Anteater: Interactive Visualization of Program Execution Values in Context

Rebecca Faust, Katherine Isaacs, William Z. Bernstein, Michael Sharp, and Carlos Scheidegger

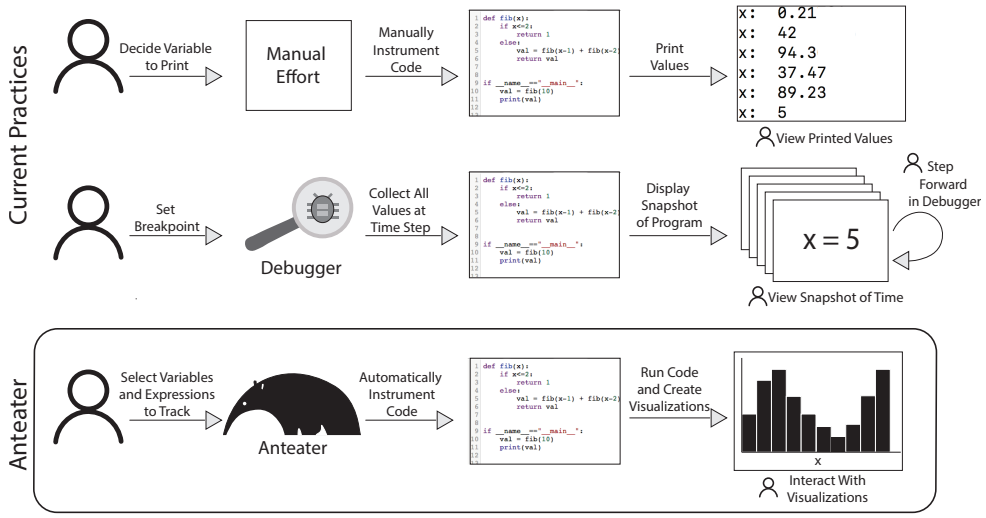

Fig. 1. A programmer investigates a bug in their code. One common practice (top row) is to instrument the program manually to collect suspicious variables (here, x), and print their values. Manual instrumentation, however, is itself repetitive and error-prone. Another common practice (second row) is to use a debugger to stop the execution of the program and view each individual value assignment of x, providing a precise, but narrow, one-at-a-time view of the values. Anteater (bottom row) automatically instruments the code to track variables along with the context of their execution. It presents the programmer with interactive visualizations providing a global view of values, enabling easy detection of erroneous values as well as interactions that narrow down the views to specific values.

Abstract—Debugging is famously one the hardest parts in programming. In this paper, we tackle the question: what does a debugging environment look like when we take interactive visualization as a central design principle? We introduce Anteater, an interactive visualization system for tracing and exploring the execution of Python programs. Existing systems often have visualization components built on top of an existing infrastructure. In contrast, Anteaters organization of trace data enables an intermediate representation which can be leveraged to automatically synthesize a variety of visualizations and interactions. These interactive visualizations help with tasks such as discovering important structures in the execution and understanding and debugging unexpected behaviors. To assess the utility of Anteater, we conducted a participant study where programmers completed tasks on their own python programs using Anteater. Finally, we discuss limitations and where further research is needed.

Index Terms—Interactive Visualization, Program Traces

1 Introduction

In this paper, we tackle the following questions: **when we take interactive visualization principles as a driving concern, what novel designs are possible for debugging systems? Can interactive visualization offer unique benefits for program debugging?** Debugging is time consuming, and current practice often involves stepping through debuggers, logging statements, or searching through source code, either manually or with a code browsing tool [25]. Traditional debuggers require programmers to set breakpoints at which they inspect the program state, stepping through its line-by-line operation. Tiarks et al. [39] observed that programmers experience difficulties in choosing breakpoint locations, forgetting analysis details while navigating the code.

- *Rebecca Faust, Katherine Isaacs, and Carlos Scheidegger are with the HDC Lab, Department of Computer Science, University of Arizona. E-mail: {rjfaust, kisaacs, cscheid}@email.arizona.edu*
- *William Z. Bernstein and Michael Sharp are with NIST. E-mail: {wzb, michael.sharp}@nist.gov*

Consider the traditional value proposition of data visualization. Visualization practitioners now have a well-defined set of principles to drive the design, development, and testing of interactive visualization software [6, 10, 34]. In contrast to inspecting datasets serially, one element at a time, well-designed visual encodings can provide a richer, faster, and more global views of potentially important patterns. The same fundamental issue of serial inspection is present in traditional debugging. We therefore see a need for an exploratory debugging solution that provides more effective global views of values, providing debugging the same set of affordances that interactive visualization provides to exploratory data analysis.

