Provenance-enhanced Algorithmic Debugging

Henrique Linhares
hlinhares@id.uff.br
Instituto de Computação
Universidade Federal Fluminense
Niterói, Rio de Janeiro, Brazil

João Felipe Pimentel
jpimentel@ic.uff.br
Instituto de Computação
Universidade Federal Fluminense
Niterói, Rio de Janeiro, Brazil

Troy Kohwalter
tkohwalter@ic.uff.br
Instituto de Computação
Universidade Federal Fluminense
Niterói, Rio de Janeiro, Brazil

Leonardo Gresta Paulino Murta
leomurta@ic.uff.br
Instituto de Computação
Universidade Federal Fluminense
Niterói, Rio de Janeiro, Brazil

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 25.26%, on average.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

algorithmic debugging, provenance, program slicing, software defects

ACM Reference Format:
Henrique Linhares, João Felipe Pimentel, Troy Kohwalter, and Leonardo Gresta Paulino Murta. 2019. Provenance-enhanced Algorithmic Debugging. In *XXXIII Brazilian Symposium on Software Engineering (SBES 2019), September 23–27, 2019, Salvador, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3350768.3350777

SBES 2019, September 23–27, 2019, Salvador, Brazil

ACM ISBN 978-1-4503-7651-8/19/09…$15.00
https://doi.org/10.1145/3350768.3350777

1 INTRODUCTION

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. The developers in charge of the debugging task usually do this search in an unsystematic way, based on guesses [18, 35]. These guesses demand a great effort, and the larger the software, the bigger the number of possible locations for the defects.

Almost 40 years ago, Shapiro [30] 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 and reducing the search space until the defective node is found [5].

After the initial proposal of algorithmic debugging in the early 1980s, researchers proposed several concrete implementations for the functional programming paradigm [2, 14, 17, 21, 23]. GADT [9] was the first algorithmic debugger to work with an imperative programming language: Pascal. GADT used a static program slicing technique [33] 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, creating a mismatch between the original program structure and the questions asked to developers. 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.

Even though algorithmic debugging is a systematic way to locate defects, the debugging sessions can still be time-consuming. Execution trees with a large number of nodes lead to algorithmic debugging sessions with many questions, and the higher the number of questions, the more time is necessary to find the defect.

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 [19]. In the specific context of debugging, it encompasses all the data and computations that were necessary to derive the program results. DebugProv uses a dynamic program slicing technique [7], which captures the steps used by a program to generate the values presented in its variables, function results, or outputs.

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 [8]. 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 25.26% in the number of algorithmic debugging questions. This 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.

The main goal of this paper is to introduce DebugProv, which brings the following contributions: (i) it is the first approach to implement algorithmic debugging for an interpreted and dynamically-typed imperative language (i.e., Python), (ii) it is the first approach that uses provenance collected through dynamic program slicing to enhance the execution tree used in algorithmic debugging, and (iii) it was able to reduce in about 25% the number of questions asked during algorithmic debugging sessions.

This paper is organized as follows: Section 2 introduces algorithmic debugging. Section 3 describes our approach, including its architecture, the provenance enhancement technique, and the implementation aspects. Section 4 presents DebugProv evaluation and discuss the results. Section 5 presents the related work. Section 6 concludes our work and presents some future work.

2 ALGORITHMIC DEBUGGING

In this section, we use the Python script presented in Figure 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 bug in the function *find_min*.

Caballero et al. [5] interpret algorithmic debugging as a process with two phases: (i) capturing and (ii) navigation. The **capturing phase** employs 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 function calls. The root node of the execution tree corresponds to the first call that started the computation. Figure 2 illustrates the execution tree built from the execution of our guiding example.

