89.7% of the incorrect student submissions. In addition, 96% of students using the tool completed their assignment within next two attempts and 97% of them finished within 30 minutes.

Head et al. [14] present MISTAKEBROWSER, a mixed-initiative system that allows teachers to combine their deep domain knowledge with the results of data-driven program synthesis techniques. This system relies on REFAZER [28], a data-driven program synthesis technique that learns code transformations from examples of bug fixes. In the MISTAKEBROWSER system, code transformations are learned from examples of student-written bug fixes. Then, the teacher reviews the incorrect student submissions that were clustered by the code transformations that corrected them. This way, the teacher can infer the shared misconception and write feedback for the whole cluster that includes explanations, hints, or references to relevant course materials. The researchers recruited 17 current and former teachers from the staff of an introductory programming class. Teachers had 40 minutes to review clusters of incorrect submissions and write feedback for each cluster. They then answered a few questions about the semantic coherence of the cluster, e.g., "Do these incorrect submissions share the same misconception?". Finally, teachers also answered Likert scale questions about their confidence in their descriptions and the depth of domain knowledge they added in the process. The results suggested that MISTAKEBROWSER helps teachers understand what mistakes and algorithms are common in student submissions. Teachers appreciated the generated fixes, but confirmed that a human in the loop is needed to review and annotate them with conceptual or high-level feedback.

Our study differs from others because we focus on analyzing the effectiveness of tools from the beginner student's perspective. We also developed a post-test to assess the impact of the tools on students' understanding of problem solving, as well as analyzed the usual metrics, such as number of attempts and student's perception of the usefulness of the feedback provided.

4.2 Design of Interactive Debugging Tools

One of the major challenges in learning to program is to relate code to the dynamics of program execution [3]. In an introductory programming course, many novice students have difficulties and misconceptions due to a lack of understanding of dynamic program execution [25]. One practical way to alleviate this cognitive difficulty is to visualize execution. Recently, researchers have proposed many program visualization tools (see [32] for a comprehensive review). These tools typically execute the program, store a snapshot of internal states at each execution step, and show a visual representation of runtime states such as stack frames, heap objects, and data structures [12]. Most educational tools are focused on visualizing and animating program aspects based on its runtime execution. Recent studies have found that using program visualization tools can be pedagogically effective if students actively engage with the tool [16], [33].

However, as program complexity increases, such visualizations can become confusing [35], and navigating the traces may become-time consuming. One design challenge is how to focus a student's attention on the differences between what the code does and what it is expected to do. Alternatively, recent debugging interfaces like WHYLINE [20], THESEUS [21] and TRACEDIFF [34] provide an overview of execution behavior and let a user find the cause of a bug through interactive question-answering, retroactive logging or execution traces.

One way visual and end-user programming environments have attempted to facilitate this exploratory programming process is through their support of "live" editing models, in which immediate visual feedback on a program's execution is provided automatically at edit time. The notion of "liveness" actually encompasses two distinct dimensions: (a) the amount of time a programmer must wait between editing a program and receiving visual feedback (feedback delay); and (b) whether such feedback is provided automatically, or whether the programmer must explicitly request it (feedback self-selection).

Hundhausen et al. [15] conducted an experimental study that investigated the impact of feedback self-selection on novice imperative programming. Their experiment adopted a within-subjects design that compared three different levels of syntactic and semantic feedback: (a) no syntactic or semantic feedback at all (the no feedback treatment); (b) syntactic and visual semantic feedback provided on demand when a "run" button is hit (the self-select treatment); and (c) syntactic and visual semantic feedback updated on every keystroke (the automatic treatment). They found that the self-select and automatic treatments promoted the development of programs with significantly fewer syntactic and semantic errors than those promoted by the no feedback treatment. However, they did not find significant differences between the self-select and automatic treatments with respect to either syntactic or semantic correctness.

4.2.1 Interactive Debugging Tools and Evaluations

Ko et al. [20] present WHYLINE, a debugging tool that lets a user find the cause of a bug through interactive question-answering. WHYLINE allows developers to choose a *"why did"* or *"why didn't"* question derived from the program's code and execution. The tool then finds one or more possible explanations for the output in question, using a combination of static and dynamic slicing, precise call graphs, and new algorithms for determining potential sources of values and explanations for why a line of code was not reached. The research analyzed four aspects of WHYLINE traces empirically: *slow down* (comparing normal running time to tracing time, as well as to profiling time), *trace size*, *compressed trace size*, and *trace loading time*. They also conducted a user study where participants had to find the cause of incorrect behaviors in programming assignments. The results showed that novice programmers with the WHYLINE were twice as fast as expert programmers without it.

