Aardvark: Comparative Visualization of Data Analysis Scripts

Rebecca Faust, Carlos Scheidegger, and Chris North

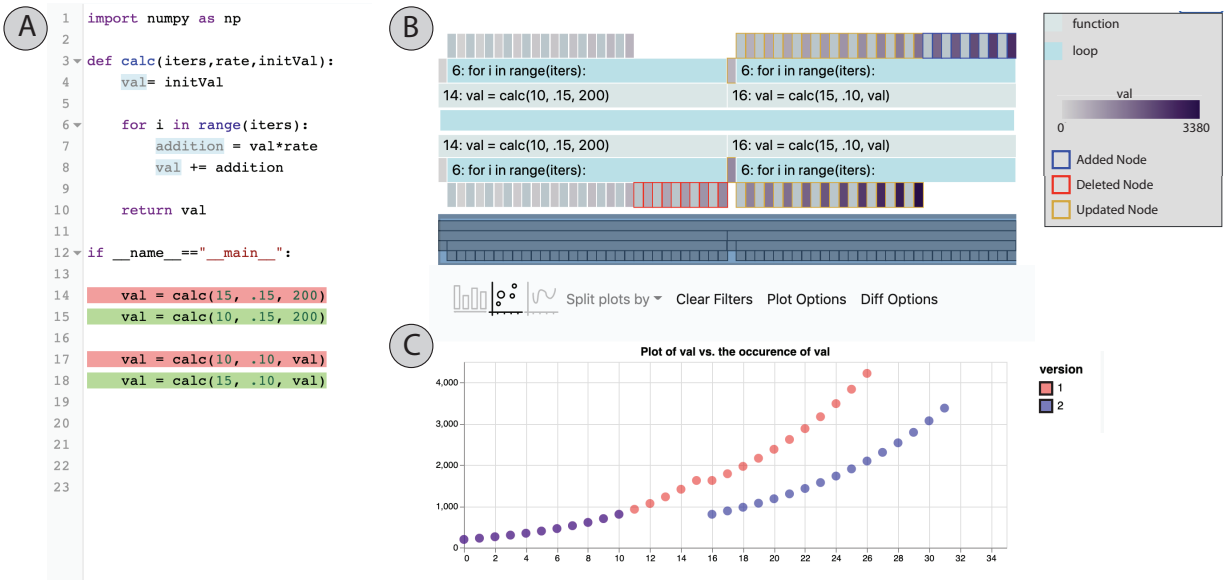

Fig. 1: An overview of Aardvark. In the source view, (A), lines highlighted in green were added, while lines highlighted in red were deleted. (B) shows the new generalized context tree. The tree for the original version builds downwards from the center block, while the tree for the modified version builds upward. The border color of blocks indicates if they were changed. Red borders indicate deleted nodes from the original execution, the blue border indicates nodes added by the modified execution, and gold borders indicate nodes whose values changed between executions. (C) shows one of the comparative plots.

Abstract— Debugging programs is one of the most challenging and time consuming parts of programming. Data science scripts present additional challenges as debugging often centers around more exploratory tasks, such as understanding the differences between results under different parameter settings. In fact, a common exploratory debugging practice is to run, modify, and re-run a script to observe the effects of the modification. Analysts perform this process frequently as they explore different settings and algorithms in their analysis. However, traditional debugging methods are not well suited to comparing across multiple executions of a script. They often require maintaining two instances of the debugging method and making manual, serial comparisons of program values. To address this gap, we present Aardvark, a comparative trace-based debugging method for identifying and visualizing the differences between two executions of data analysis scripts. Aardvark traces two consecutive instances of an analysis script, identifies the differences between them, and presents them through comparative visualizations. We present a prototype implementation in Python as well as an extension to support scripts in Jupyter notebooks. Finally, to demonstrate Aardvark, we provide two usage scenarios on real world analysis scripts.

Index Terms—Interactive Visualization, Program Traces, Jupyter, Debugging, Comparison

1 Introduction

Debugging and understanding program behavior is one of the most time consuming and challenging aspects of programming. Data analytics programs present an additional challenge in that, frequently, no traditional "bug" exists to locate and correct. Rather, analysts often perform exploratory debugging tasks where they want to understand the effects of variations to their analysis programs, such as different parameters or analysis algorithms. Analysts commonly perform comparative exploratory debugging tasks where they run the analysis, evaluate the execution values, make a small change (e.g., modify a parameter), re-run the analysis, and re-evaluate the execution values to understand the effects of the change on the analysis. In fact, this is a recognized process for general debugging tasks in the program development process, labeled as the edit-run cycle by Alaboudi and LaToza [3]. The edit-run cycle necessitates the comparison of consecutive executions to understand the impacts of the edits. Several other studies of debugging practices and processes identify strategies that also employ this comparative process [17, 23, 36].

The serial nature of current debugging practices already present limitations for exploratory debugging tasks, as described by Faust et. al. [16]. These practices have virtually no support for comparative debugging and understanding tasks. They either require the inspection of simultaneous instances of the debugging method or require people adequately remember the values from the initial execution. In print statement debugging, people must compare simultaneous printouts. Not only does comparing simultaneous printouts still suffer from the

- *Rebecca Faust and Chris North are with the Department of Computer Science, Virginia Tech. E-mail: {rfaust,north}@vt.edu.*
- *Carlos Scheidegger is with the HDC Lab, Department of Computer Science, University of Arizona. E-mail: cscheid@arizona.edu*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

problem of serial inspection, but it suffers an additional problem of requiring people to make many pairwise comparisons and build those comparisons into an overall view of the changes in the data. We know that building mental representations of data causes significant mental overhead and are often not an accurate view of the data [29]. Similarly, to fully compare program executions with a step-through debugger, people must step through simultaneous instances of the debugger, one on the new version and one on the old. Just like with print debugging, simultaneous inspection of individual pairs of values may not help people build a faithful mental view of the data.

Faust et al. introduced Anteater, a method for visualizing analysis scripts via their traces [16]. While employing visualization helps overcome the limitations of serial value inspection by automatically collecting and visualizing program values, it suffers the same problem of requiring people to run and compare individual instances of Anteater to identify the effects of changes. While visualizations help show larger trends and behaviors in data, they still present challenges when comparing side-by-side or recalling past instances. For example, Fig. 6 shows the visualizations from two instances of Anteater. It may not be immediately apparent that the modified version (the bottom row) has fewer NaN's than the original, especially if these are viewed in succession rather than side-by-side. As a result, debugging methods need to support the direct comparison of consecutive script executions to quickly illustrate the effects of changes. To do so, we must address the following two questions: 1) how can we identify and capture the differences between two consecutive script executions and 2) how can we apply the principles of comparative visualization to illustrate the differences between two script executions?

In this paper, we present Aardvark, a method for creating comparative visualizations of multiple instances of analysis scripts to illustrate the effects of changes to the analysis. Aardvark leverages the existing infrastructure in Anteater and expands beyond it in two primary ways: 1) it expands the tracing infrastructure to identify the differences between two consecutive executions of analysis script and 2) it modifies the visualization system to apply principles of comparative visualization to present visualizations that illustrate the differences in the values across the two executions. Rather than requiring people to contrast two instances of their debugging methods, Aardvark provides a single view to illustrate the effects of source code change. Fig. 1 presents an overview of the comparative views provided by Aardvark.