Another common problem users face when programming is that of making sense of unfamiliar code [39]. Whether programs are pulled from the internet, handed off to others by a collaborator, or existing in a just-joined project, sifting through code to understand its execution and determining where to start working is challenging. Something as simple as identifying the dependencies of a variable can become a significant burden when the program is large. There is a need for facilitating a deeper understanding of the structure of a program's execution to assist programmers in exploring how functions, variables, and values depend

on one another.

In response, we present Anteater, a system for debugging and understanding programs designed with principles of interactive visualization as a driving concern. In taking a visualization-first approach, Anteater provides more informative overviews of a program's behavior while supporting interaction to dig deeper into the details of the execution. Anteater aims to reduce the effort required from a user by 1) automatically instrumenting programs to collect the values they want to inspect and 2) allowing them to browse values of interest easily throughout the entire executing, without resorting to a step-through debugger.

We present a prototype implementation in Python that traces a Python program to capture not only the execution structure but also values of interest in context of the execution. Anteater then presents this trace to the user through interactive visualizations. In summary, this paper contributes (i) a goals-and-tasks analysis [24] of the typical practice of program debugging, (ii) a description and prototype implementation of Anteater in Python, aimed at providing first-class interactive-visualization support to program debugging and understanding, (iii) a paired analytics evaluation of the prototype and its analysis, and (iv) case studies of real-world programs that show how our system compares to existing approaches.

2 Related Work

Literature Search We compare Anteater to work we have found in software engineering, user interface design, information visualization, and visual analytics. Specifically, we have searched the last 25 years of work related to visual debugging in the following venues: ACM ICSE, ACM CHI, ACM UIST, IEEE VIS, and the SoftVis symposium. The field of software visualization is large and we cannot hope to add every possible reference; we recommend both textbooks from Diehl and Stasko as starting points into the literature [13, 37].

Visual Debugging Many attempts have been made to leverage visualization principles to augment the debugging process. Some efforts add visualization options to breakpoint and step-through debuggers [9, 11, 14, 26, 27, 30, 32]. Others show task-specific information about the execution, such as an overview of the heap [2], the impact of resource utilization on control flow [28], object mutation [33], or runtime state and data structures of the program [38]. Generally, these tools present localized views that describe one particular state of the execution. Some tools provide additional context by allowing backstepping in the debugger or providing a history of the execution [16, 26, 30]. Aftandilian et al. [2] give a global view of the heap by taking snapshots throughout the program. Schulz et al. [33] provide a global view of object mutations; if the object is numeric, the global view shows the value behavior throughout the execution. Some tools give global views of value behaviors by introducing sparklines next to the line of source code defining the value [5, 19]. In contrast, Anteater displays global views that take the execution context into account. As we show in Section 7, this perspective can be particularly helpful in debugging scenarios.

Alsallakh et al. [3] created an Eclipse plugin that tracks specific tracepoints (equivalent to a breakpoint in a debugger) throughout a program's execution. Watchpoints can also be added to a field on which the tracer will track assignments. The tracepoint instances are visualized as individual line charts where interactions provide additional information about the program at that point and watchpoints are viewed as a step chart of the values over time. While the plugin's goals closely relate to those of our prototype, Anteater stands apart for two reasons. First, Anteater traces all calls and loops, rather than user-defined tracepoints, along with the values desired by the user. Second, Anteater presents all this information in a trace visualization with corresponding plots of the tracked values. This information can provide the context necessary to better understand why variables take on certain values.

The most similar tool to Anteater is Kang et al.'s [22] Omnicode. Omnicode provides run-time visualizations of program states, designed so that novice users build mental models about programs. Crucially, Omnicode visualizes values in a live coding environment which updates in real time. The primary visualization provided is a scatterplot matrix displaying plots for each variable over all execution steps. While Omnicode and Anteater have much in common, they were designed for different audiences (novices vs. general programmers) and thus support different types of programs. We compare Omnicode and Anteater directly in Section 8.