Lieber et al. [21] propose THESEUS, an IDE extension that visualizes runtime behavior within a JavaScript code editor. This tool visualizes the program's runtime state using code coloring and marginal notes, allowing the programmer to perceive that information unobtrusively as they read the code. In addition, THESEUS organizes the log entries into a call tree that accounts for asynchronous invocations (such as event handlers), allowing programmers to quickly answer many time-consuming reachability questions. The researchers ran a lab study with undergraduate students, and interviewed nine professional programmers who were asked to use THESEUS in their day-to-day work. Participants in the first lab study performed programming tasks with and without the tool activated. In the second study, professional software developers were interviewed to see how THESEUS fit into their programming activities. The results showed that programmers enjoyed the availability of reachability coloring and call counts, and adopted new problem-solving strategies to take advantage of their strengths.

Guo et al. [12] present PYTHON TUTOR, a web-based program visualization tool for Python. This tool allows users to step forwards and backwards through execution to visualize the runtime state of a program's data structures. The PYTHON TUTOR backend takes the source code of a Python program as input and produces an execution trace as output. It executes the input program, stores a snapshot of internal states at each execution step, and shows a visual representation of runtime states such as stack frames, heap objects, and data structures. Karnalim et al. [19] evaluate PYTHON TUTOR based on a questionnaire survey applied in *Basic Data Structures* classes. Their purpose was to evaluate the impact of the tool to complete assignments, understand programming aspects, and collect information about the students' experience. According to the results, PYTHON TUTOR provides positive impacts for completing *Basic Data Structures* laboratory tasks and understanding general programming aspects (i.e. execution flow, variable content change, method invocation sequence, object reference, syntax error, and logic error). In addition, such tool also provides positive feedback when perceived from student experiences in general.

Suzuki et al. [34] propose TRACEDIFF, an automated feedback system that leverages both program synthesis and program visualization techniques to provide interactive personalized hints for introductory programming assignments. Given the student's incorrect code and a synthesized code fix, TRACEDIFF performs dynamic program analysis to capture the execution of both the incorrect and fixed code, comparing internal states and runtime behaviors. Then, it highlights how and where the trace of the incorrect code diverges from the trace of the fixed code. The student can inspect these behavioral differences further by clicking on an item in the trace, triggering the PYTHON TUTOR interactive visualization interface to render the stack frames and objects at that point of execution and indicate which line of code was just executed. The researchers conducted a controlled experiment with 17 students (with various levels of experience) where participants were asked to debug incorrect student code from introductory programming assignments. They compared student performance using TRACEDIFF with those using PYTHON TUTOR. During a 60-minute session, each participant was asked to perform two bug-fixing tasks for each incorrect code: (1) locate the bug and (2) fix the bug. They evaluated whether or not each participant correctly answered these questions and measured the time spent to complete these tasks. After the session, each participant rated and explained their experience using each tool. Although no statistically significant differences were found in the quantitative measures of the two groups, 64.7% of the participants believed that TRACEDIFF was the more valuable to identify and fix the bugs and 29.4% thought that both tools were equally important (only 5.9% preferred PYTHON TUTOR).

Our study differs from others because besides evaluating an interactive visual debugging tool (PYTHON TUTOR) only with novice programmers, we also compared the effectiveness of this tool with an automated feedback generation tool (CLARA) that provides specific hints. There are few works studies like ours that compare two completely different feedback approaches.

Chapter 5

Conclusions

In this work, we conduct user studies in introductory programming classes to evaluate the effectiveness of a tool for generating personalized hints (CLARA) and a program visualization tool (PYTHON TUTOR). We investigate whether using these tools students can solve programming assignments better than using only test case suites. Specifically, we analyze the effectiveness of these tools with respect to three aspects: (i) whether the tool leads students to faster results when compared to test cases alone; (ii) what is the impact of the tool on the student's understanding of how to solve the problem; (iii) whether the tool is more useful than using test cases alone.