Every node in the execution tree can contain descendants. A 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, the function *readfile* has two immediate descendants: *open* and *load*.

```
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 1: Python script used as a guiding example

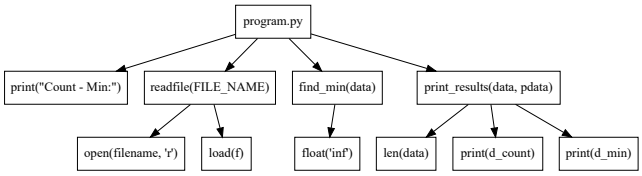

Figure 2: The execution tree of our guiding example

The **navigation phase** is the core of an algorithm debugging process. It uses a navigation strategy to traverse the generated execution tree and interacts with the user 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.

Therefore, every question asked in an algorithmic debugging session leads to a prune. The algorithmic debugger reduces the search space by assuming only one defective node per session. For programs with multiple defects, the programmer must run multiple debugging sessions. When a node *n* is evaluated and classified as valid (i.e., correct), *n* and all the subtree rooted at *n* are pruned (removed) from the search space. In Figure 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. Thus, *n* becomes the new root of the execution tree. In Figure 2, it occurs when the debugger classifies *find_min* as invalid. In this case, the debugging session continues to check if the bug 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 debugger finds the defective node, the session ends, and the debugger presents to the user the defective node, which relates to a defective function call.

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

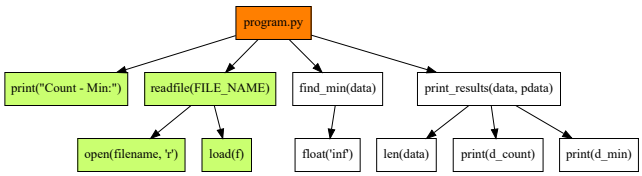

Figure 3: An algorithmic debugging session (Step 3)

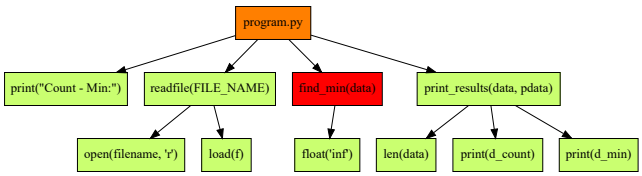

Figure 4: An algorithmic debugging session (last step)

navigation strategy. Many navigation strategies have been proposed for algorithmic debugging in the literature [32]. The most relevant are Single Stepping, Top Down, Heaviest First, and Divide and Query. Table 1 describes these strategies and presents the navigation steps to find the defective node *find_min* in Figure 2.

For instance, suppose that we are performing an algorithmic debugging session in our example shown in Figure 1, using the Top Down navigation strategy. Following the Top Down navigation strategy, the debugger first evaluates the root node, 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 next evaluated node is *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.

Then, 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 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 3. The nodes in green are valid, and the nodes in orange are invalid. Just the white ones are still in the search space.

In the next 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.

After this 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 4. The red color distinguishes the defective node.

3 DEBUGPROV

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 the debugging of Python programs. It is an open-source project available for download at https://github.com/gems-uff/debugprov. The algorithmic debugging session starts in command-line mode by making the following call, according to our guiding example: *debugprov program.py*.

In our algorithmic debugging process, we define a new phase before the navigation phase: the provenance enhancement phase. Thus, the architecture of DebugProv is based on three modules: the capturing module, the provenance enhancement module, and the navigation module. Figure 5 illustrates the main modules of DebugProv, which are detailed in the following.

3.1 Capturing module

The capturing module is responsible for running the defective program and capturing the data necessary to build an execution tree, the traditional algorithmic debugging structure, and the provenance graph, a structure used in our provenance enhancement technique. 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

Table 1: Navigation Strategies

Nav. Strategy	Description	Steps in Figure 2	# Quest.
Single Stepping	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
Top Down	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
Heaviest First	Variation of the Top Down strategy that selects the child with the biggest sub-tree instead of simply navigating according to the execution order.	program.py, print_results, readfile, find_min, float	5