In summary, the contributions of this paper include:

- A trace-based comparative debugging method to automatically identify and visualize the effects of script changes
- A prototype implementation in Python with an extension to Jupyter notebooks.
- Two usage scenarios to illustrate the benefits of comparative visualizations in analysis script debugging.

2 Related Work

In this section, we discuss relevant works related to identifying and visualizing differences in analysis programs from both the source code and execution information. Note, we do not go into detail about visual debugging approaches here. While many works exist that aim to bring visualization into traditional debugging methods, such as adding visualization in breakpoint debuggers [10, 32], visualizing symbolic executions [6], or approaches similar to ours that visualize data from traces or value tracking [5,12], we aim to focus specifically on methods that support comparison across multiple executions. For more detail on visual debugging methods, see [16].

Source Code Diffing Several tools exist for diffing source code. The Unix diff utility and Git diff command support the diffing of two textual source files by finding the minimum number of line additions and removals to produce the new file from the old one. This method provides very coarse grained differences in source code files. Canfora et al. [8] expand on the Unix diff utility to provide more fine grained diffing of textual source files that supports changes and moves, as well as additions and deletions. Horwitz [25] expands their definition of diffing to differentiate between semantic and textual changes when identifying changes to the source code. Semantic Diff [26] provides a semantic report of the differences between two versions of a procedure through the inspection of input-output behavior of the procedure. Duszyanski et al. present a method for generating N-way diff to compare across many software variants [14].

Rather than diffing text source files, other methods first parse source code into abstract syntax trees (AST's) and then find diff the resulting trees [13, 15, 19, 24]. Similar to textual diffing methods, these methods look for the minimum number of edits to move from the original AST to the new one, where an edit can be an addition, removal, change, or move. These methods build off of Chawathe et al.'s algorithm for detecting change in hierarchically structured information [9] by leveraging structure and information specific to AST's to more accurately identify differences. As part of Aardvark, we use gumtree [15] for source code differencing to highlight the textual program changes. However, Aardvark requires additional methods for diffing the resulting traces (discussed in Section 5.1).

Trace Diffing and Comparison Suzuki et al.'s TraceDiff [33] relates most closely to Aardvark. TraceDiff provides automatic feedback and hints for students completing introductory programming assignments. It uses program traces to illustrate differences between an incorrect version of a program and a synthesized correct version. Leveraging these differences, it provides hints to guide students through their mistakes. While similar to Aardvark, this tool focuses on assisting students in their assignments and utilizes minimal visualization.

Taheri et al.'s DiffTrace [34] provides diffs of program traces for debugging high-performance computing code. It collects whole program function call traces per process/thread and uses concept lattices to build the traces and identify the differences between a normal trace and a fault laden trace. In contrast, Aardvark operates only on sequential programs and presents differences in program values, such as variables and expressions, in addition to calling structure.

Other methods exist for comparing traces without formally diffing them. Miranksyy et. al. use trace comparison to find sources of errors in software systems [28]. They take an existing trace with correct behavior and compare it against a trace in which the software misbehaves to calculate the entropy between them. German et al. [20] do not directly collect a trace but allows people to specify an area of failure and then build a change impact graph to demonstrate the effects of prior code changes on this area.

Comparative Visualization Munzner's framework for visual design identifies comparison as a low-level user goal when analyzing data [29]. Many visualization applications support this goal in a variety of domains, including volume rendering [38], event sequences [39], and brain connectivity analysis [4]. While there exist many applications supporting comparison, there exist few taxonomies and frameworks for creating comparative visualizations. Pagendarm and Post describe approaches for comparative visualization in images [30]. Graham and Kennedy surveyed methods for visualizing (and thus comparing) multiple trees [22]. In 2011, Gleicher et al. [21] developed a taxonomy for designing comparative visualizations for information visualization. This taxonomy remains as the primary taxonomy for designing comparative visualizations. It defines three types of comparative visualization: juxtaposition, superposition and explicit encoding. We employ this taxonomy in Aardvark's visual design and discuss these types in more detail in Section 5.2.

Visual Comparison of Traces Cornelissen and Moonen create visualizations to highlight repetitive behavior and execution phases in a single program identified by matching the trace to itself [11]. Intel's Trace Analyzer provides timeline visualizations to compare traces of parallel applications [2]. Trümper et. al. create multiscale visualizations for comparing large traces using icicle plots and edge bundles [35]. Voigt et al. [37] create trace visualizations of method calls and object accesses for large scale traces. While all of these tools visually compare traces, they all address a specific problem: detecting similarities inside a single program, comparing executions of parallel programs, and comparing executions of large scale programs.

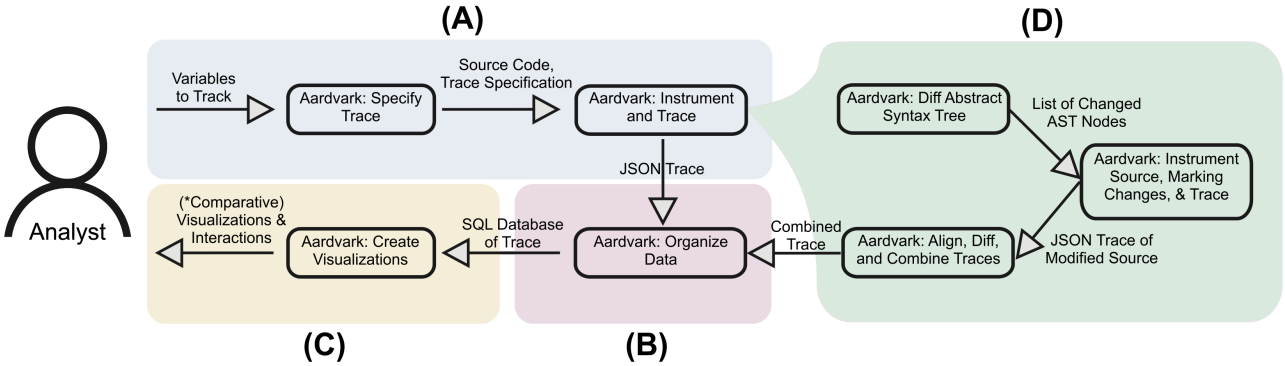

Fig. 2: Overview of the Aardvark system. In (A), the analyst first specifies which values to track. Aardvark passes this information to the tracer which automatically transforms the script to collect the specified values and execution structure. If no prior trace exists, it moves directly to (B) to organize the data for visualization, before passing the organized data to the front-end for visualization, in (C). If a prior trace exists, Aardvark expands step (A) to identify the differences between the source code of the scripts and the resulting traces, shown in (D). From (D) it passes the combined trace for organization in (B) and then provides comparative visualizations, in (C).

In contrast, Aardvark supports comparing traces to understand the impact of small changes to single-threaded Python analysis programs.

3 Background

Aardvark leverages a prior method, Anteater, that visualizes traces for debugging analysis scripts [16]. In this section, we give a brief overview of the components leveraged from Anteater's design. For full description, see the Anteater paper [16].

