

UNIVERSIDADE FEDERAL FLUMINENSE

HENRIQUE DO PRADO LINHARES

# Provenance-enhanced Algorithmic Debugging

NITERÓI

2019

UNIVERSIDADE FEDERAL FLUMINENSE

HENRIQUE DO PRADO LINHARES

# Provenance-enhanced Algorithmic Debugging

Dissertation presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Research area: Systems and Information Engineering.

Advisor:

Prof. D.Sc. Leonardo Gresta Paulino Murta

Co-Advisor:

D.Sc. Troy Costa Kohwalter

NITERÓI

2019

Ficha catalográfica automática - SDC/BEE  
Gerada com informações fornecidas pelo autor

L735p Linhares, Henrique do Prado  
Provenance-enhanced Algorithmic Debugging / Henrique do  
Prado Linhares ; Leonardo Gresta Paulino Murta, orientador ;  
Troy Costa Kohwalter, coorientador. Niterói, 2019.  
70 f.

Dissertação (mestrado)-Universidade Federal Fluminense,  
Niterói, 2019.

DOI: <http://dx.doi.org/10.22409/PGC.2019.m.06127233720>

1. Programação (Computação). 2. Programa de computador.  
3. Engenharia de software. 4. Python (Linguagem de  
programação de computador). 5. Produção intelectual. I.  
Murta, Leonardo Gresta Paulino, orientador. II. Kohwalter,  
Troy Costa, coorientador. III. Universidade Federal  
Fluminense. Instituto de Computação. IV. Título.

CDD -

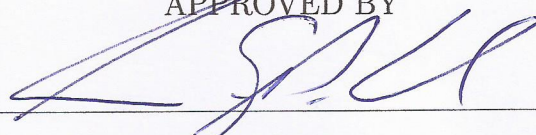
# HENRIQUE DO PRADO LINHARES

## Provenance-enhanced Algorithmic Debugging

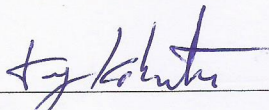
Dissertation presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfilment of the requirements for the degree of Master of Science. Research area: Systems and Information Engineering.

Approved in July, 2019.

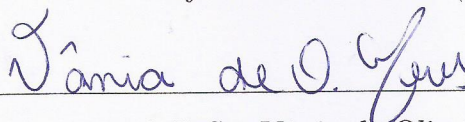
APPROVED BY



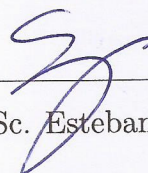
Prof. D.Sc. Leonardo Gresta Paulino Murta (Advisor), UFF



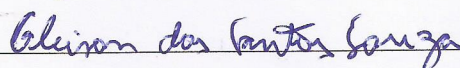
D.Sc. Troy Costa Kohwalter (Co-Advisor), UFF



Prof. D.Sc. Vânia de Oliveira Neves, UFF



Prof. D.Sc. Esteban Walter Gonzalez Clua, UFF



Prof. D.Sc. Gleison dos Santos Souza, UNIRIO

Niterói

2019

*For my family*

# Acknowledgements

Firstly, I would like to thank my parents and my family for all the support, patience, and believing in my potential.

This dissertation would not have been possible without the help, patience, and wisdom of my advisors Leonardo Murta and Troy Kohwalter.

I would like to thank the researcher João Felipe Pimentel for the valuable guidance and support.

I also would like to thank all students, researchers, and professors that I interacted during the last years.

I gratefully acknowledge the support of CAPES for the award of a research scholarship.

# Resumo

A localização de defeitos em software é uma atividade notoriamente difícil. Pesquisadores propuseram diversas técnicas para ajudar os desenvolvedores a localizar defeitos. Uma dessas técnicas é a depuração algorítmica, que consiste em executar o programa defeituoso, construir uma árvore de execução com as subcomputações, fazer perguntas ao desenvolvedor sobre a corretude de algumas subcomputações específicas e podar o espaço de busca de acordo com as respostas a estas perguntas. No entanto, dependendo da complexidade do programa, o número de perguntas pode ser alto, aumentando a duração da sessão de depuração. Neste trabalho propomos o DebugProv, uma abordagem de depuração algorítmica para programas em Python que aprimora a árvore de execução com proveniência para reduzir o número de questões necessárias para localizar o defeito e, consequentemente, reduzir a duração das sessões de depuração. Avaliamos nossa técnica em diferentes programas e descobrimos que ela conseguiu reduzir o número de perguntas em 25.2%, em média.

**Palavras-chave:** depuração algorítmica, proveniência, fatiamento de programa, defeitos de software

# Abstract

Localizing defects in a faulty software is a notoriously difficult activity. Researchers proposed several techniques to help developers to locate defects. One of these techniques is Algorithmic Debugging, which consists on executing the defective program, building an execution tree with the subcomputations, asking questions to the developer about the correctness of some specific subcomputations, and pruning the search space according to the answers to those questions. However, depending on the complexity of the program, the number of questions can be high, increasing the duration of the debug session. In this work we propose DebugProv, an algorithmic debugging approach for Python programs that enhances the execution tree with provenance to reduce the number of necessary questions to locate the defect, and, consequently, reduce the duration of debug sessions. We evaluated our technique over different programs and found that it was able to reduce the number of questions in 25.2%, on average.

**Keywords:** algorithmic debugging, provenance, program slicing, software defects



# List of Figures

2.1	Python script used as a guiding example . . . . .	6
2.2	The execution tree of our guiding example . . . . .	7
2.3	Example of <i>Single Stepping</i> Navigation Strategy (first step) . . . . .	9
2.4	Example of <i>Single Stepping</i> Navigation Strategy (second step) . . . . .	10
2.5	Example of <i>Single Stepping</i> Navigation Strategy (third step) . . . . .	10
2.6	Example of <i>Single Stepping</i> Navigation Strategy (fourth step) . . . . .	11
2.7	Example of <i>Single Stepping</i> Navigation Strategy (fifth step) . . . . .	11
2.8	Example of <i>Single Stepping</i> Navigation Strategy (sixth step) . . . . .	12
2.9	Example of <i>Single Stepping</i> Navigation Strategy (ET after navigation) . .	12
2.10	Example of <i>Top Down</i> Navigation Strategy (first step) . . . . .	13
2.11	Example of <i>Top Down</i> Navigation Strategy (second step) . . . . .	13
2.12	Example of <i>Top Down</i> Navigation Strategy (third step) . . . . .	14
2.13	Example of <i>Top Down</i> Navigation Strategy (fourth step) . . . . .	14
2.14	Example of <i>Top Down</i> Navigation Strategy (fifth step) . . . . .	15
2.15	Example of <i>Heaviest First</i> Navigation Strategy (first step) . . . . .	16
2.16	Example of <i>Heaviest First</i> Navigation Strategy (second step) . . . . .	17
2.17	Example of <i>Heaviest First</i> Navigation Strategy (third step) . . . . .	17
2.18	Example of <i>Heaviest First</i> Navigation Strategy (fourth step) . . . . .	18
2.19	Example of <i>Heaviest First</i> Navigation Strategy (fifth step) . . . . .	18
2.20	Example of <i>Divide and Query</i> Navigation Strategy (first step) . . . . .	19
2.21	Example of <i>Divide and Query</i> Navigation Strategy (second step) . . . . .	20
2.22	Example of <i>Divide and Query</i> Navigation Strategy (third step) . . . . .	21

2.23	Example of <i>Divide and Query</i> Navigation Strategy (fourth step) . . . . .	21
3.1	Architecture of DebugProv . . . . .	27
3.2	A provenance graph . . . . .	29
3.3	The provenance enhanced execution tree . . . . .	30
4.1	Boxplot of the number of questions required to locate the defective node without provenance enhancement (left) and with provenance enhancement(right). Outliers removed. . . . .	39
4.2	Boxplot of the number of questions required to locate the defective node without provenance enhancement (left) and with provenance enhancement(right) in the <i>Single Stepping</i> navigation strategy. Outliers removed. . . . .	43
4.3	Boxplot of the number of questions required to locate the defective node without provenance enhancement (left) and with provenance enhancement(right) in the <i>Top Down</i> navigation strategy. Outliers removed. . . . .	44
4.4	Boxplot of the number of questions required to locate the defective node without provenance enhancement (left) and with provenance enhancement(right) in the <i>Heaviest First</i> navigation strategy. Outliers removed. . . . .	45
4.5	Boxplot of the number of questions required to locate the defective node without provenance enhancement (left) and with provenance enhancement(right) in the <i>Divide and Query</i> strategy. Outliers removed. . . . .	46

# List of Tables

2.1	Navigation Strategies . . . . .	8
2.2	Steps of <i>Single Stepping</i> Navigation Strategy . . . . .	9
2.3	Steps of <i>Top Down</i> Navigation Strategy . . . . .	13
2.4	Steps of <i>Heaviest First</i> Navigation Strategy . . . . .	15
2.5	Weight of subtrees for <i>Heaviest First</i> navigation strategy . . . . .	16
2.6	Steps of <i>Divide and Query</i> Navigation Strategy . . . . .	18
2.7	Weight of nodes for <i>Divide and Query</i> Navigation Strategy . . . . .	19
2.8	Updated (1) weight of nodes for <i>Divide and Query</i> Navigation Strategy . .	20
2.9	Updated (2) weight of nodes for <i>Divide and Query</i> Navigation Strategy . .	21
2.10	Related work . . . . .	23
3.1	Improvements obtained by DebugProv in the guiding example . . . . .	31
4.1	Selected Programs . . . . .	36
4.2	Mutants Characterization . . . . .	37
4.3	Individual analysis of provenance enhancement by selected program. . . . .	40
4.4	Navigation strategies performance over trees with and without provenance enhancement. . . . .	41

# List of Acronyms and Abbreviations

ET	: Execution Tree;
AD	: Algorithmic Debugging;
Prov.	: Provenance;
Enhc.	: Enhancement;
Reduct.	: Reduction;
GADT	: Generalized Algorithmic Debugging and Testing;
IDTS	: Integrated Debugging, Testing and Slicing;
JavaDD	: Java Declarative Debugger;
DDJ	: Declarative Debugger for Java;
HDT	: Hybrid Debugging Technology;
LCS	: Longest Common Subsequence;
OOP	: Object-oriented Programming;
ODB	: Omniscient Debugging;

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Goal . . . . .	2
1.3	Organization . . . . .	4
<b>2</b>	<b>Algorithmic Debugging</b>	<b>5</b>
2.1	Introduction . . . . .	5
2.2	Guiding Example . . . . .	6
2.3	Capturing Phase . . . . .	6
2.4	Navigation Phase . . . . .	7
2.4.1	<i>Single Stepping</i> Navigation . . . . .	9
2.4.2	<i>Top Down</i> Navigation . . . . .	12
2.4.3	<i>Heaviest First</i> Navigation . . . . .	15
2.4.4	<i>Divide and Query</i> Navigation . . . . .	18
2.5	Related Work . . . . .	22
2.6	Final Remarks . . . . .	24
<b>3</b>	<b>DebugProv</b>	<b>26</b>
3.1	Introduction . . . . .	26
3.2	Capturing Module . . . . .	27
3.3	Enhancement Module . . . . .	28
3.4	Navigation Module . . . . .	30

3.5	Implementation . . . . .	31
3.6	Final Remarks . . . . .	32
<b>4</b>	<b>Evaluation</b>	<b>34</b>
4.1	Introduction . . . . .	34
4.2	Materials and Methods . . . . .	35
4.3	Results and Discussion . . . . .	38
4.3.1	Does the provenance enhancement reduce the number of questions in algorithmic debugging (RQ1)? . . . . .	38
4.3.2	How provenance enhancement improves each navigation strategy (RQ2)? . . . . .	40
4.3.2.1	How provenance enhancement improves <i>Single Stepping</i> strategy . . . . .	42
4.3.2.2	How provenance enhancement improves <i>Top Down</i> strategy	43
4.3.2.3	How provenance enhancement improves <i>Heaviest First</i> strat- egy . . . . .	44
4.3.2.4	How provenance enhancement improves <i>Divide and Query</i> strategy . . . . .	45
4.4	Threats to Validity . . . . .	46
4.5	Final Remarks . . . . .	48
<b>5</b>	<b>Conclusion</b>	<b>49</b>
5.1	Contributions . . . . .	49
5.2	Limitations . . . . .	49
5.3	Future Work . . . . .	50
	<b>References</b>	<b>53</b>

# Chapter 1

## Introduction

### 1.1 Motivation

Finding defects in software is a notoriously difficult activity. When a program presents an unexpected behavior, programmers must search, among several lines of code, the one that contains the defect. Developers in charge of the debugging task usually do this search in an unsystematic way, based on guesses [50, 27]. These guesses demand great effort and, the larger the software, the bigger the number of possible locations for the defects.

Nowadays, two techniques are widely used to help developers during the debugging task [39]: logging and breakpoint debugging. Both require the developer to select the right variables. When using the logging technique, if the selected variables are not related to the defect, then the values in the logs are also not going to be related to the defect [22]. A very similar limitation occurs with breakpoint debuggers but, instead of selecting a variable, the challenge is the selection of a statement to insert the breakpoint. Moreover, any change in the logged variables or the statements with breakpoints requires a complete re-execution of the program. Thus, these techniques are based on trial and error and eventually demands multiple executions to find a specific defect.

In the last decades, researchers proposed more elaborate debugging techniques, going far beyond the logging and breakpoint techniques mentioned above. Almost 40 years ago, Shapiro [43] introduced algorithmic debugging as an iterative technique to find defects. An algorithmic debugging session starts when a computation presents an incorrect or unexpected result. Then, the algorithmic debugger builds a tree that represents this computation. In this tree, the root node is the initial computation, and the children nodes are the routines (or function activations) that executed in that computation. Then, the algorithmic debugger asks questions to an oracle (usually a developer) to compare the

result of the routines with the expectations of the developer, e.g., “*the routine AVERAGE(10,20) returned 15. Is this valid (Y/N/I Don’t Know)?*”. The algorithmic debugger traverses the execution tree, checking the validity of the nodes to find the defective node: the **invalid** node that has all of its children nodes as valid nodes [5].

After the initial proposal of algorithmic debugging in the early 1980s, researchers proposed several concrete implementations for the functional programming paradigm [1, 16, 26, 32, 34]. GADT [10] was the first algorithmic debugger to work with an imperative programming language: Pascal. GADT used a static program slicing technique [48] to remove irrelevant nodes from the execution tree. The major limitation of GADT is the necessity to apply a program transformation technique before building the execution tree. Due to this program transformation, the questions asked to the developers are based on the transformed program, instead of the original one, which increases the difficulty of the algorithmic debugging questions. Another approach that works with imperative programs written in Java is JDD [3]. JDD employs equivalence classes to reduce the number of questions that the debugger presents to the developer. However, this technique only removes equivalent questions, still presenting a sub-optimum set of questions to developers.

Therefore, the main motivations of this dissertation are:

- Debugging is an expensive, time-consuming, and challenging activity.
- Algorithmic debugging has emerged as an alternative technique to locate defects in software. However, the number of questions to locate the defect can be high, leading to long debugging sessions.
- Hence, if we reduce the number of questions in a session, then it will reduce the time and effort necessary to locate a defect in a faulty program.

## 1.2 Goal

In this work, we propose DebugProv, an algorithmic debugging approach for Python programs. DebugProv enhances the execution tree with provenance captured from the execution of a defective program. In general, provenance refers to the origin of a data object [30]. In the specific context of debugging, it encompasses all the data and computations that were necessary to derive the program results.



A computer program often does more than one task and generates more than one result. When a program contains a defect and produces an incorrect result, the parts of the program that were not responsible for generating that incorrect result are irrelevant to find the defect in question. The provenance enhancement technique removes from the search space parts of the program that are irrelevant to a specific incorrect outcome.

The traditional data structure of algorithmic debugging is the execution tree, which represents all the routines (functions and methods) that were activated in a program execution. In our proposal, we enhance the execution tree with a provenance graph, a structure that represents which routines were responsible for producing each one of the results. By asking the programmer which outcome of the program is incorrect, we can select an intersection between the execution tree and the provenance graph: a selection containing only the nodes that were responsible for generating the incorrect outcome. Consequently, DebugProv can reduce the number of questions asked to developers during algorithmic debugging sessions by removing from the search space nodes that did not influence the production of the incorrect result.