Divide and Query	Navigates to nodes that prune almost half of the execution tree for every answer.	print_results, readfile, find_min, float	4

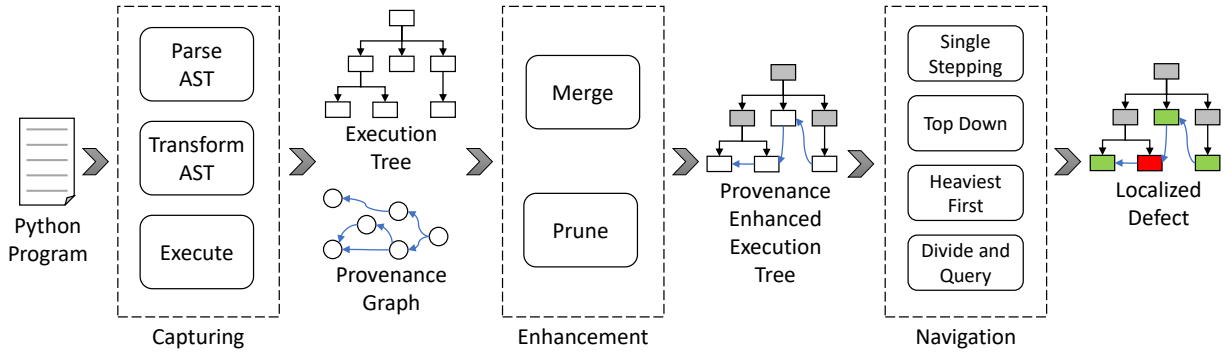

Figure 5: DebugProv overview

evaluation, (2) the activation associated with an evaluation, and (3) the representation (or the result of) that evaluation.
- Dependencies: by using dynamic program slicing [7], 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.

As a result, this module provides to the enhancement module both the execution tree and the provenance graph. Considering our example shown in Figure 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*.

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. In DebugProv, we select only the activities that appear in the provenance of the entity that represents the incorrect result. 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 [19].

To implement the capturing module, we chose to adopt an existing tool named noWorkflow [20, 24, 25] in version 2.0-alpha. We chose this tool because it not only captures the provenance of Python script, as required by our provenance enhancement module, but it also captures enough execution data in the provenance for the generation of execution trees by our approach.

3.2 Enhancement module

The enhancement module receives two structures: an execution tree and a provenance graph. The responsibility of the enhancement module is to use both of these structures to produce a new structure: The provenance-enhanced execution tree.

The production of the provenance-enhanced execution tree begins 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.

Therefore, after the user report which output data is not correct, the enhancement module searches in the provenance graph for the dependencies that influenced (directly or indirectly) such incorrect output. This process is done in backward, answering the following question: "Which function activations contributed in the production of the incorrect output?". The enhancement of the execution tree consists of inserting over the execution tree the dependency relations that contributed to the production of the incorrect output.

When the execution tree is enhanced with provenance data, the enhancement module can distinguish the nodes that contributed to the production of the incorrect output from those nodes that did not contribute. This information allows DebugProv to perform

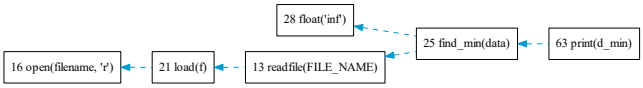

Figure 6: A provenance graph

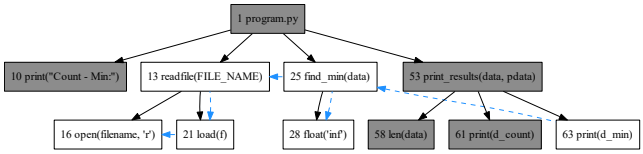

Figure 7: The provenance enhanced execution tree

an additional tree pruning by removing from the search space the nodes that did not contribute to the production of the incorrect outcome. Thus, the provenance enhancement potentially reduces the total number of required questions to detect defective nodes, speeding up the algorithmic debugging session.

For example, we present a provenance graph in Figure 6. The result of the provenance enhancement can be observed in Figure 7. The blue dashed edges represent the dependencies between nodes and they go from the influenced to the influencer node, as dictated by PROV [19], 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.

3.3 Navigation module

The navigation module of DebugProv employs the standard navigation strategies of algorithmic debugging, discussed in Section 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 1. For instance, similarly to the example discussed in Section 2, suppose that DebugProv is using the Top Down navigation strategy, but now over the provenance enhanced execution tree, shown in Figure 7. 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 askes 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 40% gain. Table 2 summarizes the improvements introduced by DebugProv in our guiding example, for each one of the navigation strategies.

Table 2: Improvements obtained by DebugProv in the guiding example

Nav. Strategy	Steps in Figure 7	# Questions	Reduction
Single Stepping	open, load, readfile, float, find_min	5	16%
Top Down	readfile, find_min, float	3	40%
Heaviest First	readfile, find_min, float	3	40%
Divide and Query	readfile, find_min, float	3	25%

3.4 Fallback mechanism