In our study, we recruited 42 undergraduate students from Computer Science and Engineering courses enrolled in introductory programming classes. We asked participants to implement Python solutions for three programming problems. For each problem, they were able to use a feedback tool and a test-case suite to assist in the resolution process. Once the participants got a correct solution, they did a post-test related to the problem solved. The results of the post-test were analyzed as an indicator of the student's understanding of how to solve the problems addressed. Finally, we conducted a survey where participants were able to rate which tools they found most useful for fixing bugs.

Our results show that, when considering getting correct solutions faster, CLARA can significantly reduce the student's effort, in number of attempts, compared to PYTHON TUTOR and TEST CASES. We also observed that students who used CLARA and those who used only TEST CASES got approximate scores. In other words, it seems that the specific hints of CLARA does not limit student understanding about the problem solution. However, students using PYTHON TUTOR got lower post-test performance than CLARA or TEST CASES. This may highlight a difficulty for novice programmers in performing debugging activities. Finally, students scored CLARA as more useful than test cases to fix bugs in their programs. They mentioned that finding bugs using only test cases is difficult, but using CLARA, they can find out where the bugs are and immediately reflect on the hints provided.

In practice, more specific feedback can benefit beginner students' programming learning. It is therefore interesting that introductory programming teachers provide students with some tool that generates more specific feedback, such as CLARA or similar. However, TEST CASES should not be excluded. Ideally, students should also receive support from verification feedback. In this way, by combining these two types of feedback, students can better understand their mistakes and learn more effectively. On the other hand, teachers should be careful when providing beginning students with debugging tools such as PYTHON TUTOR. Since debugging activities require more experience and practice, students may have difficulty understanding their mistakes through a debugging tool.

This work contributes to studies about the effectiveness of tools that assist in programming education. We evaluated a personalized hints generation tool and a program visualization tool, however, studies still need to be done to assess the effectiveness of other types of tools. In addition, our results are based on a perspective of beginning students, however, other subjects as more experienced students should be considered in future studies. After all, feedback has an important role in cognitive learning and is essential for improving knowledge and skills acquisition [30].

5.1 Future Work

As future work, we intend to evaluate other feedback tools commonly used in programming education. There are many other tools that need to be evaluated and compared with existing ones. Then, we intend to establish guidelines for the use of feedback tools at each stage of programming learning. It is important to establish which types of tools are most effective at each stage of learning, so programming teaching can be improved in educational institutions. Finally, we want to develop a new post-test capable of assessing other aspects of learning outcomes and also conduct a longitudinal study based on this new post-test.

Bibliography

[1] Brett A. Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. Effective compiler error message enhancement for novice programming students. *Computer Science Education*, pages 148–175, 2016.

[2] B.S. Bloom. *Taxonomy of Educational Objectives: The Classification of Educational Goals*. Number v. 1 in Taxonomy of Educational Objectives: The Classification of Educational Goals. 1956.

[3] Benedict Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 1986.

[4] Matthew Butler and Michael Morgan. Learning challenges faced by novice programming students studying high level and low feedback concepts. In *Proceedings of ASCILITE - Australian Society for Computers in Learning in Tertiary Education Annual Conference 2007*.

[5] Albert T. Corbett and John R. Anderson. Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI, pages 245–252, 2001.

[6] Loris D'antoni, Dileep Kini, Rajeev Alur, Sumit Gulwani, Mahesh Viswanathan, and Björn Hartmann. How can automatic feedback help students construct automata? *ACM Trans. Comput.-Hum. Interact.*, pages 9:1–9:24, 2015.

[7] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 384–401, 2016.

[8] A. Drummond, Y. Lu, S. Chaudhuri, C. Jermaine, J. Warren, and S. Rixner. Learning to grade student programs in a massive open online course. In *IEEE International Conference on Data Mining*, pages 785–790, 2014.

[9] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Trans. Comput.-Hum. Interact.*, pages 7:1–7:35, 2015.

[10] Paul Gross and Kris Powers. Evaluating assessments of novice programming environments. In *Proceedings of the First International Workshop on Computing Education Research*, ICER, pages 99–110, 2005.

[11] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 465–480, 2018.

[12] Philip J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE, pages 579–584, 2013.

[13] Philip J. Guo. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software &#38; Technology*, UIST, pages 599–608, 2015.

[14] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S, pages 89–98, 2017.

[15] Christopher D. Hundhausen and Jonathan Lee Brown. An experimental study of the impact of visual semantic feedback on novice programming. *J. Vis. Lang. Comput.*, pages 537–559, 2007.