We evaluated our proposed approach over a set of 15 Python programs. We first artificially inserted different defects into each one of them, generating 458 mutants [9]. Then, we ran DebugProv over these mutants and measured the number of questions asked to the developer to detect each defect. We contrasted the performance of DebugProv with the classic algorithmic debugging technique, without provenance enhancement. Our results show that provenance enhancement produces an average reduction of 25.2% in the number of algorithmic debugging questions. The reduction in the number of questions brings initial evidence that enhancing execution trees with provenance may reduce the duration and the effort necessary to locate defects in algorithmic debugging sessions.

As such, the main goal of this dissertation is to propose our DebugProv approach, which brings the following contributions:

- It is a novel approach to implement algorithmic debugging for an interpreted and dynamically-typed imperative language (i.e., Python).
- It is the first approach that uses provenance collected through dynamic program slicing to enhance the execution tree used in algorithmic debugging.
- It was able to reduce in 25.26% the number of required questions during algorithmic debugging sessions.

## 1.3 Organization

This dissertation is organized as follows:

- Chapter 2 introduces algorithmic debugging and related work.
- Chapter 3 describes our approach, including its architecture, the provenance enhancement technique, and the implementation aspects.
- Chapter 4 presents the DebugProv evaluation and discuss the results.
- Chapter 5 concludes our work and presents some future work.

# Chapter 2

## Algorithmic Debugging

### 2.1 Introduction

Algorithmic debugging is a semi-automated technique for locating defects in a faulty computer program. The technique consists of decomposing the computation that produces an incorrect output into smaller sub-computations and asking the developer about the correctness of the sub-computations until a defective sub-computation is found.

An algorithmic debugging session starts when a computation produces an incorrect output. Then, the debugger builds a data structure that represents the incorrect computation. This structure is usually a tree, which is called an execution tree. A navigation strategy is an algorithm used to traverse the execution tree, selecting nodes and asking questions to the developer about the correctness of the nodes. The developer answers the questions until the defective node is found.

Caballero et al. [5] interprets algorithmic debugging as a process with two phases: (i) capturing and (ii) navigation. The capturing phase is responsible for running the defective program and gathering data related to function executions with parameters and results to build an execution tree. The navigation phase is responsible for navigating in the execution tree nodes, asking questions to developers, to find the defective node.

In this chapter, we introduce a defective Python program in Section 2.2, which is used in a guiding example. In section 2.3, we describe the capturing phase of algorithmic debugging. In Section 2.4, we present the navigation phase, and describe the usage of four different navigation strategies (*Single Stepping*, *Top Down*, *Heaviest First* and *Divide and Query*) over our guiding example. In Section 2.5, we present the related work. We conclude this chapter with the final remarks in Section 2.6.

## 2.2 Guiding Example

In this chapter, we use the Python script presented in Figure 2.1 as an intentionally simple but didactic guiding example to introduce algorithmic debugging. This script prints a header (*print*), reads data from a file (*readfile*), finds the minimum value in the data (*find\_min*), and prints the count of elements and the minimum value (*print\_result*). The script has a defect in the function *find\_min*.

```
1 from json import load
2 FILE_NAME = 'data.json'
3
4 def readfile(filename):
5     f = open(filename, 'r')
6     return load(f)
7
8 def find_min(data):
9     current = float('inf')
10    for d in data:
11        if d < current:
12            pass # Defective Line
13    return current
14
15 def print_results(data, d_min):
16     d_count = len(data)
17     print(d_count)
18     print(d_min)
19
20 print("Count--Min:")
21 data = readfile(FILE_NAME)
22 pdata = find_min(data)
23 print_results(data, pdata)
```

Figure 2.1: Python script used as a guiding example

## 2.3 Capturing Phase

The **capturing phase** can employ techniques such as program transformation, code instrumentation, reflection, or modifications in the compiler to identify all function and method calls during the execution of the program and compose the execution tree. In this tree, nodes represent executed computations. The root node of the execution tree corresponds to the first call that started the computation. In many languages, the root node represents the function *main* of the program. Figure 2.2 illustrates the execution tree built from the execution of our guiding example. Note that the root node is the script *program.py* itself, a consequence of having code outside functions in Python.

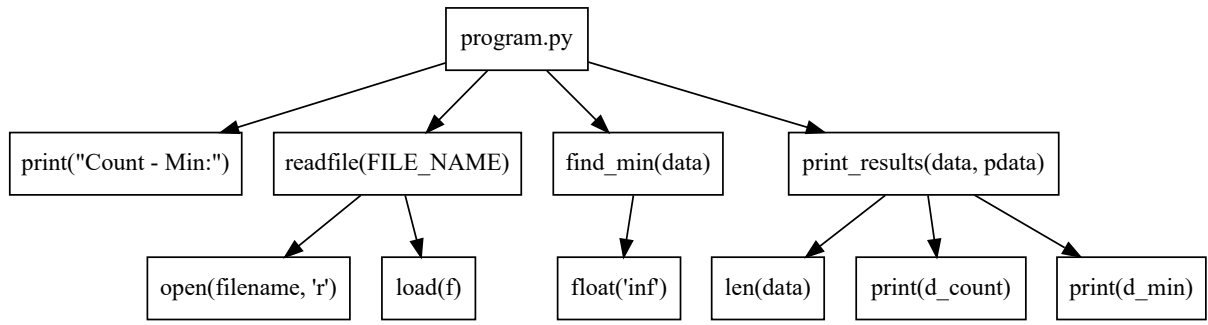


Figure 2.2: The execution tree of our guiding example

Every node in the execution tree can contain descendants. A descendant (or child) of a node  $n$  corresponds to a computation (a function or method, in Python) that was activated during the execution of  $n$ . In Figure 2.2, we can notice that the function *readfile* has two descendants: *open* and *load*. This indicates that the execution of the computation *readfile* activated two other computations: first, the computation *open*, and then, the computation *load*.

## 2.4 Navigation Phase

The **navigation phase** is the core of an algorithm debugging process. During the navigation phase, the algorithmic debugger uses a navigation strategy to traverse the generated execution tree and interacts with the user, asking questions about the correctness of the computations to find the defective node. The navigation strategy decides the order of nodes to visit and, consequently, the questions to be asked to the user. After receiving an input from the user, the algorithmic debugger process the answer and prunes the execution tree before moving forward and asking another question to the user. Depending on the order of questions and on the execution tree structure, an algorithmic debugger may prune more or fewer nodes.

Therefore, every question asked in an algorithmic debugging session leads to a prune. When a node  $n$  is evaluated and classified as valid (i.e., correct),  $n$  and all the subtree rooted at  $n$  is pruned (removed) from the search space. In Figure 2.2, when the debugger classifies *readfile* as valid, it marks both *open* and *load* as valid as well. On the other hand, in case that  $n$  is classified as invalid (i.e., incorrect), all nodes that are not descendants of  $n$  are removed from the search space. In other words,  $n$  becomes the new root of the execution tree. In Figure 2.2, it occurs when the debugger classifies *find\_min* as invalid. In this case, the debugging session continues to check if the defect is in the *find\_min* node

itself or if it is in one of its descendants.

The defective node is a node that is invalid while all of its children are valid. When the developer finds the defective node, the algorithmic debugging session stops and the debugger presents to the user the defective node, which relates to a defective function call since we are using an execution tree. The algorithmic debugger assumes that there is only one defective node per debugging session [7].

Nevertheless, the pruning process that occurs in the execution tree can be more or less effective depending on the sequence of questions asked to the developers, which depends on the chosen navigation strategy. Many navigation strategies have been proposed for algorithmic debugging in the literature [46]. The most relevant are *Single Stepping* [43], *Top Down* [1], *Heaviest First* [2], and *Divide and Query* [43]. Table 2.1 describes these strategies and presents the navigation steps to find the defective node *find\_min* in the execution tree presented in Figure 2.2 and we describe each in more details in the following subsections.

Table 2.1: Navigation Strategies

Nav. Strategy	Description	Steps in Figure 2.2	# Quest.
<i>Single Stepping</i>	Navigates the execution tree in a post-order depth-first traversal, respecting the original execution order (left to right).	print, open, load, readfile, float, find_min	6
<i>Top Down</i>	Navigates the execution tree in a breadth first traversal, respecting the original execution order (left to right).	program.py, print, readfile, find_min, float	5
<i>Heaviest First</i>	Variation of the <i>Top Down</i> strategy that selects the child with the biggest subtree instead of simply navigating according to the execution order.	program.py, print_results, readfile, find_min, float	5
<i>Divide and Query</i>	Navigates to nodes that prune almost half of the execution tree for every answer.	print_results, readfile, find_min, float	4

### 2.4.1 *Single Stepping* Navigation

The *Single Stepping* strategy navigates the execution tree in a post-order depth-first traversal, respecting the original execution order (left to right). The sequence of steps to locate the defective node with this navigation strategy is presented in Table 2.2, and the execution tree after the end of the navigation is presented in Figure 2.9.

Table 2.2: Steps of *Single Stepping* Navigation Strategy

Step	Node	Arguments	Return	Validity	Figure
1	print	"Count - Min:"	-	Valid	Figure 2.3
2	open	"data.json", "r"	<code>_io.TextIOWrapper</code>	Valid	Figure 2.4
3	load	<code>_io.TextIOWrapper</code>	<code>[738,967,667,122]</code>	Valid	Figure 2.5
4	readfile	"data.json"	<code>[738,967,667,122]</code>	Valid	Figure 2.6
5	float	"inf"	<code>inf</code>	Valid	Figure 2.7
6	find_min	<code>[738,967,667,122]</code>	<code>inf</code>	Invalid	Figure 2.8

Following the post-order depth-first traversal, the **first step** is to evaluate the node *print*. The algorithmic debugger informs the developer that the computation *print* received an argument that is a string, and printed the string without returning any value. Since this behavior was correct, the developer evaluates the node as valid. The algorithmic debugger defines the *print* node as a valid node. The execution tree after the first step is presented in Figure 2.3.

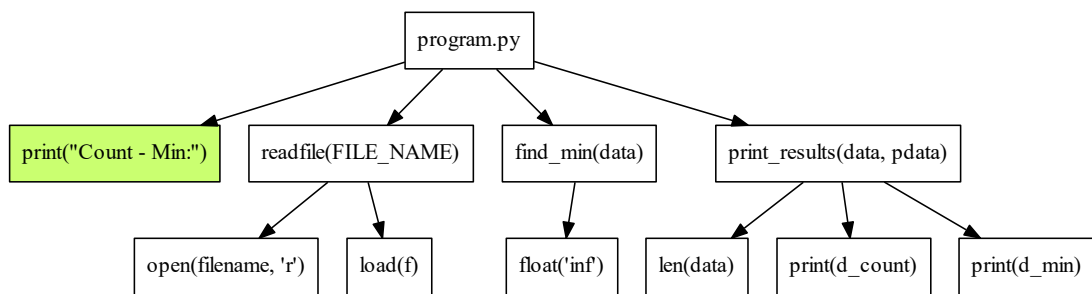


Figure 2.3: Example of *Single Stepping* Navigation Strategy (first step)

Following the *single stepping* strategy, the **second step** evaluates the node *open*. The debugger informs the developer that *open* received two arguments: An argument named filename, that stores the value 'data.json', and an argument named mode, containing the value 'r'. The computation returns a file object which can be used to read files. The description of the file object is: `_io.TextIOWrapper name='data.json' mode='r'`.

encoding 'cp1252'. The developer concludes that the behavior was correct, and answer that the node is valid. The algorithmic debugger defines the *open* node as a valid node. The execution tree after the second step is presented in Figure 2.4.

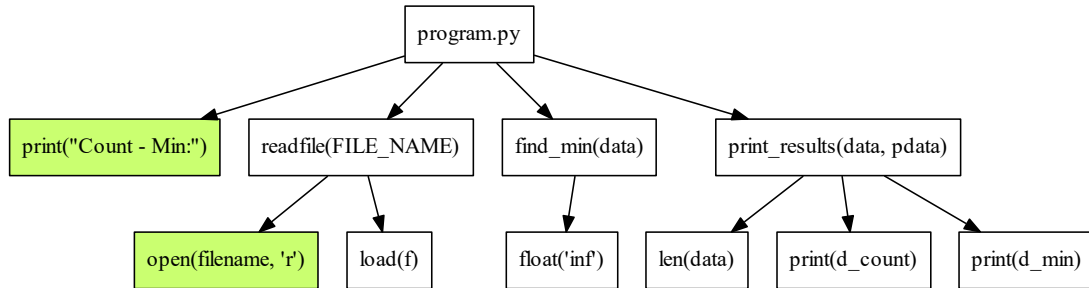


Figure 2.4: Example of *Single Stepping* Navigation Strategy (second step)

The **third step** evaluates the node *load*. The debugger informs the developer that the computation *load* received a file object as an argument, and returned a list containing the following numbers: 738, 967, 667, and 122. The developer confirms that the behavior was correct, and the debugger defines the node as valid. The execution tree after the third step is presented in Figure 2.5.

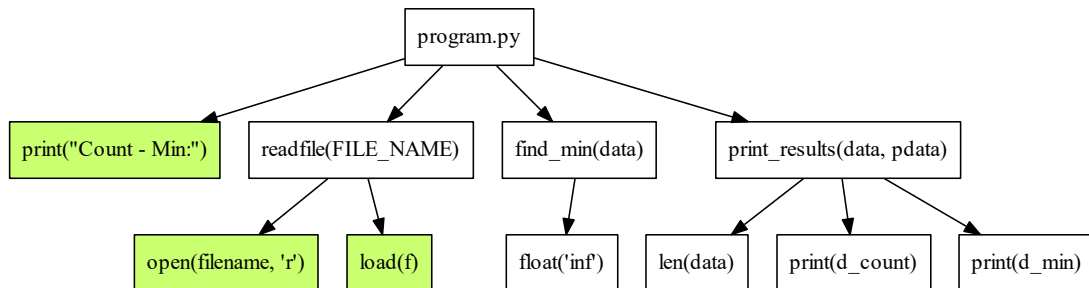
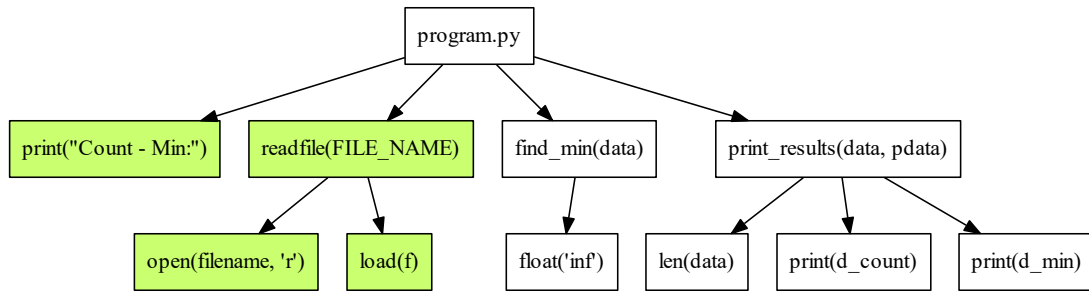


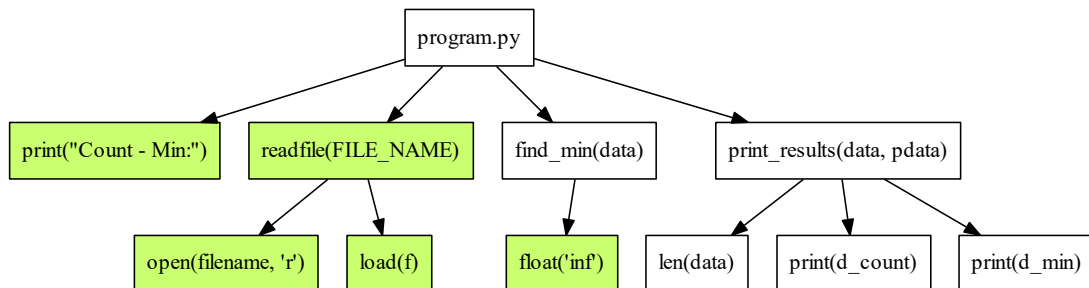
Figure 2.5: Example of *Single Stepping* Navigation Strategy (third step)

The **fourth step** evaluates the *readfile* node. The debugger informs the developer that the computation *readfile* received a string named *FILE\_NAME* as an argument, containing the value 'data.json', and returned a list containing the following numbers: 738, 967, 667, and 122. The developer evaluates *readfile* as a correct computation, and the debugger defines it as a valid node. The execution tree after the fourth step is presented in Figure 2.6.