In a perfect scenario, the provenance data collected by the capturing module would always precisely represent the nodes that were responsible for producing an incorrect output. However, the 2.0 alpha version of noWorkFlow, as every provenance collection tool, has limitations. It currently does not capture provenance from async programming (async def, async for, async with, and await), methods defined inside classes in the main script, and some other Python constructs (with, assert, raise, del, global, nonlocal, and yield from). In these cases, the provenance data captured by noWorkFlow could miss the defective node.

In order to provide a workaround to these limitations of noWorkFlow, we implemented the fallback mechanism that is applied when an execution tree is enhanced with provenance and traversed by a navigation strategy but the defective node is not found. In such cases, we assume that the provenance data did not capture the dependency on the defective node. Therefore, the search continues by evaluating the nodes that were not on the provenance graph.

The fallback mechanism only needs to be executed when a node that was supposed to be on the provenance graph is not due to provenance capture limitations. In these cases, it can increase the number of questions to locate the defective node. We expect that, as the provenance collection tools improve and get more precision, the fallback mechanism will be less necessary.

4 EVALUATION

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

- ***RQ1***: *How effective is provenance enhancement in the reduction on the number of questions in algorithmic debugging?*
- ***RQ2***: *How provenance enhancement improves each navigation strategy?*

4.1 Materials and methods

Our evaluation process is described by the following workflow:

(1) Select a set of Python programs

Table 3: 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	Longest Common Subsequence	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	llSourcell/linear_regression_demo	Machine Learning	21

(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, the respective repositories, and the number of lines (LOCs) are presented in Table 3.

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 [8] 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 ">=" (grater 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* [12], which is a multi-language regex-based mutant generator, to produce mutants for our selected programs. 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. The complete set of mutation operators supported by universalmutator is available in the tool repository. In this experiment, we used the python operators[1] and the universal operators[2].

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[3] (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

[1] https://github.com/agroce/universalmutator/blob/master/universalmutator/static/python.rules
[2] https://github.com/agroce/universalmutator/blob/master/universalmutator/static/universal.rules
[3] The current version of DebugProv does not deal with silent defects.

Table 4: 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 errors	4	1,794
Mutants without a unique oracle	5	2,506
Used Mutants	6	458

produced the same output of the original 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 an error. 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* [27], a Python module that compare 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 basically 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 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.

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.

4.2 How effective is provenance enhancement in the reduction on 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 [31] 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 2.2×10^{-16}). Therefore, we used the Wilcoxon Signed-Rank test [34], a non-parametric test to compare two paired samples. The resulting p-values were below 2.2×10^{-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 8 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 [28]. In addition to the 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. It is important to notice that the questions asked during a provenance-enhanced session are similar to the questions asked in a traditional algorithmic debugging session, as all of them come from the same execution tree.