Fig. 2 shows an overview of the Aardvark method. Aardvark leverages the base infrastructure of the Anteater system in Fig. 2 (A), (B), and (C). In (A), the analyst specifies which variables and expressions to track, forming a trace specification. Anteater then passes this information to the tracer. Once in the tracer, Anteater automatically transforms the script to track the specified values and capture the execution's structure, i.e., function calls and loops. It does this by parsing the abstract syntax tree (AST) and inserting new nodes to capture script behaviors. Once the transformation is complete, Anteater executes the transformed script to generate the trace. After generating the trace, Anteater structures the data into a SQL database for efficient visualization. Finally it passes the structured data to the front-end for visualization.

Anteater provides two primary visualizations: the generalized context tree and the value visualization. The generalized context tree (GCT) illustrates the structure of the execution structure using an icicle plot, as shown in Fig. 9. This enables people to see the calling and looping structures of the script. The second visualization shows the behavior of the values the analyst specified to track. Anteater generally uses either a histogram or scatterplot to give overviews of the program values. Aardvark builds on Anteater's design, directly using its visualizations as the base instance when no prior trace exists to compare with. Additionally, its comparative visualizations extend the base visualizations of Anteater to illustrate the differences between the traces.

Aardvark uses this base infrastructure to visualize singular traces. To support comparison, it expands each component. It expands component (A) to identify and capture the differences between two instances of a script (shown in Fig. 2 (D)). Aardvark modifies component (B) to both traces and link between matching instances of values. Finally, in (C) it provides comparative visualizations that illustrate the effects of the changes on the program values. Each of these modifications is described in greater detail in Sec. 5.

4 Classification of Program Changes

In this section we describe a classification of changes to a program from two perspectives: changes to the source code and changes to the trace. We have found, through the inspection of various program changes, that the two classifications do not have a direct mapping. As such, a change in the source code does not have a single, well-defined corresponding change in the trace. In fact, it may cause any type of change to the trace. As such, we discuss the two classifications disjointly.

Several works exist on classifying types of changes to programs, for several types of program representations. While most focus on changes to static representations of the program (e.g., source code), these changes generalize to traces as well.

Purushothaman and Perry classify changes based on the textual changes to the source code [31]. They present 4 types of changes: (1) modifications to existing lines, (2) insertions of new statements between existing lines, (3) deletions of existing lines, and (4) modifications of lines accompanied by an insertion and/or deletion of lines. The fourth type combines the first three types.

Fluri and Gall [18] define a similar classification for changes to the AST. They describe four program modification operations on an AST: insert a new leaf node, delete a node from its parent, move a node to a new parent, update the value of an existing node. Additionally, they provide several higher level changes for object oriented programs stem from one or more of the base modification operations. We use this classification when describing the types of source code changes.

Lehnert et al. [27] have a similar taxonomy for classifying change, based on that of Fluri and Gall [18]. Rather than classifying changes to AST's, they define software as a graph where nodes are artifacts such as UML diagrams or C++ classes and edges are dependencies between the artifacts. Lehnert et al. present two tiers of changes: atomic and composite. Atomic changes include the addition, deletion, and modification of both nodes and edges. Composite changes require multiple atomic changes and include: move, replace, split, merge, swap.

While all of these classifications operate on slightly different program representations, there seems to be a core set of operations for classifying change: add, update, and delete, with more complex changes consisting of combinations of these operations. We use these core change types to classify the changes in both the source code and the traces. In the remainder of this section, we discuss the two classifications in more detail and how Aardvark supports them.

4.1 Source Code Changes

Fluri and Gall's classification of change types includes a variety of higher level changes that depend on additions, deletions and updates. Aardvark only supports a subset of these changes, primarily those that modify the execution and functionality of a program. We use these changes to classify the changes supported by Aardvark and discuss those it does not support.

Addition Fluri and Gall present a variety of changes that rely on the addition operation, including additional functionality, statement insert, parameter insert, and else-part insert [18]. Aardvark inherently supports all of these when diffing the source, however some of them may not directly appear in the trace and each type may cause updates, additions or have no effect in the trace (discussed in the next section). For example, the addition of a new parameter to a function will appear in a source code diff, however unless that parameter affects the calculation

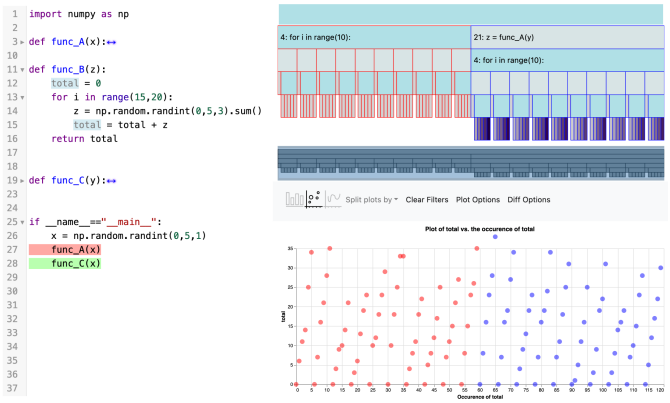

Fig. 3: The execution diff of a program after wrapping the initial function call to func_A in a call to func_C. Doing this changes the depth of the call to func_A, causing this call and all subsequent children to be marked as additions.

of a value or the execution of a function call or loop, the change will not appear in the trace. Aardvark highlights added lines of code using green highlighting, as shown in Fig. 1 (A) line 18.

Deletion Fluri and Gall present a complementary set of changes that rely on the deletion operation, including removed functionality, statement deletion, parameter deletion, and else-part deletion. Again, Aardvark supports all of these when diffing the source but they may not directly appear in the trace. Deletions to the source may propagate to the trace as deletions, updates, or not at all. Aardvark highlights deleted lines of code using red highlighting, shown in Fig. 1(A) line 17.

Update The majority of update changes that Aardvark supports fall under the "statement update" type (e.g., updated parameter/variable values) with the rest falling under "condition expression change". The trace likely will not reflect the modifications themselves (e.g., the trace likely does not directly capture an updated parameter) but may capture residual changes caused by the update. Thus, source code updates may result in any type of trace change. Rather than having an explicit encoding for line updates, Aardvark treats them as an addition and deletion, such that it highlights the original line of the update as deleted and the new version as added, shown in lines 14 and 15 of Fig. 1(A).

Unsupported changes Some changes defined by Fluri and Gall do not have any effect on the trace and, as a result, Aardvark does not support them. The renaming of any program components, e.g., parameters, functions, variables, etc., will not have an effect on the behavior of the program. The source code diff highlights this change but, in the back-end, the originally named component still maps to the newly named component and they are considered the same when running the trace diffing algorithm.

Additionally, we do not support changes to the accessibility of a component (e.g., a private or public variable), the type of variables, modifications to object state, or changes to the inheritance structure of classes. While these changes have significant effects in object oriented programs, they do not inherently have a significant effect in Python programs or on execution traces.

Last, we do not precisely support modifications that alter the depth of components. For example, if, in the original program, function A calls function B and someone modifies the program so that function C calls function A which then calls function B, the call to A is now at a new depth and will not be matched with the call to A in the original trace. Aardvark will mark the calls to A and B in the original trace as deleted and the calls to A and B in the new trace as added. We show an example of this in Fig. 3.

4.2 Trace Changes

Addition Additions to the trace result from a variety of source changes including: added function calls, increased iterations in a loop, update of conditional statement, etc. We define an addition to a trace as any node (function call, loop, assignment, etc.) that does not have a corresponding node at the same depth in the previous execution. Note, if a function call exists in the prior version but the call appears at a different depth or from a different parent than in the original execution, it will be marked as an addition. Fig. 3 shows an example of this.