[16] CHRISTOPHER D. HUNDHAUSEN, SARAH A. DOUGLAS, and JOHN T.

STASKO. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, pages 259 – 290, 2002.

[17] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, 2010.

[18] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. Semi-supervised verified feedback generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 739–750, 2016.

[19] Oscar Karnalim and Mewati Ayub. The use of python tutor on programming laboratory session: Student perspectives. *Kinetik: Game Technology, Information System, Computer Network, Computing, Electronics, and Control*, pages 327–336, 2017.

[20] Andrew J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE, pages 301–310, 2008.

[21] Tom Lieber, Joel R. Brandt, and Rob C. Miller. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the 32th Annual ACM Conference on Human Factors in Computing Systems*, CHI, pages 2481–2490, 2014.

[22] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR, pages 125–180, 2001.

[23] Mary Muir and Cristina Conati. An analysis of attention to student – adaptive hints in an educational game. In *Intelligent Tutoring Systems*, pages 112–122, 2012.

[24] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. Sk-p: A neural program corrector for moocs. In *Companion Proceedings of the ACM*

SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH Companion, pages 39–40, 2016.

[25] Noa Ragonis and Mordechai Ben-Ari. On understanding the statics and dynamics of object-oriented programs. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE, pages 226–230, 2005.

[26] Ruan Reis, Gustavo Soares, and Melina Mongiovi. Evaluating feedback tools in introductory programming classes. *Frontiers In Education (to appear)*, 2019.

[27] Kelly Rivers and Kenneth R. Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education*, pages 37–64, 2017.

[28] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE, pages 404–415, 2017.

[29] Benjamin Shih, Kenneth Koedinger, and Richard Scheines. A response-time model for bottom-out hints as worked examples. pages 117–126, 2008.

[30] Valerie J. Shute. Focus on formative feedback. *Review of Educational Research*, pages 153–189, 2008.

[31] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 15–26, 2013.

[32] Juha Sorva. *Visual Program Simulation in Introductory Programming Education*. PhD thesis, 2012.

[33] Juha Sorva, Ville Karavirta, and Lauri Malmi. A review of generic program visualization systems for introductory programming education. *Trans. Comput. Educ.*, pages 15:1–15:64, 2013.

[34] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Björn Hartmann. Tracediff: Debugging unexpected code behavior using trace divergences. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017.

[35] Jaime Urquiza-Fuentes and J Angel Velázquez-Iturbide. A survey of program visualizations for the functional paradigm. In *Proceedings of the 3rd Program Visualization Workshop*, 2004.

[36] Kurt VanLehn, Collin Lynch, Kay Schulze, Joel A. Shapiro, Robert Shelby, Linwood Taylor, Don Treacy, Anders Weinstein, and Mary Wintersgill. The Andes physics tutoring system: Lessons learned. *International Journal of Artificial Intelligence in Education*, pages 147–204, 2005.

[37] Ke Wang, Rishabh Singh, and Zhendong Su. Search, align, and repair: Data-driven feedback generation for introductory programming exercises. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 481–495, 2018.

Appendix A

Informed Consent Form

TERMO DE CONSENTIMENTO LIVRE E ESCLARECIDO

Você está sendo convidado(a) a participar, como voluntário(a), da pesquisa intitulada UM ESTUDO SOBRE A EFETIVIDADE DE ABORDAGENS DE FEEDBACK NO ENSINO DE PROGRAMAÇÃO DE COMPUTADORES, conduzida pelos pesquisadores em Ciência da Computação: Mestrando Ruan Victor Bertoldo Reis de Amorim e Profa. Melina Mongiovi - Universidade Federal de Campina Grande. Este estudo tem por objetivo investigar a efetividade de abordagens de feedback no processo de ensino e aprendizagem de programação de computadores e algoritmos.

Você foi selecionado(a) por ser um(a) estudante universitário (maior de 18 anos) que está cursando a disciplina de Introdução à Ciência da Computação na Universidade Federal de Campina Grande (UFCG). Sua participação não é obrigatória, bem como não é remunerada. A qualquer momento, você poderá desistir de participar e retirar seu consentimento. Sua recusa, desistência ou retirada de consentimento não acarretará prejuízos para você.