Figure 2.6: Example of *Single Stepping* Navigation Strategy (fourth step)

The **fifth step** evaluates the *float* node. The debugger informs the developer that the computation *float* received as argument a string containing 'inf' and returned an object of class *float* containing the value *inf*. The developer evaluates *float* as a correct computation, and the debugger defines it as a valid node. The execution tree after the fifth step is presented in Figure 2.7.

Figure 2.7: Example of *Single Stepping* Navigation Strategy (fifth step)

The **sixth step** evaluates the *find\_min* node. The debugger informs the developer that the computation *find\_min* received as argument a list containing the following numbers: 738, 967, 667, and 122. The computation *find\_min* returned *inf*. In this case, the developer answer that this computation was not correct, since the smallest number between the list is 122, and not *infinite*. This step is presented in Figure 2.8.

After answering the question about *find\_min*, the debugger defines it as an invalid node. Since *find\_min* only have valid descendants, it is the defective node. The algorithmic debugging session ends in this step, and the developer is informed that the defective node is *find\_min*.

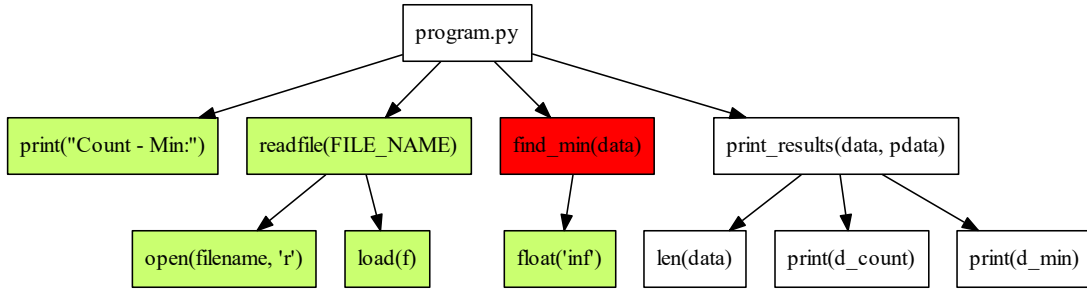


Figure 2.8: Example of *Single Stepping* Navigation Strategy (sixth step)

The execution tree after the end of the algorithmic debugging session is presented in Figure 2.9. The node painted with red color is the defective node, and the nodes painted with orange color are the invalid nodes. In Figure 2.9 the root node is the only invalid node.

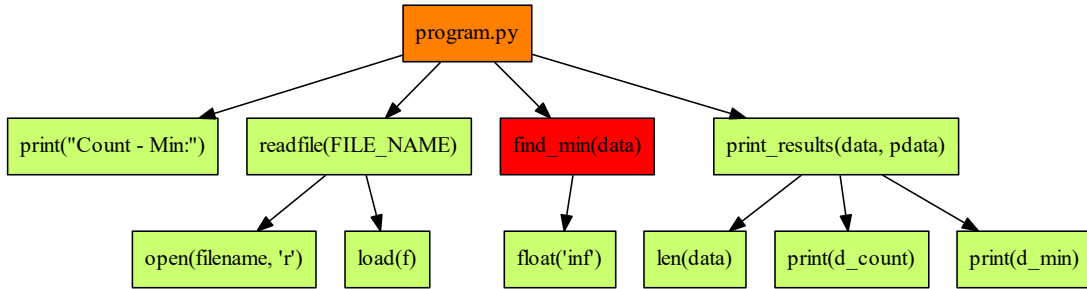


Figure 2.9: Example of *Single Stepping* Navigation Strategy (ET after navigation)

## 2.4.2 Top Down Navigation

This navigation strategy traverses the execution tree from top to bottom. The root node is the first to be evaluated. After evaluating a node  $n$ , the children of  $n$  are evaluated following the left to right order, i.e., the execution order. The sequence of questions for the *Top Down* navigation strategies is defined in Table 2.3.

The debugger first evaluates the root node (**first step**), which refers to the program itself. The developer answers that this node is presenting a wrong result, and the debugger marks the root as invalid. As this is the root node, the debugger does not prune any other node. The state of the execution tree after the first step of this algorithmic debugging

session is presented in Figure 2.10.

Table 2.3: Steps of *Top Down* Navigation Strategy

Step	Node	Arguments	Return	Validity	Figure
1	program.py	-	-	Invalid	Figure 2.10
2	print	"Count - min"	-	Valid	Figure 2.11
3	readfile	"FILE_NAME"	[738,967,667,122]	Valid	Figure 2.12
4	find_min	[738,967,667,122]	<i>inf</i>	Invalid	Figure 2.13
5	float	"inf"	<i>inf</i>	Valid	Figure 2.14

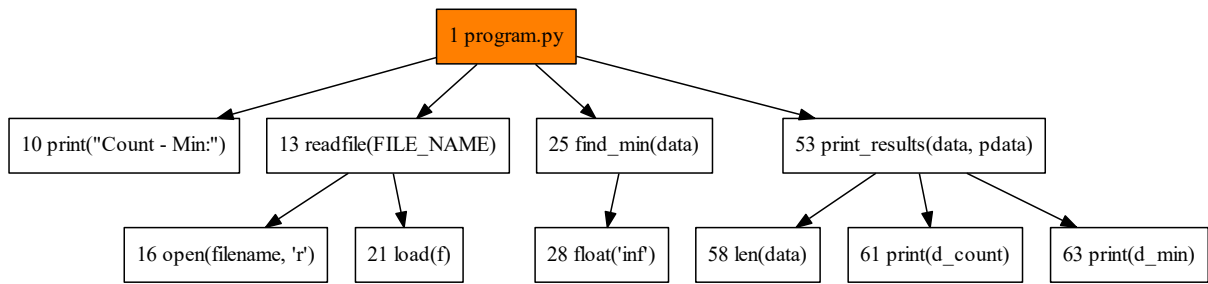


Figure 2.10: Example of *Top Down* Navigation Strategy (first step)

The **second step** evaluates the node *print*. The debugger informs that the function received a string and outputted it without returning any value. Since the developer answers that this behavior is correct, the debugger marks it as valid without pruning other nodes, as this node has no children. The second step of this algorithmic debugging session is presented in Figure 2.11. The node painted with green color is defined as valid and consequently removed from the search space.

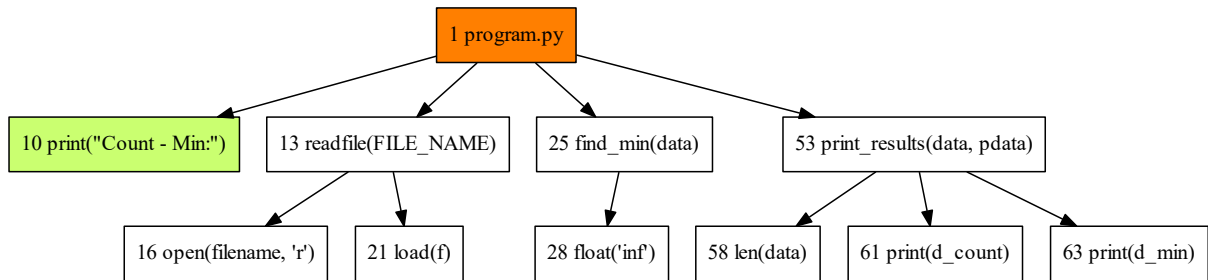


Figure 2.11: Example of *Top Down* Navigation Strategy (second step)

Then, in the **third step** the debugger evaluates *readfile*. The developer is informed that *readfile* received a parameter named *FILE\_NAME* with value *'data.json'* and returned an array with the following integer numbers: *738, 967, 667, 122*. As the computation of *readfile* is correct, the developer answers that the node is valid as well. Following the rules of algorithmic debugging pruning, the debugger classifies the node *readfile* and

all of its subtree nodes as valid, removing them from the search space. This operation results in the execution tree presented in Figure 2.12. The nodes in green are valid, and the nodes in orange are invalid. Just the white ones are still in the search space.

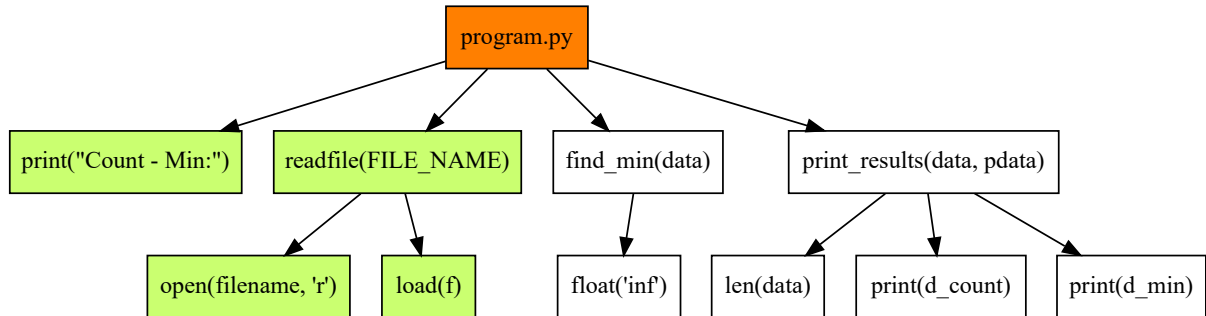


Figure 2.12: Example of *Top Down* Navigation Strategy (third step)

In the **fourth step**, the debugger evaluates the node *find\_min*. Here, the developer is informed that *find\_min* received a parameter named *data* with values *738,967,667,122* and returned the value *inf*. It is an incorrect computation: the smallest number among the inputs is 122, and not infinite. Thus, the developer answers that the computation is invalid. Following the algorithmic debugging pruning rules, the debugger defines *find\_min* as invalid, and all nodes that are not descendants of *find\_min* are marked as valid and consequently removed from the search space.

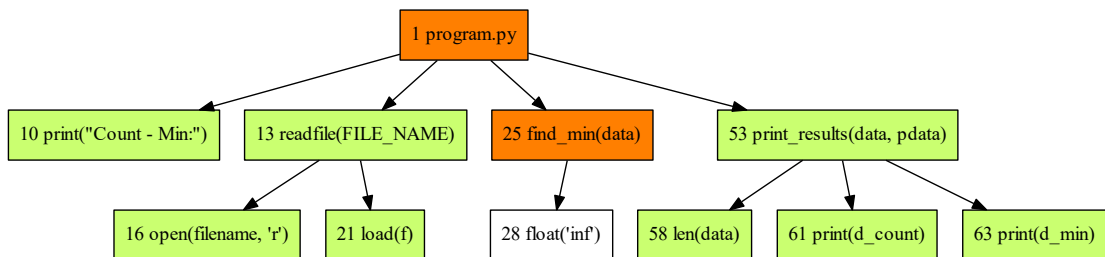
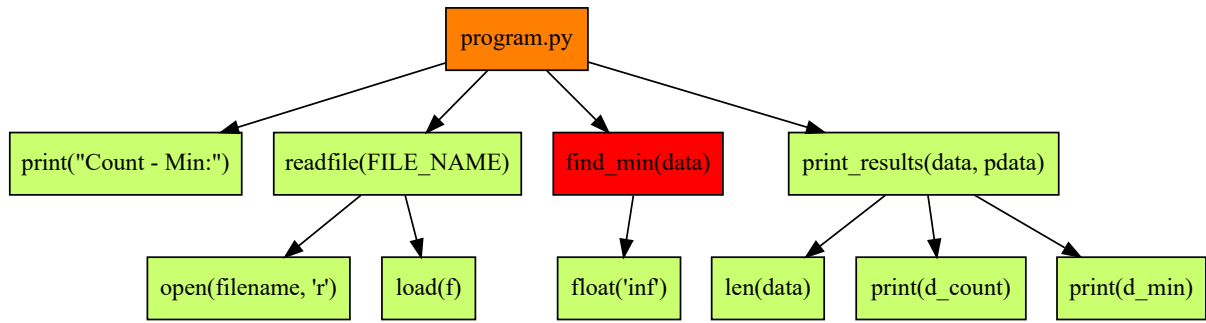


Figure 2.13: Example of *Top Down* Navigation Strategy (fourth step)

In the **fifth step**, the algorithmic debugger evaluates the descendant *float*. This node receives the string 'inf' and returns the float representation of infinite. Since the developer considers this as a correct behavior, the debugger marks this node as valid. Hence, the debugging session finishes, indicating that the defective node is *find\_min* since this is the only invalid node that has all of its children marked as valid. The resulting execution tree is presented in Figure 2.14. The red color distinguishes the defective node.

Figure 2.14: Example of *Top Down* Navigation Strategy (fifth step)

### 2.4.3 Heaviest First Navigation

The *Heaviest First* navigation strategy is a variation of the *Top Down* strategy. The *Top Down* strategy traverses the execution tree from the top to the bottom, following the execution order, that means, after evaluating a node  $n$ , the *Top Down* strategy select the first descendants of  $n$  following the execution order.

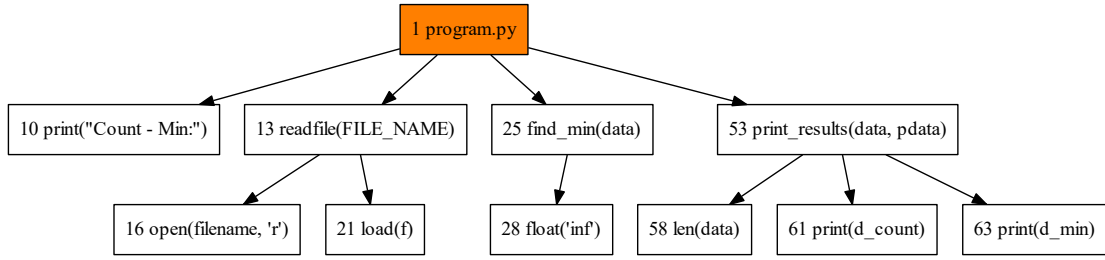
The difference between the *Top Down* and the *Heaviest First* is that the *Heaviest First* selects the descendant with the biggest sub-tree instead of simply navigating according to the execution order. The sequence of steps to locate the defective node with the *Heaviest First* navigation strategy is presented in Table 2.4, and the execution tree after the end of the navigation is presented in Figure 2.19.

Table 2.4: Steps of *Heaviest First* Navigation Strategy

Step	Node	Arguments	Return	Validity	Figure
1	program.py	-	-	Invalid	Figure 2.15
2	print_results	[738,967,667,122], <i>inf</i>	-	Valid	Figure 2.16
3	readfile	"data.json"	[738,967,667,122]	Valid	Figure 2.17
4	find_min	[738,967,667,122]	<i>inf</i>	Invalid	Figure 2.18
5	float	"inf"	<i>inf</i>	Valid	Figure 2.19

The *Heaviest First* navigation strategy first evaluates the root node (**first step**), which refers to the program itself. The developer answers that this node is presenting a wrong result, and the debugger marks the root as invalid. As this is the root node, the debugger does not prune any other node. The state of the execution tree after the first step is presented in Figure 2.15.

To select the next node to be evaluated, the *Heaviest First* strategy weights the subtrees of the descendants of *program.py*. The weight is calculated by measuring the

Figure 2.15: Example of *Heaviest First* Navigation Strategy (first step)

number of nodes in the subtree. In this step, the weights of each descendant of *program.py* are presented in Table 2.5.

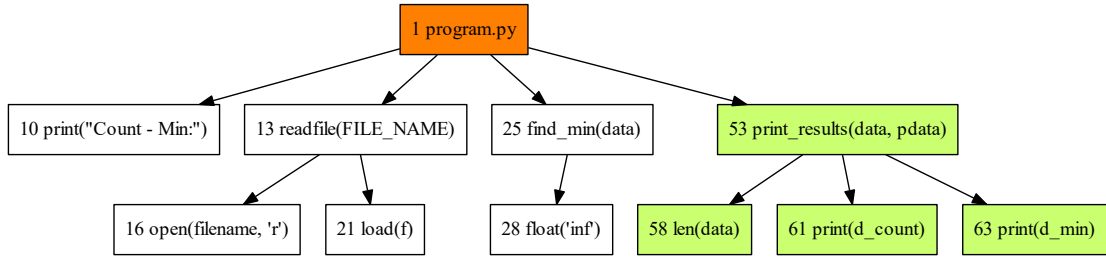
Table 2.5: Weight of subtrees for *Heaviest First* navigation strategy

Node	Weight
<code>print('`Count - min:')</code>	1
<code>readfile(FILE_NAME)</code>	3
<code>find_min(data)</code>	2
<code>print_results(data,pdata)</code>	4

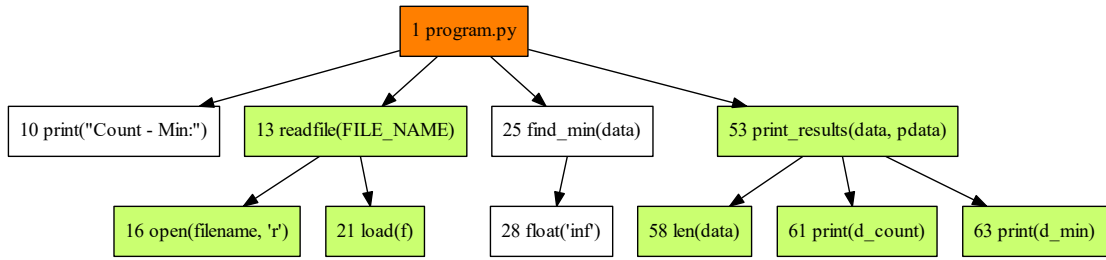
Since *print\_results(data,pdata)* is the node with the heaviest weight, it is selected to be evaluated on the **second step**. The debugger informs that the function received two arguments: a list containing integer numbers(*data*), and a number(*d\_min*) that is supposed to be the smallest number of the list. The length of the list and the value of *d\_min* are printed, and the function does not return any value. Since the developer answers that this behavior is correct, the debugger marks it as valid, and all the descendants of *print\_results* are consequently marked as valid and removed from the search space.