Finally, we analyzed each program individually. In this analysis, we computed at the number of questions for each mutant of that program, both for the execution tree with and without provenance. In Table 5, 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. In other cases, such as the 04 - LU Decomposition, we were able to observe an impressive reduction. Since the reduction in the number of questions come from the irrelevant nodes removed from the search space, the performance of the provenance enhancement technique is associated with the number of unrelated nodes to a given result. The higher the number of unrelated nodes, the higher will be the reduction in the number of questions. The number of unrelated nodes for a given result explains the difference in the performance between the mutants of 02 - Bisection and 04 - LU Decomposition. However, in situations where the fallback

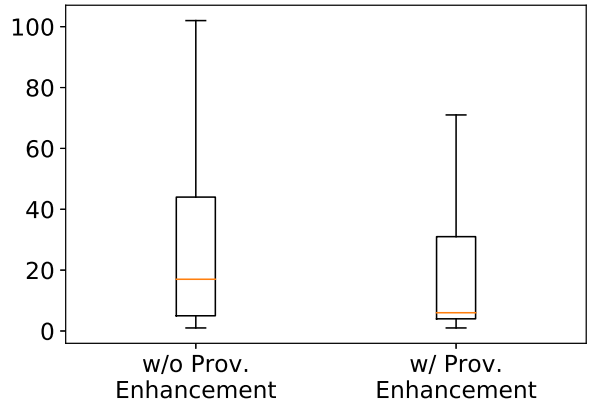

Figure 8: Boxplot of the number of questions required to locate the defective node without provenance enhancement (left) and with provenance enhancement (right). Outliers removed.

mechanism was largely used, the number of questions increased with the enhancement (e.g., 19 - Bubblesort and 20 - Quicksort).

RQ1: *How effective is provenance enhancement in the reduction on 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 to 6. We also observed that the reduction varies from program to program.

4.3 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 2.2×10^{-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), and observed a significant difference between the samples for all navigation strategies. We also used the Cliff's delta (for paired samples) to calculate the effect size and found medium effect size for Single Stepping and Heaviest First. For Top Down we observed a large effect size, and for Divide and Query a small effect size.

Table 6 shows the reduction of the number of questions for each navigation strategy. Moreover, Single Stepping was the navigation strategy with the largest proportional reduction, followed by Top Down. However, by analyzing the total number of questions, we conclude that Heaviest First is the navigation strategy with the best performance, both for execution trees with and without provenance enhancement. Surprisingly, Divide and Query is the worst navigation strategy in terms of proportional reduction of the number of questions.

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.

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 questions asked to the user during a debugging session. However, different questions may have different difficulties, and some questions can be more time-consuming than others. Consequently, debugging sessions with fewer questions are not necessarily easier or faster than the ones with more questions.

Construct. We implemented the *fallback* mechanism (described in Section 3) to handle imperfections in the provenance data. The usage of the *fallback* mechanism 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* [12], implements a set of operators for generating mutants. Even though the mutant generated by *universalmutator* are defective programs, 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

Table 5: Individual analysis of provenance enhancement by selected program.

#	Program	Mutants	# Questions without Prov.	# Questions with Prov.	Reduction	Fallbacks
02	Bisection	71	24,546	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

Table 6: Navigation strategies performance over trees with and without provenance enhancement.

Navigation Strategy	# Questions without Prov.	# Questions with Prov.	Reduction	Effect size	Interpretation
Single Stepping	36,498	25,181	31.01%	0.347	Medium
Top Down	13,787	10,252	25.64%	0.476	Large
Heaviest First	11,683	9,170	21.51%	0.454	Medium
Divide and Query	13,251	11,617	12.33%	0.163	Small

mutants that shared the same behavior, compromising somehow the generalizability of our results.

5 RELATED WORK

Since the introduction of algorithmic debugging in 1982 [30], several tools were developed to instantiate the base concept in different programming paradigms and environments, such as logic [22] and functional [4, 26]. However, in this work, we focus only on approaches for imperative programming languages.