Sua participação nesta pesquisa consiste em resolver três problemas de programação utilizando um ambiente virtual desenvolvido pelos pesquisadores. Este ambiente virtual será responsável por lhe ajudar, através do fornecimento de feedback automático, no processo de resolução destes problemas. Logo após solucionar algum dos problemas propostos, você será redirecionado para um quiz contendo um desafio extra com relação ao problema resolvido. Por fim, será aplicado um questionário eletrônico para saber sua opinião sobre as funcionalidades do ambiente virtual.

Dado que este estudo requer que sejam realizadas atividades de programação para resolução de problemas, eventualmente, você poderá sentir cansaço, estresse ou aborrecimento ao se deparar com dificuldades. Em razão da minimização desses efeitos indesejados, você pode, a qualquer momento, desistir de resolver alguns dos problemas de programação ou até desistir de participar do estudo por completo. Para lhe garantir uma experiência mais agradável, recomendamos que inicie suas atividades pelos problemas de programação os quais você acredite serem mais fáceis, e que lhe deixe mais confortável em resolvê-los. Em caso de dúvidas ou dificuldades, por favor, entre em contato com um dos pesquisadores disponíveis durante a execução do estudo. Em caso de danos decorrentes desta pesquisa, você receberá assistência integral e imediata, de forma gratuita, pelo tempo que for necessário, bem como terá direito a indenização pelos danos causados.

Esta pesquisa beneficia diretamente o seu aprendizado de programação, uma vez que lhe permite uma experiência prática na resolução de problemas no âmbito da computação, bem como lhe apresenta os principais meios tecnológicos utilizados para a resolução de erros no código-fonte. Os resultados obtidos através deste estudo também pode ajudar os professores a melhorar a qualidade do ensino de programação, direcionando-os para a utilização de abordagens e sistemas de feedback mais eficazes.

Os dados obtidos por meio desta pesquisa serão confidenciais e não serão divulgados em nível individual, visando assegurar o sigilo da sua participação. Estes dados serão armazenados e protegidos em um servidor local, o qual somente o pesquisador principal possui acesso. O pesquisador responsável se comprometeu a tornar públicos nos meios acadêmicos e científicos os resultados obtidos de forma consolidada sem qualquer identificação de indivíduos participantes.

Caso você concorde em participar desta pesquisa, assine ao final deste documento, que possui duas vias, sendo uma delas sua, e a outra, do pesquisador responsável da pesquisa. Quaisquer dúvidas sobre a pesquisa ou sua participação nela podem ser tiradas diretamente com Ruan Victor Bertoldo Reis de Amorim ou Melina Mongiovi, pelos e-mails ruanvictor@copin.ufcg.edu.br e melina@computacao.ufcg.edu.br, ou diretamente com Comitê de Ética em Pesquisa com Seres Humanos - CEP/ HUAC. Rua: Dr. Carlos Chagas, s/n, São José. Campina Grande- PB. Telefone: (83) 2101-5545.

Consentimento Livre e Esclarecido

Declaro que compreendi os objetivos desta pesquisa, como ela será realizada, os riscos e benefícios envolvidos e concordo em participar voluntariamente da pesquisa UM ESTUDO SOBRE A EFETIVIDADE DE ABORDAGENS DE FEEDBACK NO ENSINO DE PROGRAMAÇÃO DE COMPUTADORES.

Caso você concorde em participar da pesquisa, leia com atenção os seguintes pontos:

(a) você é livre para, a qualquer momento, recusar-se a participar do estudo e retirar seu consentimento. Sua recusa, desistência ou retirada de consentimento não acarretará prejuízo;

(b) você pode deixar de participar da pesquisa, não precisando apresentar justificativas para isso;

(c) sua identidade será mantida em sigilo, não sendo coletado nenhum dado de identificação pessoal;

(d) caso deseje, você poderá ser informado(a) de todos os resultados obtidos com a pesquisa.

____

Assinatura do(a) participante

Como pesquisador responsável pelo estudo UM ESTUDO SOBRE A EFETIVIDADE DE ABORDAGENS DE FEEDBACK NO ENSINO DE PROGRAMAÇÃO DE COMPUTADORES, declaro que assumo a inteira responsabilidade de cumprir fielmente os procedimentos metodologicamente e direitos que foram esclarecidos e assegurados ao participante desse estudo, assim como manter sigilo e confidencialidade sobre a identidade do mesmo.