Deletion Deletions in the trace result from a complementary set of source changes to those that cause additions in the trace including: a removed function call, fewer loop iterations, etc. Consequently, we define a deletion in the trace as any node in the original execution that does not have a corresponding node at the same depth in the new version of the trace.

Update In traces, we limit the definition of update to only include updated variable/expression values. When two matched instances of a variable differ in value, we mark that value as updated. These updated values stem from program changes such as the modification of a calculation statement or parameter, the insertion/deletion of a calculation step or simply the use of a different randomly generated value.

We do not consider "updates" to function calls or loops in our traces. A clear definition of what it means to update a function call does not exist in this setting. Likewise, the definition of "updating" a loop could mean a variety of things, such as changing the number of iterations, changing the values iterated over, etc. However, those updates would likely spawn subsequent trace changes that our definitions encompass.

Mapping between source changes and trace changes Creating a well defined mapping between any change in the source to corresponding changes in the trace proves difficult without enumerating all possible changes of each type. We identify situations in which each type of source change leads to each type of trace changes. For example, adding a function or loop to the source creates additions in the trace. Meanwhile, adding new expressions that impact a variable's calculation causes updates in the trace (i.e., different values of the variable). Finally, adding new conditional statements that reduces the how often a function is called causes deletions in the trace. Similar cases exist for source updates and deletions as well.

5 Aardvark's Design

Aardvark's design consists of two primary challenges: (1) identifying the differences between two consecutive traces, as well as the source code, and (2) designing visualizations to facilitate comparison of the values from the two traces. In this section, we discuss each extension in detail. We implement Aardvark's design in a prototype Python tool as well as a Jupyter notebooks library. Note, with computational notebooks, Aardvark currently supports diffing across two complete workflows (e.g., two runs of the notebook) rather than within a single workflow (see Sec. 7 for more discussion).

5.1 Source and Trace Diffing

Aardvark incorporates two forms of program diffing: diffing the source code and diffing the resulting trace. Aardvark uses source diffing to seed the tracer with hints about where changes may occur to help it identify differences in the traces. In this section, we discuss how Aardvark performs each of these diffs.

Source Diffing As previously mentioned, Aardvark uses Gumtree [15] to create the source code diff by diffing the AST's of the original and modified version of the program. This allows Aardvark to highlight the textual differences between the two versions of the source code, as shown in Fig. 1 (A).

Additionally, Aardvark uses these changes to seed the tracer with areas of the program that may cause changes in the trace. Gumtree identifies nodes of the AST that are added, deleted or updated. Because Aardvark already parses and traverses the AST to transform the script for tracing, it easily marks the AST nodes affected by the source changes, specifying the type of change, as defined in Sec. 4.1. When Aardvark traces the script, this information enables it to mark areas of the trace that may have changed. This in turn provides hints to the trace differ for identifying differences between two consecutive traces.

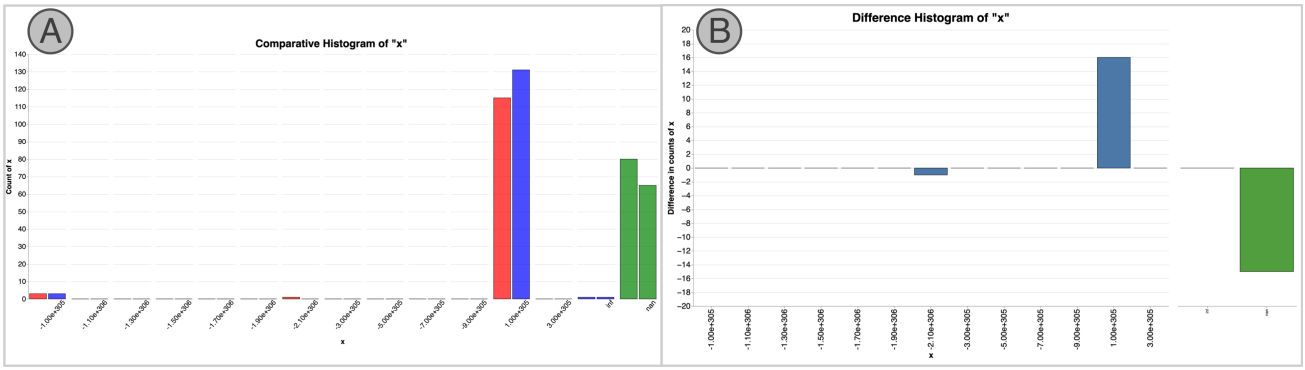

Fig. 4: The comparative histogram views. (A) shows the superposition view. The left bars (colored red for all quantitative values) represent the count from the original version and the right bar (colored blue) represent the count from the modified version of the program. (B) shows the explicit encoding of the difference of counts between the two traces. The count represents the count of the original version subtracted from the count of the modified version.

However, it alone is insufficient for identifying all changes for two primary reasons.

First, not all changes in a trace result from changes to the corresponding node in the AST. For example, consider a variable that sets how many iterations a loop runs. Changing the value of that specific variable will cause the loop to execute more or fewer times. However, the loop node of the AST will remain unchanged. Thus, simply diffing the AST's and marking the changed nodes would not mark the loop nodes as new/changed, just the iteration variable. Thus, the AST diff to mark changes in the trace would not mark all changes.

Second, to use the AST diff to match the old trace with the new trace, Aardvark would need to re-run the original trace to mark the nodes that we know will change, thus allowing it to match them with the nodes in the new trace. However, we do not want to re-run the original trace because if the script contains any randomness, this may change the tracked values from the original trace. As a result, AST diffing only marks nodes in the new trace and does not provide means to link the two traces together. Thus, while AST diffing makes sense for identifying the differences in the two versions of the source code, we need an additional algorithm for diffing the resulting traces.

Trace Diffing Using the hints provided by the source diffing, Aardvark performs trace diffing to identify how these changes actually affect the execution, as captured by the trace and merge the two traces into a single combined trace. The trace differ operates in two steps. First, using the output from the source diffing, it creates a mapping, M, between the two versions of the program to enable easy identification of matching nodes in the trace. With this mapping, the trace differ moves to the second step of aligning the traces and identifying the differences.

The traces inherently have a tree structure, where each child node was executed from within the parent node. Because of this, to identify the differences in the traces, Aardvark can use a tree diffing algorithm. Aardvark employs a basic tree diffing algorithm, adapted from [7]. The algorithm operates as follows. First, the algorithm takes in two traces, V1 and V2, and traverses them simultaneously with the goal of labeling each node in the trace with one of the types in Sec. 4.2 or as "unchanged". Starting with the root nodes, Aardvark inspects the children of both nodes and identifies pairs of matching children in V1 and V2 as possible. Aardvark considers two nodes to match if 1) the mapping, M, identifies them as corresponding to the same source code and 2) they are in the same relative order with preceding paired child nodes. If Aardvark identifies the nodes as a match, by default it marks them as "unchanged" because the node exists in both traces. However, in the case of variable and expression nodes, it performs an additional check to determine if the recorded values changed. If the nodes match but the recorded values differ, it identifies them as "updated", rather than "unchanged". All remaining, unmatched children were either added in the new trace or removed from the original trace. As such, Aardvark marks the remaining unmatched nodes in V1 and all of their descendants as "deleted" and the remaining nodes in V2 and all of their descendants as "added".