The first project that adapted the concept of algorithmic debugging for an imperative programming language with side-effects was presented in 1990 [29]. The implemented prototype, called GADT (Generalized Algorithmic Debugging and Testing), was able to run algorithmic debugging sessions in programs written in Pascal. GADT [29] shares several points in common with our work. Both GADT and DebugProv work with the imperative paradigm. GADT also 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 contrast, DebugProv performs transformations in the AST that are transparent to the users and builds execution trees that represent the source codes, without artificial changes.

In 2003, HDT [16] was the first tool to bring algorithmic debugging to Java. HDT combines algorithmic debugging with breakpoint debugging but uses the standard execution tree proposed in 1982. Some other tools apply algorithmic debugging in Java. JavaDD [10] keeps a deductive database with information about past executions of a Java program and queries this database to perform a debugging session. DDJ [15] uses concepts of equivalence classes and def-use chains to reduce the number of questions. JHyde [13] is a hybrid debugger for Java that combines techniques from algorithmic debugging and omniscient debugging. HDJ [11] is an extension of DDJ [15] that combines algorithmic debugging, omniscient debugging, and breakpoint 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 [13] and González et al. [11] are capable of performing omniscient debugging, which allow navigation not only forward but also backward into the program execution, without needing to re-execute the program.

As far as we know, DebugProv is the first approach that uses provenance to enhance the execution tree and, consequently, reduce the number of questions asked to users during algorithmic debugging sessions. It is also the first approach that runs algorithmic

debugging in an interpreted language (Python). Moreover, the only tool that employed program slicing to enhance the execution tree until now (GADT) used static program slicing due to the static typing characteristic of Pascal. In general, the use of dynamic program slicing, adopted by DebugProv, reaches more precise results [1].

6 CONCLUSION

In this work, we presented DebugProv, an algorithmic debugging tool for Python. DebugProv is the first tool that uses provenance to enhance algorithmic debugging over a dynamically-typed and interpreted imperative language, reducing the number of questions during the debugging session. 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. Our results showed that provenance enhancement reduced the number of algorithmic debugging questions by 25.26%, on average.

The adoption of noWorkflow in the capturing module imposes limitations over Debugprov, due to some Python constructs that are not currently supported by noWorkflow. Moreover, since the provenance enhancement technique demands the developer 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). Additionally, limitations in the provenance collection of DebugProv may prevent it from reducing the number of questions in some kinds of programs. This occurs when a node from the execution tree has an indirect dependency on another node (e.g., through reading/writing tuples in a database).

We have identified several possibilities for future work. Currently, DebugProv runs the debugging session based on a single execution of a program. We intend to take advantage of multiple executions to refine even further the pruning process. Another future work is to build a probabilistic system that integrates with DebugProv. Additionally, the why-not provenance [6] could be explored in DebugProv to detect silent defects, such as infinite loops or the absence of outputs. Lastly, a possible future work would be related to debugging inside the function, line-by-line, using provenance information analogously to what DebugProv currently does for pruning the execution tree.

ACKNOWLEDGMENTS

The authors would like to thank CAPES and CNPq for the financial support.

REFERENCES

[1] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. *ACM SIGPlan Notices* 25, 6 (1990), 246–256.
[2] Evyatar Av-Ron. 1984. *Top-down diagnosis of Prolog programs.* Master's thesis. Weizmann Institute of Science, Rehovot, Israel.
[3] Rafael Caballero, Christian Hermanns, and Herbert Kuchen. 2007. Algorithmic debugging of Java programs. *ENTCS* 177 (2007), 75–89.
[4] Rafael Caballero, Enrique Martin-Martin, Adrian Riesco, and Salvador Tamarit. 2014. EDD: A declarative debugger for sequential erlang programs. In *TACAS*. Springer Berlin Heidelberg, Berlin, Heidelberg, 581–586.