The **second step** of this algorithmic debugging session is presented in Figure 2.16. The nodes painted with green color are defined as valid and consequently removed from the search space.

Between the three descendants of *program.py* remaining in the search space (*print*, *readfile*, and *find\_min*) *readfile* is the one with bigger weight, so it is evaluated on the **third step**. The debugger informs that the function received a parameter named *FILE\_NAME* with value *data.json* and returned an array with the following integer numbers: 738, 967, 667, 122. The computation of *readfile* is correct, and the developer answers that the node is valid as well. The debugger classifies the node *readfile* and all of its subtree nodes as valid, removing them from the search space. This operation results in

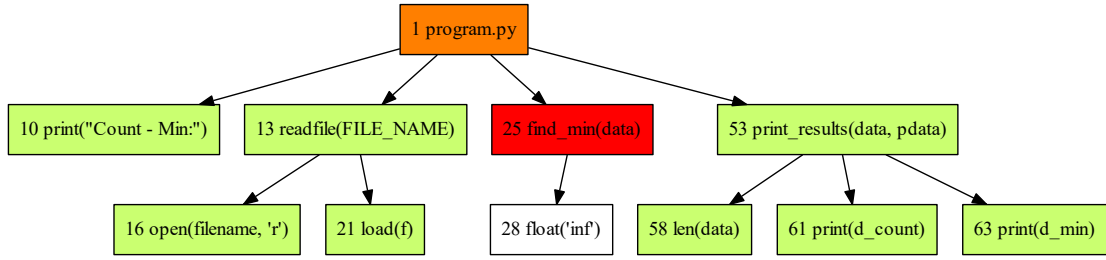
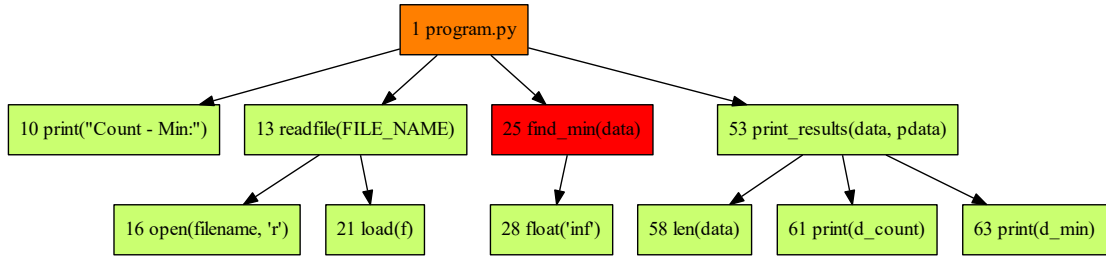
Figure 2.16: Example of *Heaviest First* Navigation Strategy (second step)

the execution tree presented in Figure 2.17. The nodes in green are valid, and the nodes in orange are invalid. Just the white ones are still in the search space.

Figure 2.17: Example of *Heaviest First* Navigation Strategy (third step)

In the **fourth step**, the debugger evaluates the node *find\_min*. Here, the developer is informed that *find\_min* received a parameter named *data* with values *738,967,667,122* and returned the value *inf*. It is an incorrect computation: the smallest number among the inputs is 122, and not infinite. Thus, the developer answers that the computation is invalid. Following the algorithmic debugging pruning rules, the debugger defines *find\_min* as invalid, and all nodes that are not descendants of *find\_min* are marked as valid and consequently removed from the search space.

In the **fifth step**, the only node with unknown validity is the *float*. The algorithmic debugger evaluates *float*. This node receives the string 'inf' and returns the float representation of infinite. Since this is a correct behavior, the debugger defines this node as valid. Hence, the debugging session finishes, indicating that the defective node is *find\_min* since this is the only invalid node that has all of its children marked as valid. The resulting execution tree is presented in Figure 2.19. The red color distinguishes the defective node.

Figure 2.18: Example of *Heaviest First* Navigation Strategy (fourth step)Figure 2.19: Example of *Heaviest First* Navigation Strategy (fifth step)

#### 2.4.4 *Divide and Query* Navigation

The *Divide and Query* navigation strategy aims to, in each step, divide the execution tree into two parts with similar weights. The steps of the *Divide and Query* strategy are presented in Table 2.6. The *Divide and Query* strategy starts calculating the weight of every node in the execution tree. The weight of a node is given by the size of its subtree. The weights for every node in our example are presented in table 2.7.

Table 2.6: Steps of *Divide and Query* Navigation Strategy

Step	Node	Arguments	Return	Validity	Figure
1	print_results	[738,967,667,122], <i>inf</i>	-	Valid	Figure 2.20
2	readfile	"data.json"	[738,967,667,122]	Valid	Figure 2.21
3	find_min	[738,967,667,122]	<i>inf</i>	Invalid	Figure 2.22
4	float	"inf"	<i>inf</i>	Valid	Figure 2.23

After calculating the weights of the nodes, the *Divide and Query* strategy calculates  $w$ , that corresponds to the weight of the nodes with unknown validity (or, the weight of the suspicious area [45]), and selects all the nodes with a weight smaller than  $w/2$ . The



node selected for evaluation is the one with the biggest weight, among the nodes with a weight smaller than  $w/2$ .

Table 2.7: Weight of nodes for *Divide and Query* Navigation Strategy

Node	Weight
program.py	10
print("Count - Min:")	0
readfile(FILE_NAME)	2
open	0
load	0
find_min	1
float	0
print_results(data, pdata)	3
len(data)	0
print(d_count)	0
print(d_min)	0

In the **first step**, the  $w$  corresponds to the weight of the root node, that is 10. In Table 2.7 we can see that among the nodes with a weight smaller than  $w/2$  the *print\_results* is the biggest one. So, it is the first selected for evaluation.

The debugger informs that *print\_results* received a list containing integer numbers(*data*), and a number(*p\_data*) that is supposed to be the smallest number of the list. The length of the list and the value of *p\_min* are printed, and the function does not return any value. Since the developer answers that this behavior is correct, the debugger marks it as valid, and all the descendants of *print\_results* are consequently marked as valid and removed from the search space. The execution tree after the first step is presented in Figure 2.20.

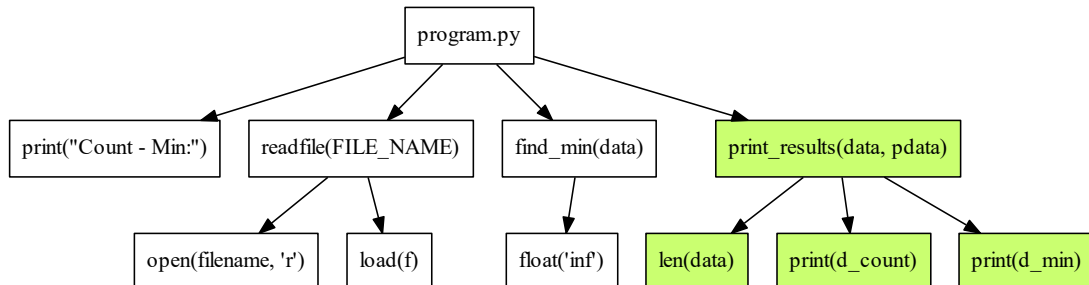


Figure 2.20: Example of *Divide and Query* Navigation Strategy (first step)

In the previous step, four nodes were defined as valid and removed from the search space. So, the *Divide and Query* strategy recalculates the weights of the nodes. Table 2.8

presents the updated weights of the nodes in the execution tree.

Table 2.8: Updated (1) weight of nodes for *Divide and Query* Navigation Strategy

Node	Weight
program.py	6
print("Count - Min:")	0
readfile(FILE_NAME)	2
open	0
load	0
find_min	1
float	0

In the **second step**, the algorithmic debugger evaluates the node *readfile*. The developer is informed that *readfile* received a parameter named *FILE\_NAME* with value *'data.json'* and returned an array with the following integer numbers: *738, 967, 667, 122*. As the computation of *readfile* is correct, the developer answers that the node is valid as well. Following the rules of algorithmic debugging pruning, the debugger classifies the node *readfile* and all of its subtree nodes as valid, removing them from the search space. This operation results in the execution tree presented in Figure 2.21.

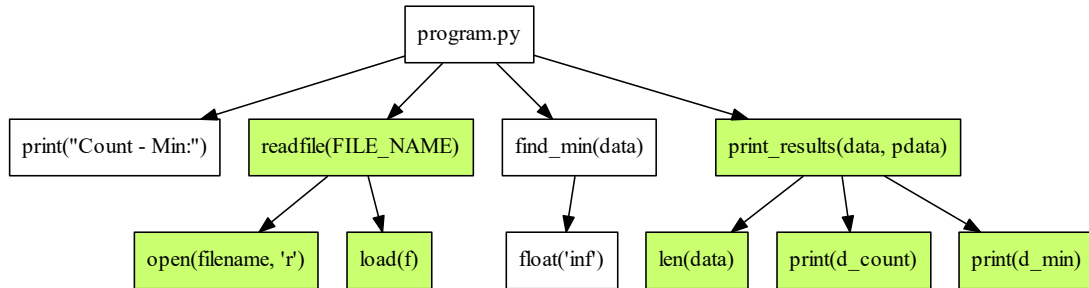


Figure 2.21: Example of *Divide and Query* Navigation Strategy (second step)

In the previous step, three nodes were defined as valid. The weights of the remaining nodes are recalculated as presented in Table 2.8.

In the **third step**, the debugger evaluates the node *find\_min*. Here, the developer is informed that *find\_min* received a parameter named *data* with values *738, 967, 667, 122* and returned the value *inf*. It is an incorrect computation: the smallest number among the inputs is 122, and not infinite. Thus, the developer answers that the computation is invalid. Following the algorithmic debugging pruning rules, the debugger defines *find\_min*

as invalid, and all nodes that are not descendants of *find\_min* are marked as valid and consequently removed from the search space.

Table 2.9: Updated (2) weight of nodes for *Divide and Query* Navigation Strategy

Node	Weight
program.py	6
print("Count - Min:")	0
find_min	1
float	0

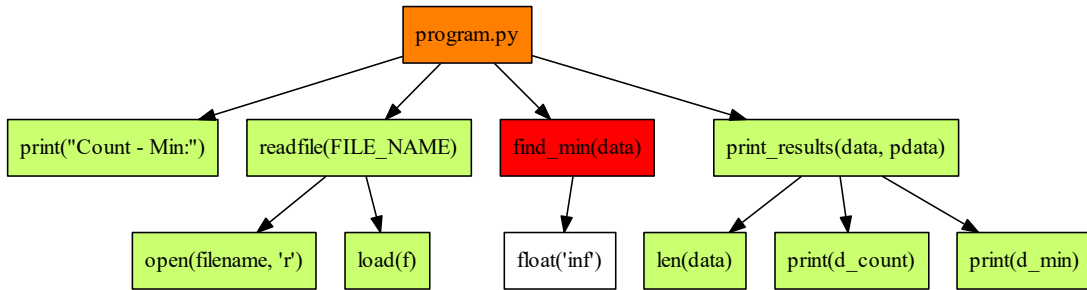


Figure 2.22: Example of *Divide and Query* Navigation Strategy (third step)

The navigation strategy updates the weights. In the **fourth step**, there is only one node with unknown validity: the *float* node. The algorithmic debugger evaluates *float*. This node receives the string 'inf' and returns the float representation of infinite. Since this is a correct behavior, the debugger defines this node as valid, defining *find\_min* as the defective node, as presented in Figure 2.23.

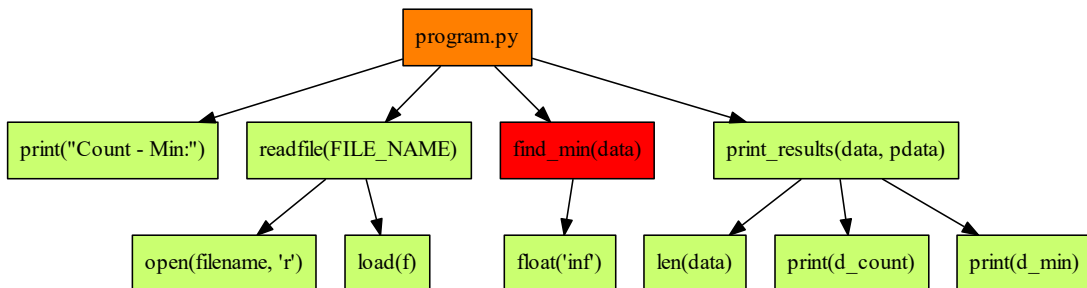


Figure 2.23: Example of *Divide and Query* Navigation Strategy (fourth step)

## 2.5 Related Work

Since the introduction of algorithmic debugging in 1982 [43], several tools were developed to instantiate the base concept in different programming paradigms and environments, such as logic [33] and functional [4, 38].

We have identified two approaches that combined algorithmic debugging with program slicing techniques to reduce the number of questions outside the context of *imperative* or *objected-oriented* programming languages. IDTS [23] is an algorithmic debugger for Prolog programs (logic programming paradigm). IDTS incorporates the Category Partition Testing Method (CPTM) [35] to remove unnecessary questions by comparing inputs and outputs to expected values of test cases. Additionally, a slicing method is applied to remove irrelevant nodes of the execution tree. The capability of IDTS to reduce the number of algorithmic debugging questions was assessed with an experiment over just three programs. They observed reductions of 0%, 50%, and 60% for the best strategy (i.e., Divide and Query) when the slicing technique was applied together with CPTM.

In the context of functional programming, a Haskell debugger combined the concept of algorithmic debugging with program slicing [47]. Instead of the traditional execution tree, this Haskell debugger applied program slicing into a new structure: the Augmented Redex Trails. Even though the authors described the inner mechanisms of the new technique, it was not evaluated through an experiment.

The first project that adapted the concept of algorithmic debugging for an imperative programming language with side-effects was presented in 1990 [42]. The implemented prototype, called GADT (Generalized Algorithmic Debugging and Testing), was able to run algorithmic debugging sessions in programs written in Pascal. GADT [42] aimed at reducing the number of questions by allowing users to indicate a variable that contains an incorrect value and using static program slicing to discover and remove the nodes of computation that were irrelevant to that variable. However, GADT employed program transformations that change the source code before the execution to perform algorithmic debugging in Pascal programs. These transformations have the disadvantage of requiring the developer to answer questions about a transformed program, instead of the program he or she is familiar with, increasing the difficulty of the task.

In 2003, HDT [25] was the first tool to bring algorithmic debugging to Java. HDT is presented as a hybrid debugger, which combines algorithmic debugging with breakpoint debugging. It uses the standard execution tree proposed in 1982, but the nodes of this

tree represents Java methods. Some other tools apply algorithmic debugging in Java. JavaDD [11] keeps the execution history of a Java program in a deductive database (built internally in Prolog) and performs debugging sessions through queries to this database. DDJ [17] is another algorithmic debugger for Java. It uses concepts of equivalence classes and def-use chains to reduce the number of questions. A later study performed by the authors of DDJ evaluated the number of questions asked by different navigation strategies over a set of execution trees, and concluded that the *Single Stepping* strategy presented the worst performance: it needs more questions than the other strategies. Among the four strategies discussed in this dissertation, the *Divide and Query* asked the smallest number of questions in this related work [46].