Trace Visualization Trace visualizations are often applied in support of understanding parallel programs [23, 35, 40]. Often, trace visualizations leverage icicle plots and flame graphs as the primary visual representation [7, 18, 23, 31, 40]. Anteater uses a visual encoding reminiscent of icicle plots and flame graphs in our generalized context tree. However, Anteater differs in its definition of *trace*. While these previous traces capture the calling structure of the execution, Anteater extends this to capture values of marked variables and expressions, as well as loop behaviors. This extension provides users with additional context for how values are reached; see Section 7 for a discussion of their utility.

3 Background

In this section, we discuss the current state of program debugging, understanding, and tracing, as well as the need for a system like Anteater.

3.1 Understanding Programs

The situation frequently arises where a programmer needs to understand code they did not write. Navigating unfamiliar source code is not an easy task and there are no tools designed specifically to facilitate such understanding. Several articles and blog posts exist to help programmers effectively read source code [12, 20]. Much of this help, however, consists of generic advice only. One article advises programmers to run the program first then find something the code is doing and trace it backwards. Another advises to use a debugger to set breakpoints and find a tool that allows for more intelligent navigation through the source code, such as Sourcegraph [36] and Pycharm [21].

From these posts, it seems that a programmer's best tools for understanding unfamiliar code are debuggers, print statements, and those designed for source code navigation. With only these tools to help understand code, programmers must expend considerable effort to carry out several burdensome tasks. Those tasks include 1) finding a starting point for navigation, 2) setting breakpoints to step through debuggers, and 3) making small changes to the program to better understand the impact on the execution.

3.2 Debugging Scenario

Programmer Patty has a bug in her code. She believes that the bug is occurring in a specific loop but cannot identify the root cause. Using a typical debugger, she sets a breakpoint at the beginning of the loop and runs the debugger. When the debugger reaches the breakpoint, she inspects values and takes a few steps through execution but does not yet see the bug. Patty continues the program until it hits the breakpoint again at the next iteration. She continues to step through each iteration of the loop but has little success in finding the bug.

After several iterations, Patty gives up on using the debugger and modifies the code with print statements. She prints the variable she believes is causing the bug and runs the program. Patty scans through the printed values, trying to find any erroneous values, but her loop has many iterations and she quickly gets lost in the print statements.

Her next idea is to write the values to a file and plot them. Patty first alters her source code to write the values to a file. She then writes a script that reads the file and plots the values. Now she can see the behavior of every instance of the value and pinpoint the incorrect values. With this information, Patty returns to the debugger and stops the program when it reaches the iteration containing incorrect values to find the root cause.

The scenario described above encompasses the typical ways programmers debug their programs [39]. While not every bug requires all of these methods, programmers typically use more than one of them. The fact that many programmers use a combination of independent debugging-methods when fixing their programs prompts the question: can we design a better debugger that 1) reduces the amount of manual instrumentation required, 2) gives the users greater control over the

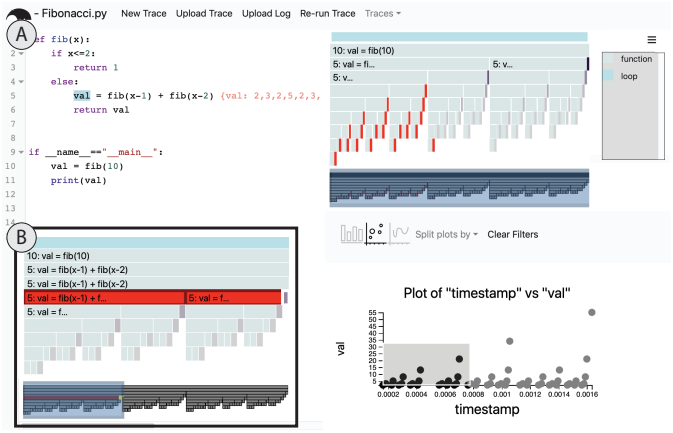

Fig. 2. An overview of Anteater on a recursive Fibonacci program, tracking the variable "val". (A) shows the initial view presented by Anteater. The generalized context tree, or icicle plot, shows the execution structure. The teal blocks represent function calls while the varying shades of purple represent the value of "val" at that instance. We can see the recursive calling structure of the Fibonacci function and can easily identify where it is repeating work. The plot shows a scatterplot view of the variable "val" over time. Brushing over the scatterplot highlights the corresponding instances in the generalized context tree and the context bar. The scatterplot shows repetitive patterns that indicate that Fibonacci is doing redundant work. In (B), we click on a block in the generalized context tree which causes its dependencies to be highlighted in red. This shows that the selected block, representing an instance of "val", depends on the prior two calls to the Fibonacci function.

values they see, and 3) provides them with a visualization option automatically? While various debugging tools address aspects of these problems, no debugger comprehensively addresses all of them. There exist tools that add visualizations on top of existing debuggers, such as Mirur [9]. However, many of them still operate within a snapshot of the program. Rather than visualizing the global behaviors of values, such tools help visualize more complex objects at a given timestep.