After pairing any matching children together the algorithm recurses on each pair and repeats this process. Once the algorithm recurses on the paired children, the V1 and V2 are combined and added to a combined trace. Unmatched nodes are added to the combined trace with empty attributes in place of a matched node. The combined trace maintains the relative ordering of the two traces, such that each trace could be recreated by removing all information from the other trace.

5.2 Comparative Visualizations

Aardvark designs the visualizations to support comparative visualizations and interactions. Gleicher et al. [21] provides a taxonomy of different comparative visualizations. They present three types of comparative visualization for two datasets: juxtaposition, superposition, and explicit encoding of relationships. Juxtaposition provides separate but adjacent plots in the aligned space (i.e., with aligned scales), that allow people to view each individually as well as facilitate comparison of the two datasets. Superposition combines the two datasets into a single plot, fully displaying both datasets in the same space. Explicit encoding directly encodes the relationship between the two datasets, rather than presenting them as two disjoint datasets. For each type of comparative visualization supported, Aardvark provides a view using each of the three types with controls to toggle between them.

Generalized Context Tree The comparative generalized context tree (GCT) needs to show the two execution structures, while highlighting the differences between the two executions. When creating the comparative GCT we must ensure that corresponding parts of the execution align horizontally in the plot. The diffing algorithm merges the two traces into a single, combined trace. Aardvark combines each pair of matched nodes into a single node, maintaining the relative order for each trace. We use the combined trace to generate the comparative GCT. We present three comparative GCT views based on Gleicher et al.'s taxonomy [21], two juxtaposition views and a superposition view.

The first juxtaposition view presents side by side GCTs, one for each trace. In the GCT for the original version, we draw all nodes that are updated, deleted, or unchanged. These nodes correspond to those that existed in the original trace. However, we leave gaps for nodes added in the second trace. This ensures that the two GCTs align correctly. We draw the GCT for the modified version in the same manner, leaving gaps for deletions from the original trace. While this view gives the entire view of both traces, it suffers when locating corresponding positions in the two traces. The second juxtaposition view presents a single GCT visualization for both traces, as shown in Fig. 1. The single view does not combine both traces, but rather, roots the GCTs at the same block and builds the original trace downwards and the trace from the modified version upwards. Doing this instead of presenting two separate views enables faster comparison of the two traces. Again, we use the combined trace to build the GCTs, leaving gaps for additions and deletions as necessary. People no longer have to shift back and forth between two plots and instead only need to find nodes at the same horizontal position and vertical depth.

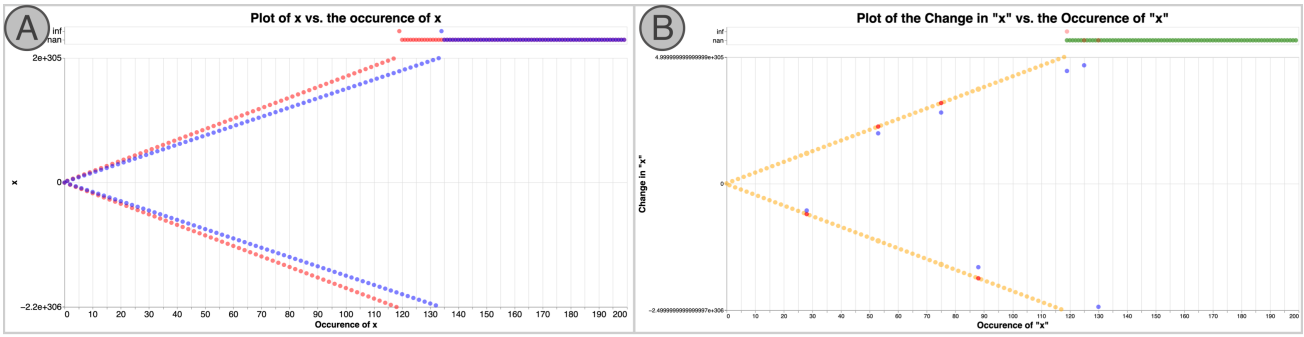

Fig. 5: The comparative scatterplot views. For the sake of space, we exclude the juxtaposition view and only show superposition and explicit encoding views. The x-axis represents the occurrence (or instance) in the execution of the shown value. (A) shows the superposition view. We color points from the original version red and points from the modified version blue. (B) shows the explicit encoding of the difference between the occurrence in the modified execution and the original occurrence. Gold points represent the difference between the two occurrences. Selecting a point shows the actual values from both versions (in red for the original and blue for the modified version).

The superposition view presents a single GCT containing both traces, shown in Fig. 3. Unlike the second juxtaposition view, this view only builds downward. It builds the entire combined trace, highlighting the additions, deletions, and updates. While this view presents all of changes to the trace, it does not as easily illustrate the individual GCTs. While Aardvark offers all three versions of the GCT, our examples primarily use the bi-directional juxtaposition view as it provides the individual GCTs for each trace while aligning for quick comparison and highlighting the changes.

Histogram Aardvark provides three histogram views - a side-by-side (juxtaposition) view, a superposition view, and a difference (explicit encoding) view. For the sake of space, we only provide examples of the comparative and difference histograms, shown in Fig. 4.

We designed three comparative histogram views. First, the juxtaposition view places a histogram of each dataset adjacent to each other. We ensure that the bins and y-axis align in both plots to enable easier identification of similarities and differences. This view offers the advantage of allowing people to view the whole dataset, without interrupting, visual clutter. However, it requires mental overhead of matching positions for comparison in two disjoint views.

The superposition view shows the two histograms side by side on the same axis. As shown in Fig. 4-A, there are two bars for each bin, one for the original version and one for the modified version. In contrast to the juxtaposition view, people can easily compare matching bars without mental overhead. However, the combined histogram interrupts the global view of each individual dataset, making it more difficult to get the entire view.

Last, the difference view (Fig. 4-C) directly encodes the difference in the frequencies between the new trace and the original trace. Each bar represents the change from the original frequency to the new frequency. As a result, some differences may end up negative, requiring the histogram to account for negative values. This view strays the farthest from a traditional histogram, as values cannot have negative frequencies. This view allows people to easily and quickly see how the frequencies changed between the two versions. However, on the other hand, people lose the context of the actual distributions of the values.

Scatterplot Aardvark also provides three scatterplot views, complementary to those in the comparative histograms. Fig. 5 shows the scatterplot views, again without the juxtaposition view for space. For scatterplots we must ensure that all corresponding instances are aligned on the x-axis and account for instances that correspond to "added" or "deleted" nodes that do not have a matching node in the other trace.

Much like the juxtaposition histogram view, the juxtaposition scatterplot view simply plots two adjacent scatterplots, one of each version of the data. While these give a clear view of each dataset, they require additional effort to match instances together and inspect individual differences, particularly if there exist unmatched instances.

The superposition view (Fig. 5-A) plots both sets of points on the same axes and colors the points depending on which version they belong to, red for the original version and blue for the modified. Aardvark plots matching instances, as identified during diffing, at the same location on the x-axis. This plot allows easy comparison of matching instances across the traces by inspecting their horizontal positions. However, in some cases, it suffers from clutter and overlapping of values that do not significantly change, making some comparisons difficult.