JHyde [15] is a hybrid debugger for Java that combines techniques from algorithmic debugging and *omniscient debugging* (ODB) [28], which is a technique that allows interruption and navigation in states of a program, but not only forward (like traditional breakpoint debuggers) but also backward, without needing to re-execute. HDJ [12] is an extension of DDJ [17] that combines algorithmic debugging, omniscient debugging, and breakpoint debugging.

In Table 2.10 we have organized related work that is related to algorithmic debugging in the context of *imperative* or *Objected-oriented* languages.

Table 2.10: Related work

Name	Language	Paradigm	Program Slicing	ODB	Year(s)	Publications
GADT	Pascal	Procedural	✓		1990-1998	[20, 42, 10, 19]
HDT	Java	OOP			2003	[25]
HDTS	Java	OOP	✓		2004	[24]
JavaDD	Java	OOP			2006	[11]
JDD	Java	OOP			2006-2007	[13, 3]
DDJ	Java	OOP			2010	[17]
JHyde	Java	OOP		✓	2011	[15]
HDJ	Java	OOP		✓	2013	[12]

In Table 2.10, we can see that GADT introduces the usage of static program slicing in (transformed) Pascal programs, and HDTS uses static program slicing in Java programs. In the column *ODB*, we present the debuggers that adopted *omniscient debugging*.

In the last years, researchers prioritized the development of tools that embraced multiple features. An example of this is that both of the tools presented by Hermanns and Kuchen [15] and González et al. [12] are capable of performing omniscient debugging.

This trend can be noticed in Table 2.10. The most recent approach that tried to apply program slicing over algorithmic debugging is HDTS[24] in 2004.

We also noticed that program slicing techniques have improved over the last years. Even though the *dynamic program slicing* produces a much more accurate slice [8], the potential of these improvements in program slicing was not applied in algorithmic debuggers yet.

One recent study presented by Insa et al. [18] introduced an approach that automatically balances the structure of the execution tree of Java programs by collapsing and projecting nodes. An experiment with real applications shows evidence that the balance technique can reduce the number of questions, speeding up the debugging sessions by 42%. According to the authors, the changes in the execution tree structure can lead to more difficult questions. The impact of the increase in the difficulty of the questions was not evaluated. This recent study, and also a survey performed in 2017 [5], provides evidence that the effort for reducing the number of questions in algorithmic debugging is still relevant.

## 2.6 Final Remarks

In this chapter, we presented the main concepts of algorithmic debugging, a technique that builds an execution tree that represents a faulty execution of a program, and traverses this tree until the defective node is found.

The size of the execution tree influences the number of questions directly: bigger execution trees leads to many questions and a longer debugging sessions. Approaches that reduce the size of the execution tree (or reduces the number of questions) can increase the efficiency of the algorithmic debugging sessions, reducing the time and effort necessary to locate the defect.

Our study of related works show that until now, all algorithmic debugging tools for imperative programming languages focused on statically typed programming languages, such as Pascal and Java. Moreover, there are only two tools that employed program slicing to enhance the execution tree in imperative or object-oriented languages programs until now: GADT and HDTS (presented in Table 2.10). None of these tools take advantage of the dynamic program slicing techniques.

In the next chapter, we describe the provenance enhancement, the technique we pro-

---

pose to enhance the execution tree and reduce the number of irrelevant questions in algorithmic debugging. We explain what the provenance enhancement is and how it works.

# Chapter 3

## DebugProv

### 3.1 Introduction

DebugProv is an algorithmic debugging approach that uses provenance enhancement to prune nodes from the execution tree before the navigation phase of the algorithmic debugging. It was implemented in Python and assists in the debugging of Python programs. It is an open-source project available for download at a public repository<sup>1</sup>. DebugProv currently runs either in command-line mode or embedded in a Jupyter Notebook<sup>2</sup>. The algorithmic debugging session starts in command-line mode by making the following command, according to our guiding example: *debugprov program.py*.

Our algorithmic debugger is composed of three modules. The capturing module is responsible for running the defective program while capturing execution data and building the execution tree. The provenance enhancement module is responsible for enhancing the execution tree with the provenance graph, and removing from the execution tree the nodes that were irrelevant for the incorrect output. The navigation module is responsible for traversing the execution tree (using a navigation strategy) and asking questions to the developer until the defective node is found. Figure 3.1 illustrates the architecture of DebugProv.

In this chapter, we present the capturing module in Section 3.2. The provenance enhancement module is presented in Section 3.3. We present the navigation module in Section 3.4. In Section 3.5 we discuss the implementation of DebugProv. The final remarks are presented in Section 3.6.

---

<sup>1</sup><https://github.com/gems-uff/debugprov>

<sup>2</sup>The Jupyter notebook is an open-source, web-based tool that allows developers to create and share documents that contain code, equations, visualizations, and narrative text [21].



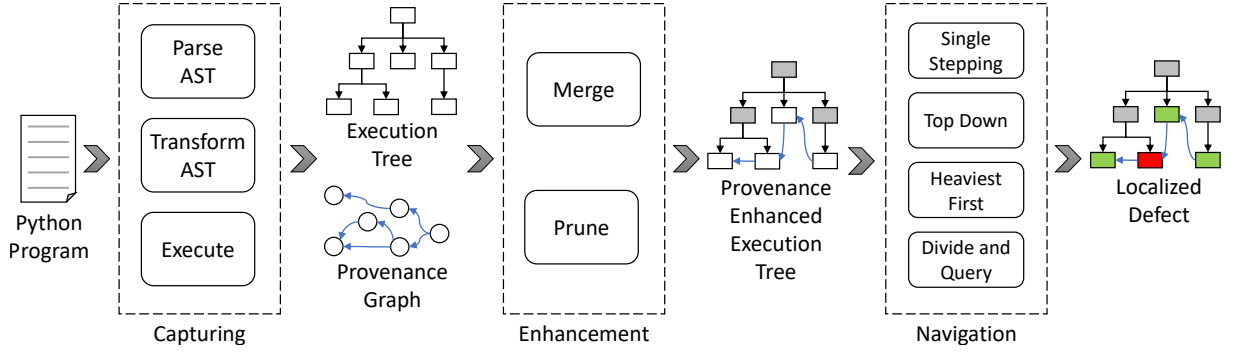


Figure 3.1: Architecture of DebugProv

## 3.2 Capturing Module

Suppose that we have a Python program that contains a logical defect and presents an incorrect result. The capturing module is responsible for running the defective program and capturing the data necessary to build an execution tree. At this moment, it also captures many other data, which compose the provenance of the incorrect outcome. We divide the captured data into two groups: the definition data and the execution data.

The **definition data** corresponds to the structure that can be extracted from the script before executing it. We capture definition data by constructing an Abstract Syntax Tree (AST) of the script code and extracting the necessary information from the AST. The definition data includes:

- Function definitions: information about functions that were defined in the source code.
- Parameters: information about received parameters of functions.
- Code components: information about code components of a Python script. Expressions and assigns are examples of code components.

The **execution data** corresponds to the data that is captured during the script execution. We capture execution data by transforming the AST before the execution to add function calls that collect the necessary information. This transformation is internal to the capturing module, with no side effects in the enhancement or navigation modules. The execution data includes:

- Function activations: information about functions that were activated (called) during the program execution and also the relationship between these functions (i.e., the sequence of activations in the call stack).

- Arguments and returns: the values of the arguments passed to the activated functions and their respective return values.
- Evaluations: the Python interpreter evaluates the code components of a Python program during their executions. Our approach stores the following information about the evaluated components: (1) the code component associated with an evaluation, (2) the activation associated with an evaluation, and (3) the representation (or the result of) that evaluation.
- Dependencies: by using dynamic program slicing [8], DebugProv can store the dependencies between evaluations. A dependency is a relationship between two evaluations, where the value of the dependent was influenced somehow by the value of the dependency. Capturing and storing the dependencies between evaluations is essential to perform the provenance enhancement in the execution tree.

In PROV terms, we map each Python evaluation to a PROV “entity”. These entities have “wasDerivedFrom” relationships to other entities, which are extracted from the dependencies collected by a dynamic program slicing technique. When the evaluation is an activation (function call), we represent its resulting value as an entity that “wasGeneratedBy” an “activity” that represents the activation. We indicate that this activity “used” the entities passed as arguments. As we capture the provenance recursively, we also use “wasInformedBy” relationships between activities to indicate which activity occurs in the context of others. Note, however, that we demonstrate the equivalences in PROV for didactic purposes, but the provenance is processed in a proprietary format, without exporting and importing to PROV [30].

As a result, this module provides the enhancement module both the execution tree and the provenance graph, which respectively contains the function activations and the dependencies among evaluations of code components. Considering our guiding example shown in Figure 2.1, while the execution tree informs that function *readfile* calls function *load*, the provenance graph informs that the input of *find\_min* depends on the results of *readfile*.

### 3.3 Enhancement Module

The enhancement module merges the provenance graph constructed through dynamic program slicing into the execution tree. This information allows DebugProv to perform

additional tree pruning based on the user input when answering typical debugging questions. Thus, the provenance enhancement potentially reduces the total number of required questions to detect defective nodes, speeding up the algorithmic debugging session.

The provenance enhancement allows this extra pruning before even starting the typical navigation in an algorithmic debugging session. It occurs by asking the developer which program output is incorrect. This question is straightforward to answer because the developer usually starts a debugging session when he or she observes some inappropriate output. By answering this question, our approach is capable of making an initial prune of all nodes in the execution tree that did not influence the inappropriate output, reducing the search space for the defective node.

Therefore, after the user report which output data is not correct, the provenance enhancement module searches in the provenance graph for the dependencies that influenced such incorrect output. This process is done in backward, answering the following question: "Which function activations contributed to the production of the incorrect output?"

We present a provenance graph from our guiding example in Figure 3.2. DebugProv merges the dependency relationships from the provenance graph into the execution tree. The result of this provenance enhancement can be observed in Figure 3.3. The blue edges represent the dependencies between nodes and they go from the influenced to the influencer node, as dictated by PROV [30], the W3C provenance notation. For example, an arrow from *find\_min* to *readfile* indicates that *readfile* influences *find\_min*. The nodes in grey do not belong to the provenance transitive closure and can be safely pruned from the search space.

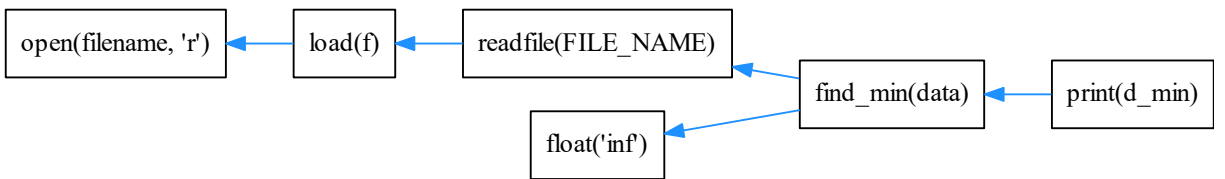


Figure 3.2: A provenance graph

Since the developer needs to inform the incorrect output, the current version of DebugProv does not work for silent defects (i.e., infinite loops or defects that consist of the absence of outputs). We intend to improve this in the future by investigating why-not provenance techniques [6]. Additionally, limitations in the provenance collection of DebugProv may prevent it from reducing the number of questions in some kinds of programs. For instance, when a node from the execution tree reads a file written by another node, there is an indirect provenance dependency that DebugProv does not consider. If the in-

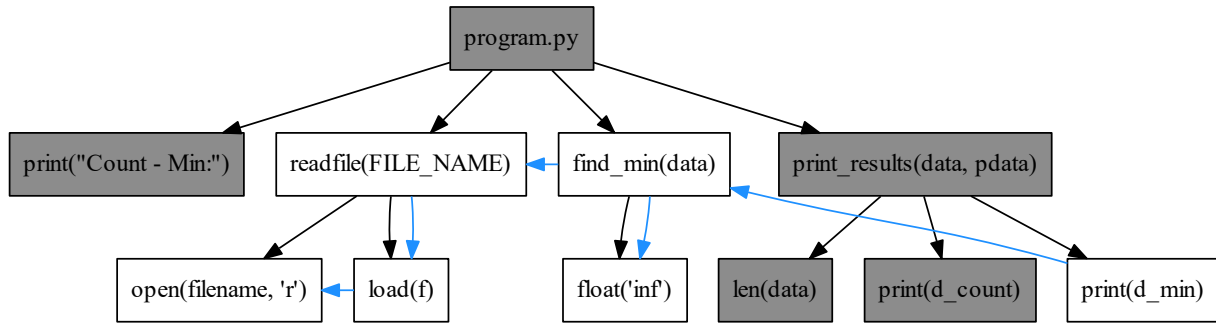


Figure 3.3: The provenance enhanced execution tree

correct node is in the transitive closure of the latter node, informing an incorrect output related to the former node does not help in finding the defect.

## 3.4 Navigation Module

The navigation module of DebugProv employs the standard navigation strategies of algorithmic debugging, discussed in Chapter 2. However, instead of using the plain execution tree, they use the provenance enhanced execution tree, which was already pruned by the enhancement module.

Currently, DebugProv allows developers to choose any of the four navigation strategies presented in Table 2.1. For instance, similarly to the example discussed in Chapter 2, suppose that DebugProv is using the Top Down navigation strategy, but now over the provenance enhanced execution tree, shown in Figure 3.3. Instead of evaluating the root and the *print* nodes, which were pruned by the enhancement module, DebugProv first evaluates the *readfile* node. The developer is informed about the inputs and outputs of this node and answers that the node is valid. Consequently, DebugProv classifies the node *readfile* and all of its subtree nodes as valid, removing them from the search space.

Afterward, DebugProv asks the developers about the *find\_min* node and the developer answers that the computation is invalid. Hence, DebugProv defines *find\_min* as invalid, and all nodes that are not descendants of *find\_min* are marked as valid and are removed from the search space. Finally, DebugProv asks about the *float* node, and the developer indicates that its behavior is correct. Thus, DebugProv marks this node as valid and finishes the debugging session, showing that the defective node is *find\_min*.

For this specific navigation strategy, the number of questions asked to the developer dropped from 5 to 3, which consists of a 40% gain. Table 3.1 summarizes the improvements introduced by DebugProv in our guiding example, for each one of the navigation strategies.

Although DebugProv currently implements just the four classic navigation strategies, it was specifically designed to accommodate additional navigation strategies in the future easily.

Table 3.1: Improvements obtained by DebugProv in the guiding example

Nav. Strategy	Steps in Figure 3.3	# Quest.	Reduction
<i>Single Stepping</i>	open, load, readfile, float, find_min	5	16%
<i>Top Down</i>	readfile, find_min, float	3	40%
<i>Heaviest First</i>	readfile, find_min, float	3	40%
<i>Divide and Query</i>	readfile, find_min, float	3	25%

## 3.5 Implementation

DebugProv is implemented in Python 3.7 and is able to run algorithmic debugging sessions on Python programs (compliant to version 3.7). We adopt noWorkflow [31, 36, 37] for capturing the provenance data required by our capturing module. We chose this tool because it not only captures the provenance of Python script, as required by our provenance enhancement module, but also captures enough execution data in the provenance for the generation of execution trees by our approach.

DebugProv invokes noWorkflow to execute the defective program, capture all the related provenance data, and store it in a directory *.noworkflow*. Then, the capturing module of DebugProv consumes the provenance data from an SQLite database stored in this directory and produces an execution tree in a fully automated step. After capturing the provenance data and building the executing tree, the capturing module stores it and sends it to the provenance enhancement module.

The collection process implemented in noWorkFlow includes dynamic program slicing [8], which captures the steps used by a program to generate the values used by its variables, function results, or outputs.

In a perfect scenario, the provenance data collected by the *capturing module* would always exactly represent the nodes that were responsible for producing an incorrect output. In other words, if a node does not contribute to producing an incorrect output, it would not be on the provenance graph, and if a node contributed for producing an incorrect output, it would be on the provenance graph.

In our implementation, we are using noWorkFlow in version 2.0 alpha. We decided to

use this version of the tool because it is the most recent and active version, with constant maintenance and corrections. Even so, every provenance collection and storage tool has its own limitations, and noWorkflow (especially in an alpha version) is not free from limitations.

We identified limitations in three situations. First, when a function influences the values of members of an object, but not the object itself, *e.g.*, a function influences an element of a list, but not the list itself. In this case, noWorkflow does not store the dependency between the list and the function. Second, when noWorkflow is not able to capture the existing dependency relation between a function and its caller. Third, when the absence of a computation influenced the program outcome, and noWorkflow can not capture the dependency relation between an absent computation and the produced outcome. In these cases, the provenance data captured by noWorkflow was not representing exactly what we expected: some nodes that were supposed to be in the provenance graph were not captured.