[5] Rafael Caballero, Adrián Riesco, and Josep Silva. 2017. A survey of algorithmic debugging. *CSUR* 50, 4 (2017), 60.
[6] Adriane Chapman and HV Jagadish. 2009. Why not?. In *SIGMOD*. ACM, Providence, RI, 523–534.
[7] Zhifei Chen, Lin Chen, Yuming Zhou, Zhaogui Xu, William C Chu, and Baowen Xu. 2014. Dynamic slicing of Python programs. In *COMPSAC*. IEEE, Vasteras, Sweden, 219–228.
[8] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.
[9] Peter Fritzson, Nahid Shahmehri, Mariam Kamkar, and Tibor Gyimothy. 1992. Generalized algorithmic debugging and testing. *LOPLAS* 1, 4 (1992), 303–322.
[10] Hani Z Girgis and Bharat Jayaraman. 2006. *JavaDD: a declarative debugger for java.* Technical Report. University at Buffalo, Department of Computer Science and Engineering.
[11] Juan González, David Insa, and Josep Silva. 2013. A new hybrid debugging architecture for eclipse. In *LOPSTR*. Springer, Madrid, Spain, 183–201.
[12] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. 2018. An extensible, regular-expression-based tool for multi-language mutant generation. In *ICSE-Companion*. IEEE, Gothenburg, Sweden, 25–28.
[13] Christian Hermanns and Herbert Kuchen. 2011. Hybrid debugging of java programs. In *ICSOFT*. Springer, Seville, Spain, 91–107.
[14] Matthew M Huntbach. 1987. Algorithmic PARLOG debugging. In *Symposium on Logic Programming*. IEEE, San Francisco, CA, 288–297.
[15] David Insa and Josep Silva. 2010. An algorithmic debugger for Java. In *ICSME*. IEEE, Timisoara, Romania, 1–6.
[16] Hoon-Joon Kouh and Weon-Hee Yoo. 2003. The efficient debugging system for locating logical errors in java programs. In *ICCSA*. Springer, Montreal, Canada, 684–693.
[17] Arun Lakhotia and Leon Sterling. 1990. ProMiX: A Prolog partial evaluation system. In *The Practice of Prolog*. MIT Press, Cambridge, MA, 137–179.
[18] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D Fleming. 2013. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering* 39, 2 (2013), 197–215.
[19] Paolo Missier, Khalid Belhajjame, and James Cheney. 2013. The W3C PROV family of specifications for modelling provenance metadata. In *EDBT/ICDT*. ACM, Genoa, Italy, 773–776.
[20] Leonardo Murta, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire. 2015. noWorkflow: Capturing and Analyzing Provenance of Scripts. In *IPAW*. Springer International Publishing, Cham, 71–83.
[21] Lee Naish. 1992. Declarative diagnosis of missing answers. *New Generation Computing* 10, 3 (1992), 255–285.
[22] Lee Naish, Philip W Dart, and Justin Zobel. 1989. The NU-Prolog Debugging Environment. In *ICLP*. MIT Press, Lisbon, Portugal, 521–536.
[23] Henrik Nilsson and Peter Fritzson. 1994. Algorithmic debugging for lazy functional languages. *Journal of functional programming* 4, 3 (1994), 337–369.
[24] João Felipe Pimentel, Juliana Freire, Leonardo Murta, and Vanessa Braganholo. 2016. Fine-Grained Provenance Collection over Scripts Through Program Slicing. In *IPAW*. Springer International Publishing, Cham, 199–203.
[25] Joao Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2017. noWorkflow: a tool for collecting, analyzing, and managing provenance from python scripts. *PVLDB* 10, 12 (2017), 4.
[26] Bernard Pope. 2004. Declarative debugging with Buddha. In *AFP*. Springer, Tartu, Estonia, 273–308.
[27] Python Software Foundation. 2019. difflib Helpers for computing deltas. https://docs.python.org/3/library/difflib.html Accessed: 2019-05-19.
[28] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys. In *Annual meeting of the Florida Association of Institutional Research*. FAIR, Cocoa Beach, FL, 1–33.
[29] Nahid Shahmehri and Peter Fritzson. 1990. Algorithmic debugging for imperative languages with side-effects. In *CC*. Springer, Schwerin, Germany, 226–227.
[30] Ehud Yehuda Shapiro. 1982. *Algorithmic Program Debugging.* Ph.D. Dissertation. Yale University, New Haven, CT. AAI8221751.
[31] Samuel Sanford Shapiro and Martin B Wilk. 1965. An analysis of variance test for normality (complete samples). *Biometrika* 52, 3/4 (1965), 591–611.
[32] Josep Silva. 2011. A survey on algorithmic debugging strategies. *Advances in engineering software* 42, 11 (2011), 976–991.
[33] Mark Weiser. 1982. Programmers use slices when debugging. *CACM* 25, 7 (1982), 446–452.
[34] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. http://www.jstor.org/stable/3001968
[35] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging.* Morgan Kaufmann, San Francisco, CA.