The explicit encoding view directly plots the difference between the two versions for each instance. For each paired instance, it calculates the difference between the new version and the original version, and plots that point. For points from the original version that do not have a matching instance, we simply treat their matching instance as 0. Thus for an unmatched value x from the original version, we plot it as $-x$. Similarly, we plot unmatched instances from the new version as their true value (i.e., if x is in the new trace and does not have a matching instance, we plot it as x). While this plot reduces the clutter of the superposition plot, it loses the context of the actual values from the traces. To bring back some context, when someone selects a point we plot the corresponding points from both traces to give them context of the original values. Fig. 5-B shows the explicit encoding scatterplot.

Aardvark currently only supports scatterplots with a single value, plotted by occurrence in the execution. Two variable scatterplots do not have an inherent ordering of the points, making it difficult to visually cue people towards matching pairs of points. Only the juxtaposition view, with accompanying interactions that link the two plots could facilitate the comparison of two variable scatterplots. However, even that view does not make the differences visually salient.

6 Usage Scenarios

To illustrate the usage of Aardvark, we present two usage scenarios. The first usage scenario mimics a usage scenario presented in the Anteater paper, with the goal of illustrating how comparative visualizations further ease the debugging and understanding process. The second usage scenario illustrates the use of Aardvark for exploratory debugging in a computational notebook from an active research project.

6.1 Gradient Descent

In this scenario, we will inspect the effects of changes on a misbehaving gradient descent program. This example comes from a question on Stackoverflow [1]. In the question, the presented gradient descent program returns NaN's instead of minimized values. To understand and fix the problem, we inspect the effects of parameter changes using Aardvark. Additionally, we contrast with the views from the original Anteater method to demonstrate the need for comparative views.

To begin, we track the values being optimized, "x" and "x1". We inspect the Aardvark views for the initial script, shown in the top row of Fig 6. From these views we see that the optimized values oscillate between increasingly large positive and negative values, before reaching infinity which then causes the NaN's.

While we see the problem, we do not know precisely how to fix it. We adjust the main parameter in the script, the training rate. We lower the training rate, re-run the script and observe the changes. At first glance in the single views, it seems that this may not have had any effect

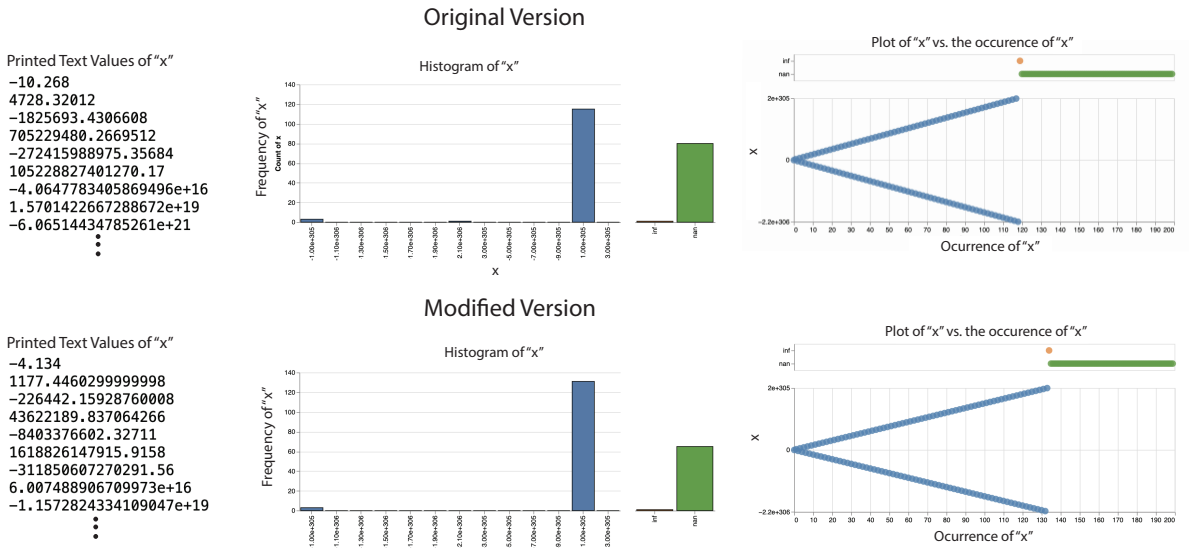

Fig. 6: The Anteater views for gradient descent along with example print statement debugging printouts. The top row shows the visualizations for the original execution. The bottom row shows the visualizations for the execution after lowering the training rate. Note, the scatterplots use a symmetric log scale. In this example, the differences between the two script executions may not be immediately apparent when comparing the two instances side-by-side, and even less so when viewing them individually, in succession.

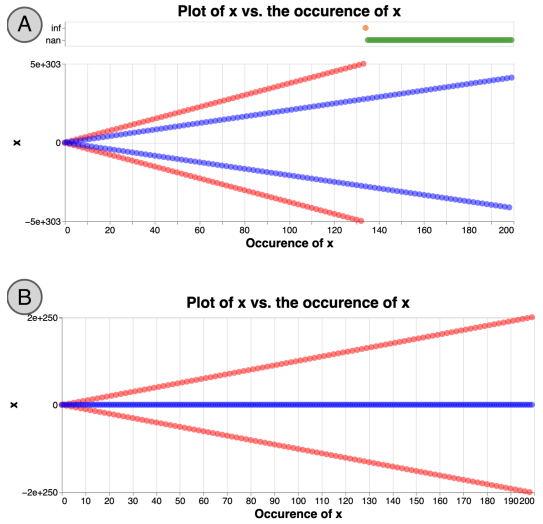

Fig. 7: The Aardvark scatterplot after dropping the training rate twice more. Note, the plots use a symmetric log scale. In (A) we see that, while the oscillation continues, we no longer have any NaN's. In (B), after dropping the training rate one more time, we see that the solution converges, which, because of the log scale, appears as a line at zero.

as shown in the bottom row of Fig. 6. However, to compare directly with the previous instance, we enable the comparative visualizations of Aardvark, shown in Fig. 4 and Fig. 5. The histograms in Fig. 4 show that the number of NaN's decreased after lowering the training rate and the scatterplots in Fig. 5 show that the oscillation narrowed which resulted in fewer NaN's. From these plots we see that, while we did not completely fix the problem, lowering the training rate still improved the optimization. We continue to drop the training rate, shown in Fig. 7 and see from the Aardvark views that continuing to lower the training rate further improves the optimization until it finally converges.

Note, were we to only use traditional methods or Anteater, it would have been more difficult to determine whether lowering the training rate improved the optimization. Fig. 6 shows the beginning of the text printouts as well as the singular visualizations as in Anteater. Typically, people would inspect these one after another, in isolation, as part of their workflow rather than in two side by side debugger instances. In isolation, the differences are particularly challenging to identify. Even side by side differences may not be immediately apparent. However, with Aardvarks comparative visualizations it is immediately apparent how the values being optimized differed across the two instances.