In order to provide a workaround to these limitations of noWorkflow 2.0 alpha, we implemented a *fallback* technique. The fallback technique is our way of handling the imperfections of the captured provenance. It is applied when an execution tree is enhanced with provenance and traversed by a navigation strategy, but the defective node is not found. Then, we assume that the provenance data did not capture the dependency on the defective node, and the search continues by evaluating the nodes that were not on the provenance slice.

The *fallback* technique only needs to be executed when a node that was supposed to be on the provenance graph is not on it. The *fallback* technique can increase the number of questions to locate the defective node. We expect that as the provenance collection and storage tools improve and get more precision, the *fallback* technique will be less necessary, consequently leading to fewer questions during debugging sessions.

## 3.6 Final Remarks

In this chapter, we introduced DebugProv, our algorithmic debugging approach for Python programs. Besides the *capturing module* and the *navigation module*, we introduced the *enhancement module*, which is responsible for implementing the provenance enhancement, our technique to remove from the execution tree the nodes that did not influence the incorrect output, reducing the number of questions and consequently the necessary effort

to locate the defective node. We also explained our implementation and the usage of noWorkflow in the *capturing module* to collect provenance data from Python programs.

In the next chapter, we present our approach for evaluating DebugProv and the provenance enhancement technique. We describe our experimental study and discuss the results.

# Chapter 4

## Evaluation

### 4.1 Introduction

In this chapter, we present our approach to evaluate DebugProv and the provenance enhancement technique. In this evaluation, we investigate the impact of the provenance enhancement technique in the number of steps (or questions) to locate the defective node in the execution tree.

We performed a quantitative evaluation of DebugProv to assess its effectiveness in comparison to the classic algorithmic debugging technique. Our evaluation is concerned with the following research questions:

- **RQ1:** *Does the provenance enhancement reduce the number of questions in algorithmic debugging?*
- **RQ2:** *How provenance enhancement improves each navigation strategy?*

Our evaluation is structured as follows: (i) we first selected a set of Python programs to form the corpus; then (ii) we generated mutant versions of these programs by inserting different types of defects; finally, (iii) we executed automated algorithmic debugging sessions with DebugProv and with the classic algorithmic debugging technique over the defective programs. For automating the algorithmic debugging sessions, we simulated the answer of users in algorithmic debugging sessions by generating oracles that store the validity of each node in the execution tree. To answer these research questions, we measured the number of questions asked to the developer to detect each defect and contrasted the performance of both approaches (with and without provenance enhancement). We explain our experiment in more detail in the following subsections.



## 4.2 Materials and Methods

Our evaluation process is described by the following workflow:

1. Select a set of Python programs
2. Run all selected programs, storing the outputs
3. Generate mutants of the selected programs
4. Run all mutants, storing the outputs
5. Generate oracles
6. Run automated algorithmic debugging sessions

In the **first step**, we selected a set of 25 Python programs from GitHub repositories to form the corpus of our experiment. We selected the repositories by searching on Github for "algorithms" and filtering by the Python language. We used three criteria to select each program: (i) the program must be exclusively written in the Python language (compliant to version 3.7), (ii) the program must be syntactically correct, and (iii) the program must produce an output. The selected programs and the respective repositories are presented in Table 4.1.

In the **second step**, we executed each program using Python 3.7 and stored their respective outputs. We assume that these outputs represent the correct executions of the programs, and we use them as baselines for the oracle generation (step 5).

In the **third step**, we generated mutants [9] for each of the selected programs. Mutants are variants of a program with the introduction of some logical change in the source code. This technique is originally used in the software testing area to assess the quality of test suites. For instance, a single mutation may be the replacement of a "==" (equal sign) to a "!=" (not equal sign), or the replacement of a "<" (less sign) to a ">=" (greater or equal sign). As the number of possible transformation is large, a single program can produce several mutants, each one containing exactly one change. We used *universalmutator* [14], which is a multi-language regex-based mutant generator, to produce mutants for our selected programs. The complete set of mutation operators supported by *universalmutator* is available in the tool repository. In this experiment, we used the python

Table 4.1: Selected Programs

#	Program Name	Repository on GitHub	Application Domain	LOCs
01	Compression Analysis	TheAlgorithms/Python	Image Processing	40
02	Bisection	TheAlgorithms/Python	Arithmetic	39
03	Intersection	TheAlgorithms/Python	Arithmetic	21
04	LU Decomposition	TheAlgorithms/Python	Arithmetic	31
05	Newton Method	TheAlgorithms/Python	Arithmetic	18
06	Basic Binary Tree	TheAlgorithms/Python	Data Structures	47
07	Dijkstra Algorithm	TheAlgorithms/Python	Graphs	212
08	Caesar Cipher	TheAlgorithms/Python	Cryptography	73
09	Brute Force Caesar Cipher	TheAlgorithms/Python	Cryptography	56
10	Basic Maths	TheAlgorithms/Python	Arithmetic	74
11	Mergesort	TheAlgorithms/Python	Sorting	69
12	Decision Tree	TheAlgorithms/Python	Machine Learning	143
13	Math Parser	keon/algorithms	Arithmetic	143
14	Merge Interval	keon/algorithms	Data Structures	83
15	Binary Search	keon/algorithms	Searches	36
16	Permute	keon/algorithms	Combinatorics	57
17	LCS	haikentcode/top10algoritms	Data Comparison	28
18	Catalan	haikentcode/top10algoritms	Combinatorics	15
19	Bubblesort	haikentcode/top10algoritms	Sorting	28
20	Quicksort	haikentcode/top10algoritms	Sorting	47
21	Heapsort	mingrammer/sorting	Sorting	80
22	Generate Parenthesis	marcosfede/algorithms	Combinatorics	34
23	Knn	harrypotter0/algorithms-in-python	Machine Learning	95
24	String Permutation	harrypotter0/algorithms-in-python	Combinatorics	35
25	Linear Regression	llSourcecell/linear_regression_demo	Machine Learning	21

operators<sup>1</sup> and the universal operators<sup>2</sup>. We generated a total of 6,197 mutants in this step of the workflow, which are the subjects of our experiment. Since *universalmutator* could not generate mutants for two programs (#01 - Compression Analysis and #22 - Generate Parenthesis), they were removed from our corpus.

In the **fourth step**, we ran all 6,127 mutants generated in step 3. We could observe three distinct situations for each mutant execution: (i) the mutant ran successfully, and thus we stored the output of the execution (4,758 cases); (ii) the mutant entered in an infinite loop and was discarded after waiting for 90 seconds<sup>3</sup> (163 cases); and (iii) the mutant did not even start to run due to invalid syntax or broken dependencies, generating an error before execution (1,276 cases). In this case, we also stored the information related to the error. Subsequently, we removed mutants that did not produce outputs (1,439 from cases ii and iii) and removed the mutants that produced the same output of the original

<sup>1</sup><https://github.com/agroce/universalmutator/blob/master/universalmutator/static/python.rules>

<sup>2</sup><https://github.com/agroce/universalmutator/blob/master/universalmutator/static/universal.rules>

<sup>3</sup>As discussed before, the current version of DebugProv does not deal with silent defects.

program (1,794 cases). This latter case indicates that the code transformation introduced during the generation of the mutant for some reason did not lead to a failure. By the end of this step, we only have mutants that ran and finished the execution with an output that is different from the output of the original program, leaving us with 2,964 mutants.

In the **fifth step**, we generated the oracle for each mutant. To do so, we first calculated the diffs between the original programs and the mutants. These diffs precisely identify the lines where a defect was introduced during the mutant generation (step 3). We used *difflib* [40], a Python module that compares files, to identify the diffs. Then, we ran the mutants using the capturing module of DebugProv to capture and store execution data about the mutants. Finally, we located the line (or lines) of code that the mutation process changed and identified the respective node in the execution tree. The generated oracle indicates this node and its ancestors as invalid and all other nodes as valid.

We removed out mutants that have more than one defective node in the execution tree (2,506 cases). Please remember that each node in the execution tree represents an activation, and activations are associated with code components. Since a code component may produce multiple activations (e.g., a recursive function that was called multiple times), the execution tree could contain more than one invalid node. We opted to remove theses imprecise situations as they could introduce bias to the results of the experiment. Consequently, after this step, we end up with 458 mutants. Table 4.2 summarizes the characterization of the mutants. In this step, eight of the selected programs were removed because they had no remaining mutants. The removed programs in this step are #8 - Caesar Cipher, #9 - Brute Force Caesar Cipher, #10 - Basic Maths, #12 - Decision Tree, #13 - Math Parser, #23 - Knn, #24 - String Permutation, and #25 - Linear Regression.

Table 4.2: Mutants Characterization

Description	Step	# of mutants
Generated Mutants	3	6,197
Mutants that do not start executing	4	1,276
Mutants that do not finish executing	4	163
Mutants without failure	4	1,794
Mutants without a unique oracle	5	2,506
Used Mutants	6	458

Finally, in the **sixth step**, we ran the algorithmic debugging sessions over each of the 458 mutants. DebugProv was originally designed to perform semi-automated debugging sessions with a developer being the oracle – the developer must answer the questions

about the validity of the execution tree nodes. During our evaluation, we adapted DebugProv to perform automated debugging sessions, by reading the oracle that contains the information about the validity of nodes (see step 5). We also run in this step the classic algorithmic debugging technique, without provenance enhancement. After finishing all automated debugging sessions, we analyzed the output data.

## 4.3 Results and Discussion

### 4.3.1 Does the provenance enhancement reduce the number of questions in algorithmic debugging (RQ1)?

To answer this question, we looked at (i) the number of steps to locate the defective node using provenance enhancement (i.e., DebugProv) and (ii) the number of steps to locate the defective node without provenance enhancement (i.e., classic algorithmic debugging), regardless of the navigation strategy. Thus, we pose the following hypotheses, which are subject to statistical tests:

- $H_0$ : The number of questions asked during debugging sessions is **the same** for execution trees with and without provenance enhancement.
- $H_1$ : The number of questions asked during debugging sessions is **different** for execution trees with and without provenance enhancement.

We applied the Shapiro-Wilk test [44] to check whether our samples followed a normal distribution. Both samples (with and without provenance enhancement) do not follow a normal distribution (p-value lower than  $2.2 \times 10^{-16}$ ). Therefore, we used the Wilcoxon Signed-Rank test [49], a non-parametric test to compare two paired samples. The resulting p-value was  $4.815 \times 10^{-16}$ , rejecting the null hypothesis ( $H_0$ ) and indicating that there is indeed a difference between the samples. A visual inspection of the boxplots in Figure 4.1 indicates that the provenance enhancement reduced the number of questions asked during the debugging sessions.

We also applied Cliff's Delta (for paired samples) to calculate the effect size between the samples. Cliff's Delta is a non-parametric test that allows quantifying the magnitude of the difference between two samples that do not meet the normality assumptions. The results have an effect size of 0.36, which is classified as *medium*<sup>4</sup> [41]. In addition to the

---

<sup>4</sup>The magnitude is determined by applying the thresholds presented by Romano et al. [41], *i.e.*  $negligible < 0.147 \leq small < 0.33 \leq medium < 0.474 \leq large$ .

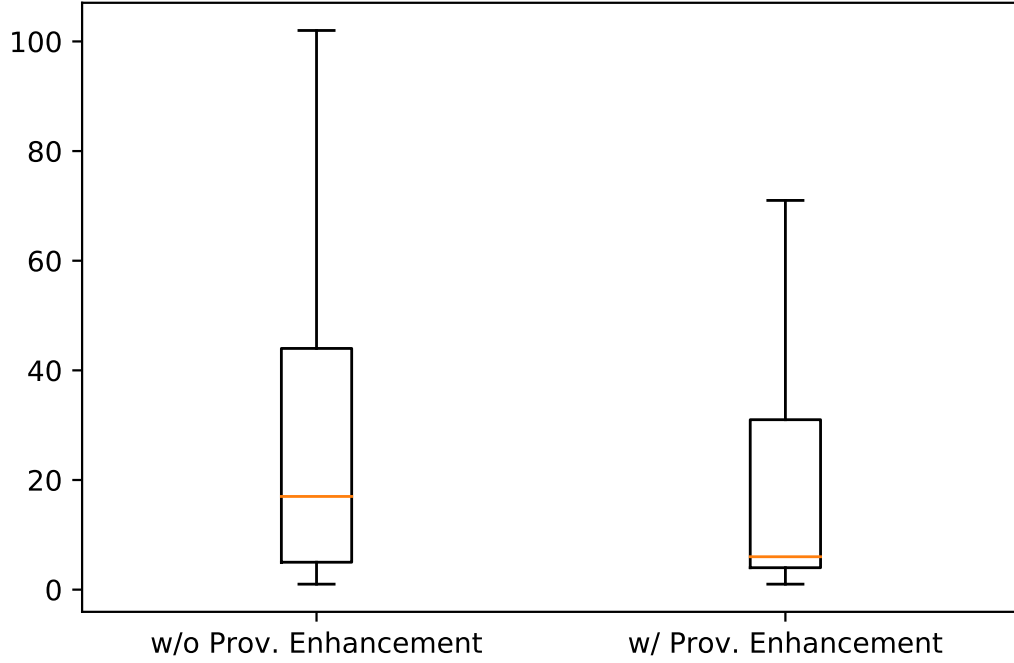


Figure 4.1: Boxplot of the number of questions required to locate the defective node without provenance enhancement (left) and with provenance enhancement(right). Outliers removed.

effect size, we also calculated the proportional reduction in the number of questions by subtracting the number of questions with provenance from the number of questions without provenance and dividing this result by the number of questions without provenance. We observed a reduction of 25.26% in the number of questions, on average.

Finally, we analyzed each program individually. In this analysis, we computed the number of questions for each mutant of that program, both for the execution tree with and without provenance. In Table 4.3, we present the results of this analysis. We can observe that the decrease in the number of questions can vary from program to program: in some cases, like the 02 - Bisection program, the provenance enhancement practically does not change the number of questions, but in other cases, such as the *LU Decomposition* program, we were able to observe an impressive reduction.

**RQ1:** *Does the provenance enhancement reduce the number of questions in algorithmic debugging?*

**Answer:** We could observe a statistically significant reduction in the number of questions asked during debugging sessions when the execution tree is enhanced with provenance, with medium effect size. The average number of questions dropped from 41.05 to 30.68, while the median number of questions dropped from 17.0 to 6.0. The maximum number of questions dropped from 102 to 71. We also observed that the reduction varies from program to program.

Table 4.3: Individual analysis of provenance enhancement by selected program.

#	Program	Muts.	Questions w/o Prov. Enhc.	Questions w/ Prov. Enhc.	Reduct.	Fallback
02	Bisection	71	24,564	24,097	1.9%	9
03	Intersection	40	6,699	5,516	17.66%	10
04	LU Decomposition	55	4,743	990	79.13	0
05	Newton Method	15	419	321	23.39%	0
06	Basic Binary Tree	5	296	254	14.19%	1
07	Dijkstra Algorithm	92	24,430	15,170	37.9%	49
11	Mergesort	3	36	24	33.33%	0
14	Merge Intervals	28	850	276	67.53%	0
15	Binary Search	17	170	112	34.12%	0
16	Permute	2	99	83	16.16%	0
17	LCS	8	112	96	14.29%	0
18	Catalan	5	52	40	23.08%	0
19	Bubblesort	51	510	710	-40%	51
20	Quicksort	10	434	471	-8.53%	10
21	Heapsort	56	11,805	8,056	31.76%	0
	Total	458	75,219	56,220	25.26%	130

It is important to notice in Table 4.3 that the usage of the *fallback* technique can produce an increase in the number of questions, comparing to the traditional algorithmic debugging technique. In the programs *Bubblesort* and *Quicksort*, the *fallback* technique was applied in every session: 51 sessions in *Bubblesort* and 10 sessions in *Quicksort*. The high usage of the *fallback* technique leads to a total increment in the number of questions.

### 4.3.2 How provenance enhancement improves each navigation strategy (RQ2)?

To answer **RQ2**, we computed the number of questions to locate the defective node by navigation strategy. Each one of the four navigation strategies was executed over

the execution tree with and without provenance enhancement. Therefore, we have eight treatments for this analysis.