Declaro ainda estar ciente que na inobservância do compromisso ora assumido estarei infringindo as normas e diretrizes propostas pela Resolução 466/12 do Conselho Nacional de Saúde – CNS, que regulamenta as pesquisas envolvendo o ser humano.

____

Assinatura do(a) pesquisador(a)

Campina Grande, 30 de novembro de 2018

Appendix B

Consubstantiated Opinion of the Research Ethics Committee

PARECER CONSUBSTANCIADO DO CEP

DADOS DO PROJETO DE PESQUISA

Título da Pesquisa: UM ESTUDO SOBRE A EFETIVIDADE DE ABORDAGENS DE FEEDBACK NO ENSINO DE PROGRAMAÇÃO DE COMPUTADORES

Pesquisador: RUAN VICTOR BERTOLDO REIS DE AMORIM

Área Temática:

Versão: 2

CAAE: 92256318.9.0000.5182

Instituição Proponente: Universidade Federal de Campina Grande

Patrocinador Principal: Financiamento Próprio

DADOS DO PARECER

Número do Parecer: 3.155.555

Apresentação do Projeto:

Trata-se de projeto que tem como instituicao proponente a Universidade Federal de Campina Grande, sem instituicoes Coparticipantes. E um estudo experimental que visa avaliar a eficacia dos sistemas tutores inteligentes (STIs) em turmas introdutorias de programacao.

Os sujeitos do estudo serao alunos universitarios maiores de 18 anos da UFCG regularmente matriculados em cursos de engenharia ou de tecnologia na Universidade Federal de Campina Grande, e que estejam cursando a disciplina de Introducao a Ciencia da Computacao. A coleta de dados será a partir de quizzes e formulário eletrônico.

Objetivo da Pesquisa:

O objetivo principal deste trabalho e investigar a efetividade dos sistemas de feedback automatico no ensino de programacao de computadores, identificando aspectos que evidenciam a efetividade deles em ajudar no processo de ensino-aprendizagem. Alem disso, pretende-se analisar em que circunstancias esses sistemas sao realmente uteis, e quais abordagens de feedback ajudam os estudantes a compreender e corrigir seus equivocos em relacao ao codigo-fonte. Em particular, este trabalho possui os seguintes objetivos especificos: (1) Realizar um estudo comparativo sobre estes sistemas, visando identificar quais deles sao efetivos e beneficos para o efeito de aprendizagem dos estudantes; (2) Identificar quais sao as vantagens e as desvantagens de cada abordagem de feedback utilizada pelos sistemas estudados nesta pesquisa.

Avaliação dos Riscos e Benefícios:

Riscos:

Dado que os participantes serao submetidos a tarefas relacionadas a resolucao de problemas de programacao, o mesmo pode eventualmente sentir cansaco, estresse ou aborrecimento ao se deparar com dificuldades. Em razao da minimizacao dos riscos mencionados, os participantes poderao, a qualquer momento, desistir de um determinado problema de programacao ou ate desistir de participar do estudo por completo. Os participantes tambem serao instruidos a iniciar suas atividades pelos problemas de programacao os quais eles acreditem ser mais faceis, e que se sintam mais confortaveis em resolver. Em casos de duvidas ou dificuldades, um dos pesquisadores sempre estara disponivel para ajudar a minimiza-los.

Beneficios:

Este trabalho beneficia diretamente o ensino de programacao de computadores em instituicoes educacionais, uma vez que investiga a efetividade da utilizacao de sistemas de feedback automatico no processo de ensino-aprendizagem. Portanto, os resultados desta pesquisa podem ajudar os professores a melhorar a qualidade do ensino de programacao, dessa forma, beneficiando tambem os alunos que podem obter melhores resultados de aprendizagem. Os participantes tambem poderao obter uma experiencia pratica na resolucao de problemas no ambito da computacao, bem como serao apresentados aos meios tecnologicos utilizados para a resolucao de erros no codigo-fonte.

Comentários e Considerações sobre a Pesquisa:

A pesquisa realizara um estudo com alunos universitarios de cursos relacionados a tecnologia, informatica, engenharias ou areas afins com o proposito de analisar quao eficazes sao os STIs no processo de ensino-aprendizagem de programacao de computadores e quanto estes sistemas ajudam os alunos a obterem melhores resultados de aprendizagem. Trata-se de pesquisa relevante para a sociedade mas que precisa satisfazer todas as exigencias dos CEPs acerca da documentacao a ser apresentada.