6.2 Interactive Dimension Reduction

In this scenario, we perform exploratory debugging in a computational notebook that performs interactive dimension reduction. The interactive dimension reduction works by allowing people to specify clusters in the 2D space and then passing that information to the dimension reduction by optimizing weights on the high dimensional features. It aims to weight the high dimensional features such that the pairwise high dimensional distances reflect the specified pairwise 2D distances. Our task is to explore the number of datapoints in each class we need to use when defining the clusters. We aim to understand how well our optimization matches the task and continually works toward a better clustering. To do this, we measure and track the quality of the clustering at each iteration of the weight optimization process. In the ideal scenario, the clustering quality would converge to a good solution (higher value), as the weight optimization converges.

Fig. 9 shows the Aardvark views for the clustering quality when defining the clusters with two points per cluster. We see that, with two points,the clustering quality does not stabilize or converge as the learning process converges. While this returned a reasonable result, this does not behave precisely as we would expect. We expect the learning process to converge towards a good clustering as the learning process converges. To further evaluate our optimization, we re-run the script, specifying three points per cluster. Fig. 8(A) shows the comparative Aardvark views after re-running the script with three points per cluster. We see that while the optimization took a bit longer to converge and the clustering overall improves, the DR quality still does not really stabilize. This suggests that our learning process may match the task but requires more points per cluster to stabilize. Fig. 8(B) shows the comparative visualizations after another instance with four points per cluster. Now, we see that the DR quality seems to largely converge towards a reasonable clustering, although not as good as in previous iterations. For good measure, we run another instance with 5 points per cluster, shown in Fig. 8(C) where we see that the DR quality continues to improve and stabilize as the optimization converges.

7 Discussion

Trace Diffing Algorithm The diffing algorithm has limitations, its design assumes more simplistic nodes where all nodes are the same type but may differ in value. However, program traces contain a variety of node types, each of which have a value. Therefore, when looking

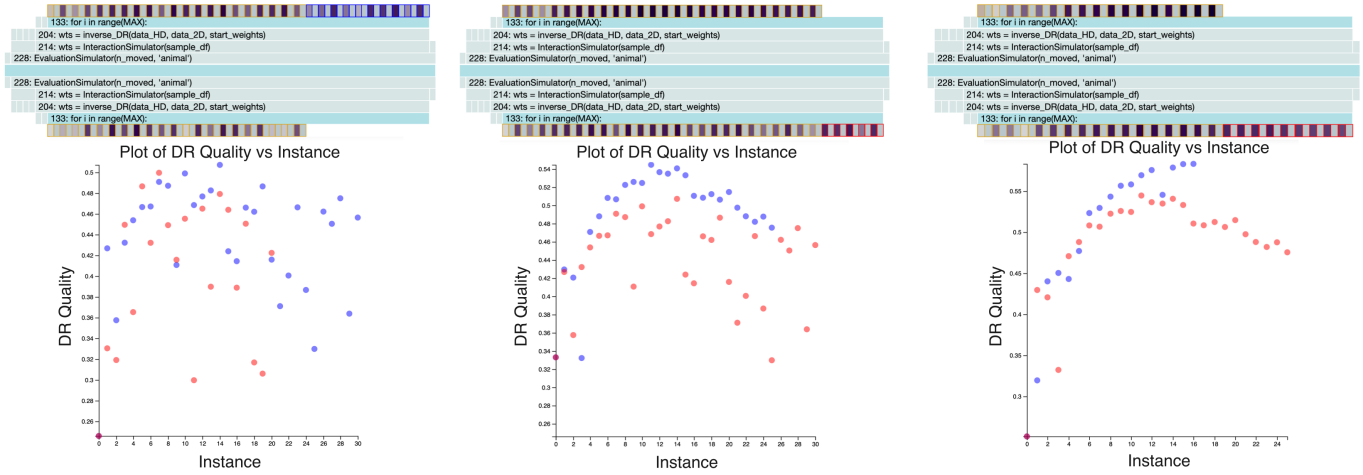

Fig. 8: The comparative visualizations when specifying three, four and five points per cluster. (A) show the comparative visualizations between two and three points per cluster. From the GCT, we see that it took longer for the DR to converge but the quality of the clustering does not seem to have greatly improved. (B) shows the comparative visualizations from three to four points per cluster. We see that the clustering largely converges, although the final clustering is of lower quality than previous iterations. (C) shows the comparative views from four to five points per cluster. Finally, we see that the clustering quality seems to converge with the optimization and takes fewer iterations to do so.

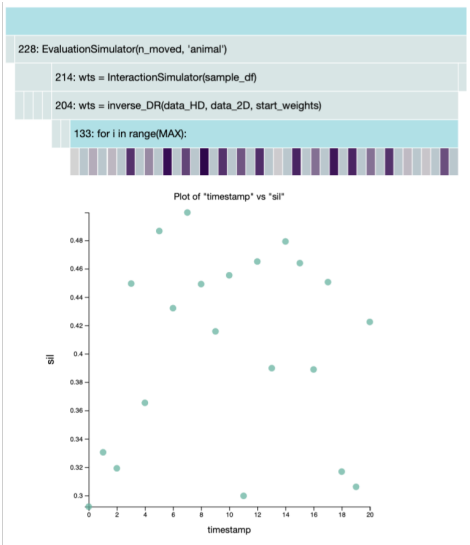

Fig. 9: The initial visualizations for the interactive DR with two points per cluster. Although the optimization converged, the plot shows that while the optimization converged, the clustering quality does not.

for diffing the execution trees, the algorithm may overlook certain commonalities due to slight structural differences. However, the diffing algorithm used in this work is only one possible example of a diffing algorithm. Due to the modular nature of the system, a different diffing algorithm can easily be inserted into the system. The exploration of more sophisticated diffing algorithms is left for future work.

Changes in Computational Notebooks The current extension to notebooks relies on consecutive traces of the entire workflow. However, analysts do not necessarily re-run the entire notebook with every change. While Aardvark captures these changes in a single trace, it does not currently provide comparative views of repeated portions in a single workflow. We explored the option of identifying repetition within a single notebook workflow, however, this relies on the ability to track and identify the specific cells that were run, such as through a cell id, to match them throughout the execution. We were unable to find a consistent way to track which specific cell was executed, short of inferring based on similarities between executed code. Future work is still needed to further explore methods for identifying and visualizing repeated cell executions within a single notebook workflow.

Supported Program Changes Aardvark is designed to support minor changes. Minor changes include those that primarily alter program values and only minorly alter the execution structure. Aardvark will still work on more significant changes, however as the changes become larger and fewer parts of the executions align, the visualizations become more complex and less readable. Additionally, Aardvark assumes nothing about a program other than that it is written in Python. It does not assume anything about the structure of the program or even the type of changes made to the program. As a result, Aardvark limits the types of questions people can ask when performing comparative tasks. Specifically, Aardvark only allows us to ask general comparative questions, such as "how does the behavior of this version of value x compare to the behavior of this other version of value x." It does not enable us to ask deeper, more program dependent questions such as "what is the influence of value x on dependent variable y". To address these questions, further work is needed to explore the types of assumptions we can make about analysis scripts and the types of comparative questions enabled by these assumptions.