Similar to RQ1, we first checked normality with the Shapiro-Wilk test and obtained p-values lower than  $2.2 \times 10^{-16}$  for all samples, which indicates that none of them followed a normal distribution. Therefore, we used the Wilcoxon Signed-Rank test (for paired samples) for comparing the number of required questions to locate the defective node for each permutation of navigation strategy (*Single Stepping*, *Top Down*, *Heaviest First*, and *Divide and Query*) and execution tree (with or without provenance enhancement). The obtained p-values for the Wilcoxon Signed-Rank test were lower than  $2.2 \times 10^{-16}$  for *Single Stepping*, *Top Down*, *Heaviest First*, while for *Divide and Query* it was  $2.103 \times 10^{-8}$ . The results indicate that for every navigation strategy, there is a statistically difference between the samples (with or without provenance enhancement). We also used Cliff's delta (for paired samples) to calculate the effect size.

Table 4.4 shows the reduction of the number of questions for each navigation strategy. We can observe that the effect size was large for the *Top Down* strategy, followed by *Heaviest First* and *Single Stepping*, with medium effect size. The *Divide and Query* strategy presented the smallest effect size among the evaluated strategies. Moreover, *Single Stepping* was the navigation strategy with the most significant proportional reduction, followed by *Top Down*. However, *Heaviest First* is the navigation strategy with the best performance in terms of the total number of questions, both for execution trees with and without provenance enhancement. Surprisingly, *Divide and Query* is just the third best navigation strategy in terms of proportional reduction of the number of questions. We believe that the pruning of nodes by the provenance reduced the number of descendants in the biggest functions, which are also the functions that are most likely to have defects. Thus, the provenance enhancement did not reduce as many nodes in the *Divide and Query* strategy, as it did in the other strategies.

Table 4.4: Navigation strategies performance over trees with and without provenance enhancement.

Navigation Strategy	Questions w/o Prov.	Questions w/ Prov.	Reduction	Effect Size
<i>Single Stepping</i>	36,498	25,181	31.01%	0.347 (Medium)
<i>Top Down</i>	13,787	10,252	25.64%	0.476 (Large)
<i>Heaviest First</i>	11,683	9,170	21.51%	0.454 (Medium)
<i>Divide and Query</i>	13,251	11,617	12.33%	0.163 (Small)

**RQ2:** *How provenance enhancement improves each navigation strategy?*

**Answer:** We could observe a reduction in the number of questions for all navigation strategies when the execution tree is enhanced with provenance. The effect size ranges from 0.163 to 0.454, and the reduction ranges from 12.33% to 31.01%. The navigation strategy with the greatest reduction is *Single Stepping*, and the navigation with the highest effect size between the number of questions with and without provenance enhancement is *Top Down*. Nevertheless, *Heaviest First* is the strategy with the smallest total number of questions when the provenance enhancement is applied.

In the following subsections we analyze and discuss the **RQ2** for each navigation strategy evaluated in this work (*Single Stepping*, *Top Down*, *Heaviest First* and *Divide and Query*).

#### 4.3.2.1 How provenance enhancement improves *Single Stepping* strategy

We can observe that in the *Single Stepping* navigation strategy, the total number of questions dropped from 36,498 to 25,181, a reduction of 31.01%. Among the four navigation strategies evaluated in this work, *Single Stepping* is the one with the most significant percentage reduction in the number of questions.

We can also observe that even though the reduction is significant, *Single Stepping* is by far the strategy that needed more questions to locate the defective node. While the *Single Stepping* with provenance enhancement asked a total of 25,181 questions, no other strategy with provenance enhancement asked more than 12,000 questions.

We conclude that even though the reduction of the total number of questions is numerous in the *Single Stepping* navigation strategy, the total number of questions was significantly smaller in *Top Down*, *Heaviest First* and *Divide and Query*.

In Figure 4.2 we can see a boxplot representing the results of the *Single Stepping* experiment: the distribution of the number of questions to locate the defective node without provenance enhancement is presented on the left, and the number of questions to locate the defective node with provenance enhancement is presented on the right. The average number of questions dropped from 79.7 to 54.9, while the median number of questions dropped from 22 to 11.



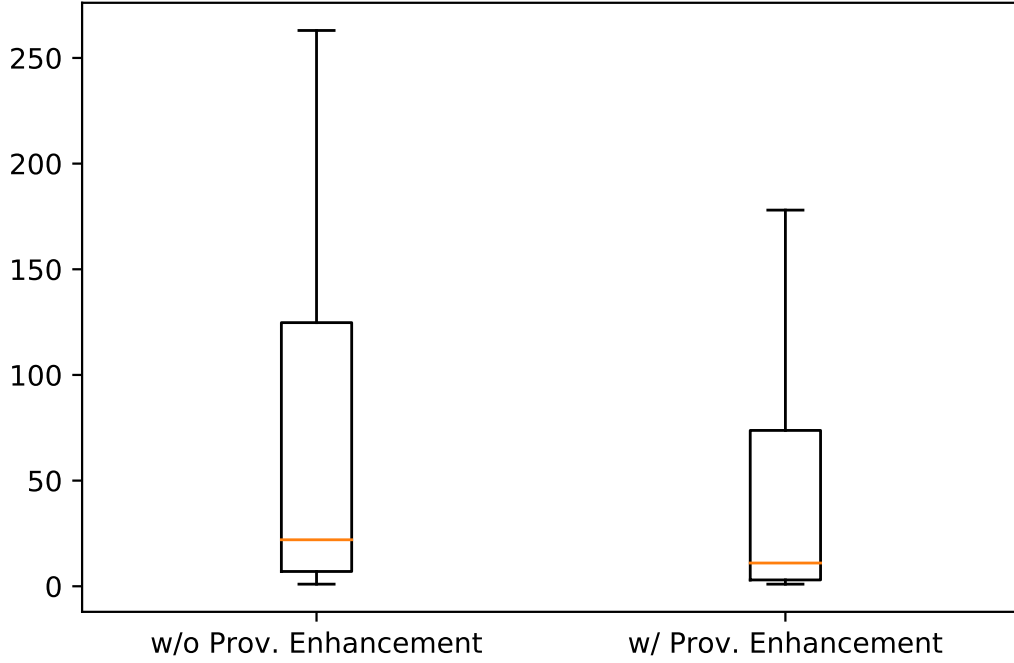


Figure 4.2: Boxplot of the number of questions required to locate the defective node without provenance enhancement (left) and with provenance enhancement(right) in the *Single Stepping* navigation strategy. Outliers removed.

#### 4.3.2.2 How provenance enhancement improves *Top Down* strategy

In the *Top Down* navigation strategy, we observed that the total number of questions dropped from 13,787 to 10,252, which represents a reduction of 25.64%. Comparing *Top Down* to the other navigation strategies evaluated in this work, we can notice that it performs the second greatest percentage reduction.

It is important to notice that we analyzed the effect size of the reduction of every navigation strategy. *Top Down* performed the greatest effect size, with value 0.476, which is classified as a large magnitude [41].

The provenance enhancement was able to reduce significantly the number of questions in the *Top Down* strategy. Even though the percentage reduction in *Single Stepping* strategy is bigger, the total number of questions with provenance enhancement performed by *Top Down* (10,252) is smaller than the total number of questions with provenance enhancement performed by the *Single Stepping* strategy (25,181).

The results of the *Top Down* analysis are presented in Figure 4.3. The distribution

of the number of questions to locate the defective node without provenance enhancement is presented on the left, and the number of questions to locate the defective node with provenance enhancement is presented on the right. The average number of questions dropped from 30.1% to 22.3%, while the median number of questions dropped from 20 to 6.

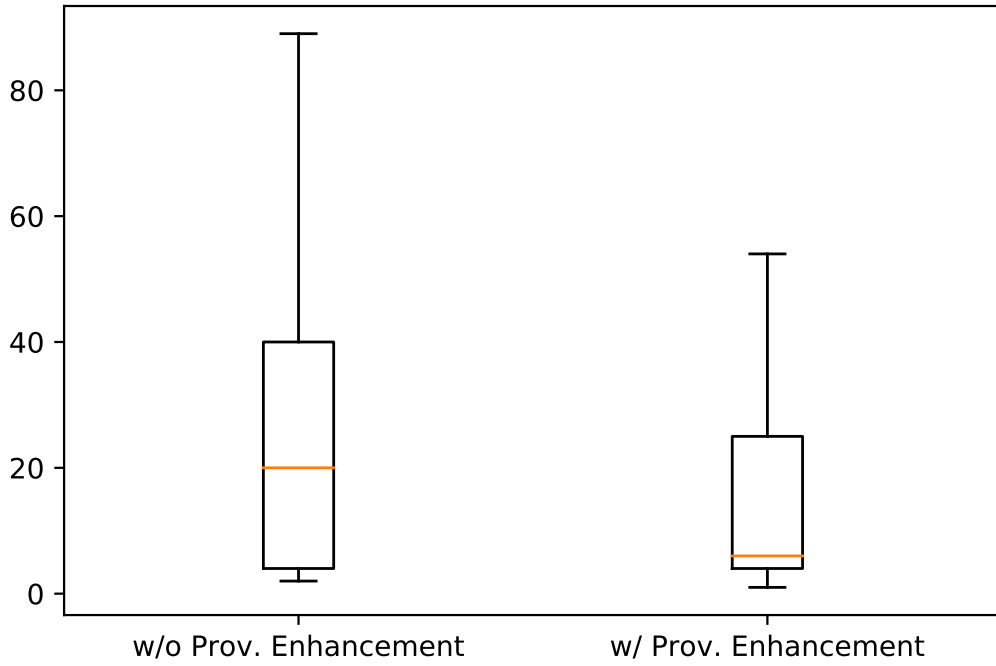


Figure 4.3: Boxplot of the number of questions required to locate the defective node without provenance enhancement (left) and with provenance enhancement(right) in the *Top Down* navigation strategy. Outliers removed.

#### 4.3.2.3 How provenance enhancement improves *Heaviest First* strategy

The total number of questions dropped from 11,683 to 9,170, which represents a reduction of 21.51% in the *Heaviest First* navigation strategy. We can notice that the percentage reduction of this strategy is below the reductions performed by *Single Stepping* and *Top Down*.

We noticed that among the navigation strategies presented in this work, *Heaviest First* is the strategy with the smallest total number of questions. Not only in the sessions without provenance enhancement, but also in the sessions with provenance enhancement.

The effect size of the reduction of *Heaviest First* is 0.454, which is classified as a

medium magnitude. Even though the *Heaviest First* is not the strategy that performed the biggest percentage reduction in the number of questions, it is the one that needed the smallest number of questions to locate the defective node.

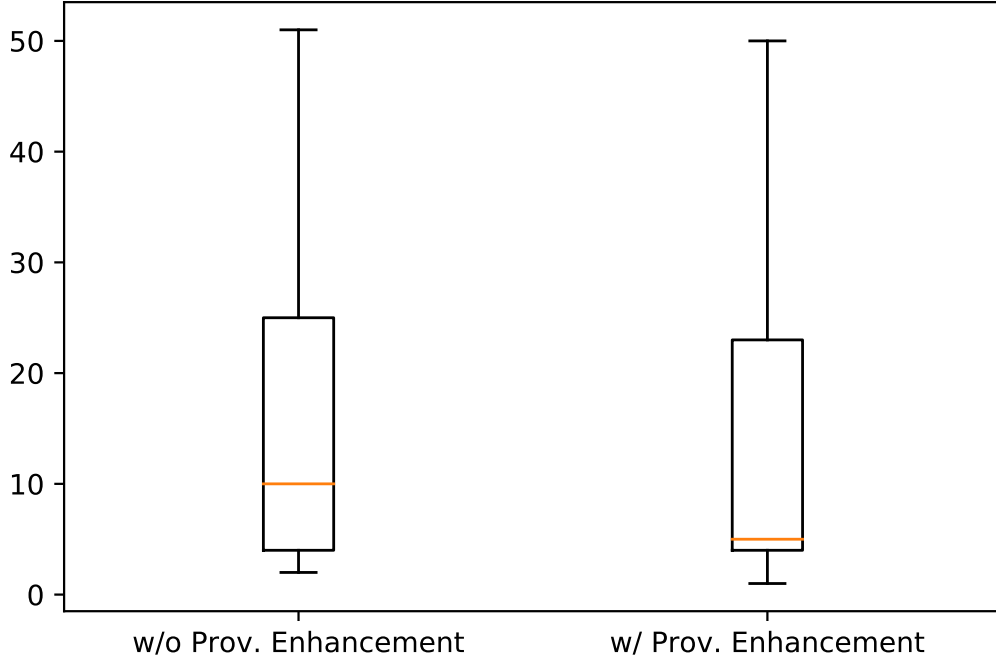


Figure 4.4: Boxplot of the number of questions required to locate the defective node without provenance enhancement (left) and with provenance enhancement(right) in the *Heaviest First* navigation strategy. Outliers removed.

The results of the *Heaviest First* analysis are presented in Figure 4.4. Even though the maximum value does not drop, the average number of questions dropped from 25.51 to 20.02 and the median number of questions dropped from 10 to 5.

#### 4.3.2.4 How provenance enhancement improves *Divide and Query* strategy

In the *Divide and Query* navigation strategy, we observe that the total number of questions dropped from 13,251 to 10,617, which represents a reduction of 12.33%. Comparing *Divide and Query* to the other navigation strategies evaluated in this work, we can notice that it performs the smallest percentage reduction.

We analyzed the effect size of the reduction for the *Divide and Query* every navigation strategy. The effect size value is 0.163, which is classified as a small magnitude [41]. Among the navigation strategies evaluated in this work, *Divide and Query* performed the

smallest effect size.

The provenance enhancement was able to reduce the number of questions in *Divide and Query* strategy. Even though the total number of questions with provenance enhancement is not big (11,617), the percentage reduction of this strategy presented the smallest value, and the effect size is also the smallest value among the strategies evaluated in this work.

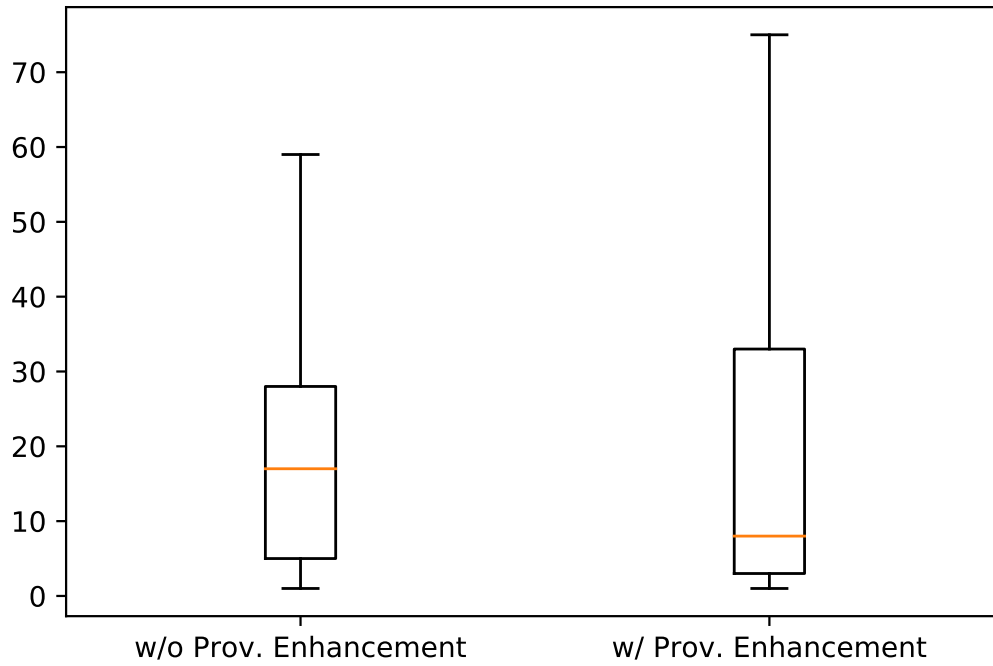


Figure 4.5: Boxplot of the number of questions required to locate the defective node without provenance enhancement (left) and with provenance enhancement(right) in the *Divide and Query* strategy. Outliers removed.

The results of the *Divide and Query* analysis are presented in Figure 4.5. Even though the maximum value increased with the provenance enhancement, the average number of questions dropped from 28.93 to 25.36, and the median number of questions dropped from 17 to 8.

## 4.4 Threats to Validity

Even though we have carefully conducted the experiment design and execution, as in any experimental study, our work is subject to some threats to validity.

**Construct.** Our dependent variable in all research questions is the number of ques-

tions asked to the user during a debugging session. However, different questions may have different difficulties. Consequently, debugging sessions with fewer questions are not necessarily faster than the ones with more questions.