Considerações sobre os Termos de apresentação obrigatória:

O pesquisador apresentou a seguinte documentacao:

1- Comprovante de recepcao

2- Termo de compromisso dos pesquisadores;

3- Termo de compromisso de divulgacao dos resultados; 4- Folha de Rosto;

5- Informacoes Basicas do Projeto de Pesquisa;

6- Orcamento;

7- Projeto detalhado;

8- Termo de Consentimento Livre e Esclarecido – TCLE;

Recomendações:

Toda a lista de pendências da primeira submissão foi satisfeita, não havendo mais pendências.

Conclusões ou Pendências e Lista de Inadequações:

Aprovado.

Considerações Finais a critério do CEP:

Este parecer foi elaborado baseado nos documentos abaixo relacionados:

Tipo Documento	Arquivo	Postagem	Autor	Situação
Informações Básicas do Projeto	PB_INFORMAÇÕES_BÁSICAS_DO_PROJETO_1115109.pdf	07/12/2018 10:56:53		Aceito
Parecer Anterior	PB_PARECER_CONSUBSTANCIADO_CEP_3021153.pdf	07/12/2018 10:56:01	RUAN VICTOR BERTOLDO REIS DE AMORIM	Aceito
Projeto Detalhado / Brochura Investigador	PROJETO_DETALHADO.pdf	07/12/2018 10:54:02	RUAN VICTOR BERTOLDO REIS DE AMORIM	Aceito
Declaração de Pesquisadores	DECLARACAO_DE_DIVULGACAO_DOS_RESULTADOS.pdf	07/12/2018 10:48:39	RUAN VICTOR BERTOLDO REIS DE AMORIM	Aceito
Declaração de Pesquisadores	DECLARACAO_DE_PESQUISA_NAO_INICIADA.pdf	07/12/2018 10:44:13	RUAN VICTOR BERTOLDO REIS DE AMORIM	Aceito
Declaração de Pesquisadores	DECLARACAO_DE_ANEXO_DOS_RESULTADOS.pdf	07/12/2018 10:42:34	RUAN VICTOR BERTOLDO REIS DE AMORIM	Aceito
Declaração de Pesquisadores	TERMO_DE_COMPROMISSO_DO_ORIENTADOR.pdf	07/12/2018 10:39:47	RUAN VICTOR BERTOLDO REIS DE AMORIM	Aceito
Declaração de Pesquisadores	DECLARACAO_DE_CONCORDANCIA_COM_O_PROJETO_DE_PESQUISA.pdf	07/12/2018 10:38:41	RUAN VICTOR BERTOLDO REIS DE AMORIM	Aceito
Declaração de Pesquisadores	DECLARACAO_DE_PESQUISADOR_RESPONSAVEL.pdf	07/12/2018 10:37:28	RUAN VICTOR BERTOLDO REIS	Aceito

Declaração de Pesquisadores	DECLARACAO_DE_PESQUISADOR_RESPONSAVEL.pdf	07/12/2018 10:37:28	AMORIM	Aceito
Declaração de Instituição e Infraestrutura	TERMO_DE_ANUENCIA.pdf	07/12/2018 10:35:52	RUAN VICTOR BERTOLDO REIS DE AMORIM	Aceito
Orçamento	ORCAMENTO.pdf	07/12/2018 10:34:45	RUAN VICTOR BERTOLDO REIS DE AMORIM	Aceito
TCLE / Termos de Assentimento / Justificativa de Ausência	TERMO_DE_CONSENTIMENTO_LIVRE_E_ESCLARECIDO.pdf	07/12/2018 10:34:10	RUAN VICTOR BERTOLDO REIS DE AMORIM	Aceito
Cronograma	CRONOGRAMA.pdf	07/12/2018 10:28:53	RUAN VICTOR BERTOLDO REIS DE AMORIM	Aceito
Folha de Rosto	FOLHA_DE_ROSTO.pdf	07/12/2018 10:28:23	RUAN VICTOR BERTOLDO REIS DE AMORIM	Aceito

Situação do Parecer:
Aprovado

Necessita Apreciação da CONEP:
Não

CAMPINA GRANDE, 19 de Fevereiro de 2019

Assinado por:
Andréia Oliveira Barros Sousa
(Coordenador(a))