8 Conclusion

In this paper, we presented Aardvark, a comparative trace-based visual debugging method for visualizing the effects of changes to analysis scripts. Aardvark builds on the tracing infrastructure of a prior trace-based visual debugging method, Anteater, to trace two versions of an analysis script, identify the changes between the two traces, and present these changes through interactive visualizations. Additionally, we presented two usage scenarios to demonstrate Aardvark's ability to illustrate the effects of changes in analysis scripts. Aardvark presents a promising first step towards automatically illustrating the differences across multiple versions of analysis scripts to help people better understand their analyses.

Acknowledgments

This work is partially supported by the NIST Graduate Student Measurement and Engineering Fellowship, through a grant with the GFSD, and the National Science Foundation under Grant # 2127309 to the Computing Research Association for the CIFellows 2021 Project.

Disclaimer

This work represents an official contribution of NIST and hence is not subject to copyright in the US. Identification of commercial systems in this paper are for demonstration purposes only and does not imply recommendation or endorsement by NIST.

REFERENCES

[1] Gradient descent implementation in python returns nan. https://stackoverflow.com/questions/15211715/gradient-descent-implementation-in-python-returns-nan, 2013. Last visited on 2020-04-30. 6

[2] Admin. Intel trace analyzer and collector, Oct 2019. 2

[3] A. Alaboudi and T. D. LaToza. Edit-run behavior in programming and debugging. *arXiv preprint arXiv:2109.02682*, 2021. 1

[4] B. Alper, B. Bach, N. Henry Riche, T. Isenberg, and J.-D. Fekete. Weighted graph comparison techniques for brain connectivity analysis. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pp. 483–492, 2013. 2

[5] B. Alsallakh, P. Bodesinsky, A. Gruber, and S. Miksch. Visual tracing for the eclipse java debugger. In *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 545–548. IEEE, 2012. 2

[6] M. Angelini, G. Blasilli, L. Borzacchiello, E. Coppa, D. C. D'Elia, C. Demetrescu, S. Lenti, S. Nicchi, and G. Santucci. Symnav: Visually assisting symbolic execution. In *2019 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pp. 1–11. IEEE, 2019. 2

[7] L. Burgess-Yeo. F# tree diff algorithm, Jan 2020. 5

[8] G. Canfora, L. Cerulo, and M. Di Penta. Tracking your changes: A language-independent approach. *IEEE software*, 26(1):50–57, 2008. 2

[9] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. *Acm Sigmod Record*, 25(2):493–504, 1996. 2

[10] Y.-P. Cheng, C.-Y. Ku, W.-C. Pan, C. Yang, and T.-S. Lin. Toward arbitrary mapping for debugging visualizations. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pp. 605–608. ACM, New York, NY, USA, 2016. doi: 10.1145/2889160.2889167 2

[11] B. Cornelissen and L. Moonen. Visualizing similarities in execution traces. In *Proceedings of the 3rd Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pp. 6–10, 2007. 2

[12] V. Dashuber and M. Philippsen. Trace visualization within the software city metaphor: Controlled experiments on program comprehension. *Information and Software Technology*, 150:106989, 2022. 2

[13] G. Dotzler and M. Philippsen. Move-optimized source code tree differencing. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 660–671. IEEE, 2016. 2

[14] S. Duszynski, V. L. Tenev, and M. Becker. N-way diff: Set-based comparison of software variants. In *2020 Working Conference on Software Visualization (VISSOFT)*, pp. 72–83. IEEE, 2020. 2

[15] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 313–324, 2014. 2, 4

[16] R. Faust, C. Scheidegger, K. Isaacs, W. Z. Bernstein, M. Sharp, and C. North. Interactive visualization for data science scripts. In *2022 IEEE Visualization in Data Science (VDS)*, pp. 37–45, 2022. doi: 10.1109/VDS57266.2022.00009 1, 2, 3

[17] S. Fitzgerald, R. McCauley, B. Hanks, L. Murphy, B. Simon, and C. Zander. Debugging from the student perspective. *IEEE Transactions on Education*, 53(3):390–396, 2009. 1

[18] B. Fluri and H. C. Gall. Classifying change types for qualifying change couplings. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pp. 35–45. IEEE, 2006. 3

[19] B. Fluri, M. Wursch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, 33(11):725–743, 2007. 2

[20] D. M. German, A. E. Hassan, and G. Robles. Change impact graphs: Determining the impact of prior codechanges. *Information and Software Technology*, 51(10):1394–1408, 2009. 2

[21] M. Gleicher, D. Albers, R. Walker, I. Jusufi, C. D. Hansen, and J. C. Roberts. Visual comparison for information visualization. *Information Visualization*, 10(4):289–309, 2011. 2, 5

[22] M. Graham and J. Kennedy. A survey of multiple tree visualisation. *Information Visualization*, 9(4):235–252, 2010. 2

[23] V. Grigoreanu, J. Brundage, E. Bahna, M. Burnett, P. ElRif, and J. Snover. Males? and females? script debugging strategies. In *International Symposium on End User Development*, pp. 205–224. Springer, 2009. 1

[24] M. Hashimoto and A. Mori. Diff/ts: A tool for fine-grained structural change analysis. In *2008 15th working conference on reverse engineering*, pp. 279–288. IEEE, 2008. 2

[25] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pp. 234–245, 1990. 2

[26] D. Jackson, D. A. Ladd, et al. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM*, vol. 94, pp. 243–252, 1994. 2

[27] S. Lehnert, M. Riebisch, et al. A taxonomy of change types and its application in software evolution. In *2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems*, pp. 98–107. IEEE, 2012. 3

[28] A. V. Miranskyy, M. Davison, R. M. Reesor, and S. S. Murtaza. Using entropy measures for comparison of software traces. *Information Sciences*, 203:59–72, 2012. 2

[29] T. Munzner. *Visualization analysis and design*. AK Peters/CRC Press, 2014. 2

[30] H.-G. Pagendarm and F. H. Post. *Comparative visualization: Approaches and examples*. Delft University of Technology, 1995. 2

[31] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005. 3

[32] D. Rozenberg and I. Beschastnikh. Templated visualization of object state with vebugger. In *2014 Second IEEE Working Conference on Software Visualization*, pp. 107–111. IEEE, 2014. 2

[33] R. Suzuki, G. Soares, A. Head, E. Glassman, R. Reis, M. Mongiovi, L. D'Antoni, and B. Hartmann. Tracediff: Debugging unexpected code behavior using trace divergences. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 107–115. IEEE, 2017. 2

[34] S. Taheri, I. Briggs, M. Burtscher, and G. Gopalakrishnan. Difftrace: Efficient whole-program trace analysis and diffing for debugging. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–12. IEEE, 2019. 2

[35] J. Trümper, J. Döllner, and A. Telea. Multiscale visual comparison of execution traces. In *2013 21st International Conference on Program Comprehension (ICPC)*, pp. 53–62. IEEE, 2013. 2

[36] I. Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985. 1

[37] S. Voigt, J. Bohnet, and J. Dollner. Object aware execution trace exploration. In *2009 IEEE International Conference on Software Maintenance*, pp. 201–210. IEEE, 2009. 2

[38] J. Woodring and H.-W. Shen. Multi-variate, time varying, and comparative visualization with contextual cues. *IEEE transactions on visualization and computer graphics*, 12(5):909–916, 2006. 2

[39] J. Zhao, Z. Liu, M. Dontcheva, A. Hertzmann, and A. Wilson. Matrixwave: Visual comparison of event sequence data. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pp. 259–268, 2015. 2