**Construct.** Our approach requires the developer to indicate the node that manifested the defect. We attempted to simulate this operation in our oracles by comparing the print statements of the original program with the mutant program and choosing the most likely to be the one that produced an invalid output. If the program has many print statements, the oracle may have indicated the wrong node. It could cause the removal of the defective node from the execution tree, and the impossibility of finding it with fewer steps.

**Construct.** We implemented the *fallback* technique (described in Section 3.5) to handle imperfections in the provenance data. The usage of the *fallback* technique can lead to cases where the provenance enhancement provides a sub-optimal number of questions. Consequently, the results presented in this experiment should be seen as a lower bound of the provenance enhancement technique. We expect that the provenance enhancement can reduce even more the number of questions in a scenario where the provenance does not present imperfections.

**Internal.** We used program mutation to simulate defective programs. The tool we use to produce mutants, *universalmutator* [14], implements a set of operators for generating mutants. Even though a mutant generated by *universalmutator* is a defective program, we cannot assume that defects naturally produced by programmers are similar to those artificially generated by the *universalmutator* tool.

**External.** We selected implementations of known algorithms in Python. However, we cannot assume that DebugProv will reach the same effectiveness when executed over other Python programs.

**External.** We used a script to generate oracles for running automated algorithmic debugging sessions. Our script was not able to generate an oracle for every mutant – when the mutated code component was associated with multiple nodes in the execution tree, we were not able to automatically distinguish which nodes exercised the mutant lines. Consequently, we could not generate the oracle in these situations. We removed several mutants from the evaluation due to this limitation. Therefore, we may have removed mutants that shared the same behavior, compromising somehow the generalizability of our results.

## 4.5 Final Remarks

In this chapter, we presented our experimental approach to evaluate DebugProv and the provenance enhancement technique. We artificially generated defective programs and executed algorithmic debugging sessions with DebugProv and with a traditional algorithmic debugging approach, measuring the number of questions to locate the defective node. In our experiment, DebugProv was able to reduce the number of questions: The average number of questions dropped from 41.05 to 30.68, a reduction of 25.26%.

# Chapter 5

## Conclusion

### 5.1 Contributions

In this work, we presented DebugProv, an algorithmic debugging tool for Python. With DebugProv, we introduce the provenance enhancement, a novel technique to remove irrelevant nodes from the execution tree, reducing the number of questions to locate the defective node. DebugProv is the first tool that uses provenance to enhance algorithmic debugging over a dynamically-typed and interpreted imperative language. Moreover, the provenance captured by DebugProv is based on dynamic program slicing instead of static program slicing, thus increasing the precision of the results.

We evaluated our approach through a quantitative study with 15 Python programs. We artificially inserted defects into each program, generating hundreds of mutants that were used during automated algorithmic debugging sessions. Our study evaluated the effects of provenance enhancement in algorithmic debugging sessions by contrasting DebugProv with traditional algorithmic debugging. We used four common navigation strategies in our evaluation. Our results showed that provenance enhancement reduced the number of algorithmic debugging questions by 25.26%, on average.

### 5.2 Limitations

We identified some limitations on our approach, DebugProv, and our technique, provenance enhancement. A limitation present in several algorithmic debuggers, including DebugProv, is only to detect one defect per debugging session. It is possible to perform an algorithmic debugging session in a program with multiple defects, but each session is going to indicate a defective node until all defects are eliminated.

In order to run an algorithmic debugging session with DebugProv, the defective program must produce an incorrect output. Latent defects cannot be detected. Since the navigation phase begins after the end of the program execution, DebugProv is not able to run algorithmic debugging sessions on programs with infinite loops. Consequently, DebugProv does not support programs that run but do not produce any output.

Other limitations were presented due to the infrastructure used in the *capturing module*, presented in section 3.2. When the *capturing module* runs a defective program, the capture of the execution data produces an overhead that slows down the program execution. The provenance data captured by the *capturing module* can present imperfections: cases where the provenance data was not representing exactly what we expected, that can lead to false negatives (computations that influenced an incorrect output, but the dependency relation was not captured). To deal with these imperfections, we implemented a *fallback* technique. The *capturing module* is also not able to capture execution data from multi-thread programs. Consequently, DebugProv is not able to debug them.

## 5.3 Future Work

We have identified several possibilities for future work. Currently, DebugProv runs the debugging session based on a single faulty execution of a program. We intend to take advantage of multiple executions to refine even further the pruning process with the aid of a statistical mechanism. Comparing the differences between successful and unsuccessful executions would allow us to propose another pruning technique that reduces, even more, the number of questions to locate the defective node. For example, assume two functions in a particular program: *function A* and *function B*. Moreover, all the executions that activated *function A* without activating *function B* produced a correct outcome. However, 82% of the executions that activated *function B* produced an incorrect output. In this example, we have an initial insight that *function A* is probably not defective, while *function B* is probably defective. We intend to bring to DebugProv statistical mechanisms to explore the correlations between the executed lines and the correctness of the outcome. Additionally, we intend to use why-not provenance in DebugProv to detect silent defects [6]. These defects manifest on infinite loops or in the absence of outputs.

We intend to implement a storage system to save in a database each question asked in a debugging session and the corresponding answer given by the programmer. This database of questions and answers can work as a *cache* to prevent DebugProv to ask



the same question multiple times. We expect that this technique will reduce the number of questions, specially in two cases: (i) when a program contains multiple defects and multiple algorithmic debugging sessions are necessary, and (ii) when a program contains multiple activations of the same routine with the same inputs and outputs.

We also intend to allow the user to inspect different types of inputs and outputs of a function (or method) to evaluate its correctness. Currently, the user can inspect the arguments and returns, but functions can consume and produce data beyond the arguments and returns: through external files, database connections, or HTTP connections. We aim at letting the user inspect these different types of inputs and outputs. Also, we intend to offer visualization techniques for complex structures so that the user can read and interpret different types of structures manipulated by the function.

The current implementation of DebugProv does not provide a graphical user interface. Another future work involves the development of an intuitive and user-friendly graphical user interface for DebugProv, which could facilitate the adoption and usage of the tool. Currently, the developer must choose the navigation strategy to traverse the execution tree. Another future work involves the development of techniques to suggest a navigation strategy to the developer based on attributes of the program or the execution tree.

We only performed a quantitative evaluation in this dissertation. We expect to do some other quantitative analyses with our experiment data. We intend to investigate the correlations between the complexity of the defective program (LOCs, cyclomatic complexity [29]) and the reduction in the number of questions. We also intend to investigate the correlation between attributes of the execution tree and the performance of the navigation strategy. Moreover, we also intend to perform a qualitative investigation of the results in order to answer questions such as "Why the provenance enhancement technique performed better in some programs than in others?".

We also envision two future works to integrate software testing techniques with DebugProv. The first one is to use existing test cases and test oracles of the defective program to provide answers on the correctness of the nodes, reducing the number of questions asked to the programmer. The other one is to use the answers provided by the programmer during a debugging session to automatically generate test cases for the valid nodes of the defective program.

Lastly, we also plan to use execution data to locate defects in a more fine-grain. Currently, the grain of DebugProv is a function. After a debugging session with DebugProv, the developer still needs to locate the defect inside the function. Since we stored data

---

about all the execution, we can use it to help the developer finding the defective line(s) of code.

# References

- [1] AV-RON, E. Top-down diagnosis of prolog programs. Master's thesis, Weizmann Institute of Science, Rehovot, Israel, 1984.
- [2] BINKS, D. F. J. *Declarative Debugging in Gödel*. Tese de Doutorado, Citeseer, 1995.
- [3] CABALLERO, R.; HERMANN, C.; KUCHEN, H. Algorithmic debugging of java programs. *ENTCS 177* (2007), 75–89.
- [4] CABALLERO, R.; MARTIN-MARTIN, E.; RIESCO, A.; TAMARIT, S. Edd: A declarative debugger for sequential erlang programs. In *TACAS* (Berlin, Heidelberg, 2014), Springer Berlin Heidelberg, pp. 581–586.
- [5] CABALLERO, R.; RIESCO, A.; SILVA, J. A survey of algorithmic debugging. *CSUR* 50, 4 (2017), 60.
- [6] CHAPMAN, A.; JAGADISH, H. Why not? In *SIGMOD* (Providence, RI, 2009), ACM, pp. 523–534.
- [7] CHEDA, D.; SILVA, J. State of the practice in algorithmic debugging. *Electronic Notes in Theoretical Computer Science* 246 (2009), 55–70.
- [8] CHEN, Z.; CHEN, L.; ZHOU, Y.; XU, Z.; CHU, W. C.; XU, B. Dynamic slicing of python programs. In *COMPSAC* (Vasteras, Sweden, 2014), IEEE, pp. 219–228.
- [9] DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.
- [10] FRITZSON, P.; SHAHMEHRI, N.; KAMKAR, M.; GYIMOTHY, T. Generalized algorithmic debugging and testing. *LOPLAS* 1, 4 (1992), 303–322.
- [11] GIRGIS, H. Z.; JAYARAMAN, B. Javadd: a declarative debugger for java. Tech. rep., University at Buffalo, Department of Computer Science and Engineering, 2006.
- [12] GONZÁLEZ, J.; INSA, D.; SILVA, J. A new hybrid debugging architecture for eclipse. In *LOPSTR* (Madrid, Spain, 2013), Springer, pp. 183–201.
- [13] GONZÁLEZ-BLANCH, F.; ROSES, F.; SERRANO, S. *Depurador Declarativo de programas JAVA*. Tese de Doutorado, Master's thesis Universidad Complutense de Madrid. Available from: <https://eprints.ucm.es/9114/>, 2006.
- [14] GROCE, A.; HOLMES, J.; MARINOV, D.; SHI, A.; ZHANG, L. An extensible, regular-expression-based tool for multi-language mutant generation. In *ICSE-Companion* (Gothenburg, Sweden, 2018), IEEE, pp. 25–28.

- [15] HERMANN, C.; KUCHEN, H. Hybrid debugging of java programs. In *ICSOF* (Seville, Spain, 2011), Springer, pp. 91–107.
- [16] HUNTBACH, M. M. Algorithmic parlog debugging. In *Symposium on Logic Programming* (San Francisco, CA, 1987), IEEE, pp. 288–297.
- [17] INSA, D.; SILVA, J. An algorithmic debugger for java. In *ICSME* (Timisoara, Romania, 2010), IEEE, pp. 1–6.
- [18] INSA, D.; SILVA, J.; RIESCO, A. Speeding up algorithmic debugging using balanced execution trees. In *International Conference on Tests and Proofs* (2013), Springer, pp. 133–151.
- [19] KAMKAR, M. Application of program slicing in algorithmic debugging. *Information and Software Technology* 40, 11-12 (1998), 637–645.
- [20] KAMKAR, M.; SHAHMEHRI, N.; FRITZSON, P. Bug localization by algorithmic debugging and program slicing. In *International Workshop on Programming Language Implementation and Logic Programming* (1990), Springer, pp. 60–74.
- [21] KLUYVER, T.; RAGAN-KELLEY, B.; PÉREZ, F.; GRANGER, B. E.; BUSSONNIER, M.; FREDERIC, J.; KELLEY, K.; HAMRICK, J. B.; GROUT, J.; CORLAY, S., ET AL. Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB* (2016), pp. 87–90.
- [22] KO, A.; MYERS, B. Debugging reinvented. In *ICSE* (Leipzig, Germany, 2008), IEEE, pp. 301–310.
- [23] KOKAI, G.; HARMATH, L.; GYIM’OTHY, T. Algorithmic debugging and testing of prolog programs in proceedings of iclp’97. In *The Fourteenth International Conference on Logic Programming, Eighth Workshop on Logic Programming Environments Leuven, Belgium* (1997), pp. 8–12.
- [24] KOUH, H.-J.; KIM, K.-T.; JO, S.-M.; YOO, W.-H. Debugging of java programs using hdt with program slicing. In *International Conference on Computational Science and Its Applications* (2004), Springer, pp. 524–533.
- [25] KOUH, H.-J.; YOO, W.-H. The efficient debugging system for locating logical errors in java programs. In *ICCSA* (Montreal, Canada, 2003), Springer, pp. 684–693.
- [26] LAKHOTIA, A.; STERLING, L. Promix: A prolog partial evaluation system. In *The Practice of Prolog*. MIT Press, Cambridge, MA, 1990, pp. 137–179.
- [27] LAWRENCE, J.; BOGART, C.; BURNETT, M.; BELLAMY, R.; RECTOR, K.; FLEMING, S. D. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering* 39, 2 (2013), 197–215.
- [28] LEWIS, B. Debugging backwards in time. *arXiv preprint cs/0310016* (2003).
- [29] MCCABE, T. J. A complexity measure. *IEEE Transactions on software Engineering*, 4 (1976), 308–320.

- [30] MISSIER, P.; BELHAJJAME, K.; CHENEY, J. The w3c prov family of specifications for modelling provenance metadata. In *EDBT/ICDT* (Genoa, Italy, 2013), ACM, pp. 773–776.
- [31] MURTA, L.; BRAGANHOLO, V.; CHIRIGATI, F.; KOOP, D.; FREIRE, J. noworkflow: Capturing and analyzing provenance of scripts. In *IPAW* (Cham, 2015), Springer International Publishing, pp. 71–83.
- [32] NAISH, L. Declarative diagnosis of missing answers. *New Generation Computing* 10, 3 (1992), 255–285.
- [33] NAISH, L.; DART, P. W.; ZOBEL, J. The nu-prolog debugging environment. In *ICLP* (Lisbon, Portugal, 1989), MIT Press, pp. 521–536.
- [34] NILSSON, H.; FRITZSON, P. Algorithmic debugging for lazy functional languages. *Journal of functional programming* 4, 3 (1994), 337–369.
- [35] OSTRAND, T. J.; BALCER, M. J. The category-partition method for specifying and generating functional tests. *Communications of the ACM* 31, 6 (1988), 676–686.
- [36] PIMENTEL, J. F.; FREIRE, J.; MURTA, L.; BRAGANHOLO, V. Fine-grained provenance collection over scripts through program slicing. In *IPAW* (Cham, 2016), Springer International Publishing, pp. 199–203.
- [37] PIMENTEL, J. F.; MURTA, L.; BRAGANHOLO, V.; FREIRE, J. noworkflow: a tool for collecting, analyzing, and managing provenance from python scripts. *PVLDB* 10, 12 (2017).
- [38] POPE, B. Declarative debugging with buddha. In *AFP* (Tartu, Estonia, 2004), Springer, pp. 273–308.
- [39] POTHIER, G.; TANTER, É.; PIQUER, J. Scalable omniscient debugging. *ACM SIGPLAN Notices* 42, 10 (2007), 535–552.
- [40] PYTHON SOFTWARE FOUNDATION. diffliB helpers for computing deltas, 2019. Accessed: 2019-05-19.
- [41] ROMANO, J.; KROMREY, J. D.; CORAGGIO, J.; SKOWRONEK, J. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen’sd for evaluating group differences on the nsse and other surveys. In *Annual meeting of the Florida Association of Institutional Research* (Cocoa Beach, FL, 2006), FAIR, pp. 1–33.
- [42] SHAHMEHRI, N.; FRITZSON, P. Algorithmic debugging for imperative languages with side-effects. In *CC* (Schwerin, Germany, 1990), Springer, pp. 226–227.
- [43] SHAPIRO, E. Y. *Algorithmic Program Debugging*. Tese de Doutorado, Yale University, New Haven, CT, 1982. AAI8221751.
- [44] SHAPIRO, S. S.; WILK, M. B. An analysis of variance test for normality (complete samples). *Biometrika* 52, 3/4 (1965), 591–611.

- 
- [45] SILVA, J. A comparative study of algorithmic debugging strategies. In *International Symposium on Logic-Based Program Synthesis and Transformation* (2006), Springer, pp. 143–159.
  - [46] SILVA, J. A survey on algorithmic debugging strategies. *Advances in engineering software* 42, 11 (2011), 976–991.
  - [47] SILVA, J.; CHITIL, O. Combining algorithmic debugging and program slicing. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming* (2006), ACM, pp. 157–166.
  - [48] WEISER, M. Programmers use slices when debugging. *CACM* 25, 7 (1982), 446–452.
  - [49] WILCOXON, F. Individual comparisons by ranking methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.
  - [50] ZELLER, A. *Why programs fail: a guide to systematic debugging*. Morgan Kaufmann, San Francisco, CA, 2009.