We designed Anteater to address these problems in a more comprehensive way by using the principles of interactive visualization. Fig. 1 gives an overview of how Anteater compares to standard debugging practices. Rather than presenting users a snapshot of everything at a single timestep, we present them with global views of targeted variables of interest with interactions to narrow down to specific values. In doing so, users can more easily discover patterns within the values they deem important. We pair these global views with a visualization of the execution structure that allows users to maintain context with the execution and inspect execution information surrounding the values. Fig. 2 shows an overview of Anteater.

If Programmer Patty had been using Anteater, she could have easily set Anteater to track the value she believed to be raising issues along with any other values that she believed to be potential roots of causation. Anteater would then trace her program and provide her with visualizations to help her identify the iterations where the value was incorrect. Patty could then filter down the execution tree to those iterations and inspect the rest of the values she tracked. With Anteater, Patty completes all of her debugging in one place using only a few interactions and requiring no manual instrumentation.

3.3 Tracing

Historically, program traces have captured performance information for a program's execution. They typically track function call and time spent within a function. Visualizations are then created to present this information to users. However, function calls and execution time only solve a subset of bugs. Primarily, program traces help identify functions that are taking more time than necessary or call structures that are not expected. They do little to help with finding bugs in the actual values being calculated by the program. Anteater expands the definition of a trace to include values from within the program. Capturing these values in the context of the trace, rather than collecting them in a separate file, provides important context to the programmer. A context that is lost when looking at the values in isolation.

4 Task Analysis

In this section, we discuss Anteater's goals that were selected based on both current standards of practice from literature and our own experiences with respect to program debugging and understanding.

G1: Identifying and understanding the source of unexpected execution behavior When programmers write and execute programs, they have some expectation of how their program should be behaving and what parts should be executing. As a result, one goal of debugging is to identify what is causing an execution to deviate from what the programmer expected.

G2: Identifying and understanding the source of unexpected values and trends Similar to G1, programmers typically have ideas about what values they should observe during the execution and thus desire to identify the root cause of unexpected values in the execution.

G3: Understanding an unfamiliar piece of code Frequently, programmers are tasked with understanding code written by someone else. Typically, this is no easy task and requires a significant amount of effort on the part of the programmer.

Under the framework of Lam et al. [24], the identifying portions of the goals G1 and G2, along with the entirety of G3, fall into the "Discover Observation" category. The understanding portions of G1 and G2, however, fall into the "Identify Main Cause" category. From these goals, we derived several sub-tasks.

T1: Track a variable or expression It is often useful to look at the values that a variable or expression take on to determine if it is behaving as-expected and to identify any erroneous values. This task supports G2 and G3.

T2: Identify what functions are called at runtime Often it is not clear from the static source code which functions will execute and when. However, identifying which functions are actually called during an execution is crucial for understanding how a program is operating. This task supports G1 and G3.

T3: Identify dependencies for a variable Understanding dependencies is crucial when trying to understand unfamiliar code. Identifying how a value is calculated, including the execution path required to complete the variable's calculation, allows programmers to better understand the underlying nature of the value in question. Such insight can lead to finding the cause of an unexpected value. This task supports G1 and G3.

T4: Identify interesting subsets of values Given a variable or expression, it is important to be able to identify the subset of values that correspond to interesting behavior. For example, if certain values indicate a failure in the program, they need to be identified so the surrounding values can be examined to understand the cause of the behavior. This task supports G1 and G3.

T5: Observe relationships between values When debugging a program, programmers often investigate relationships between variables. For example, if variable x changes, how does variable y change? These relationships may not be explicitly defined by the code, i.e., y may not directly depend on x. Uncovering such relationships contributes to program understanding. This task supports G1 and G3.

T6: Maintain context between runtime state and static source When trying to debug and understand a program, maintaining context with the actual code is critical. If the programmer is manually instrumenting print statements, they also must codify contextual information to derive insight, e.g., representing the location of a variable's modification. This task supports G1, G2, and G3.

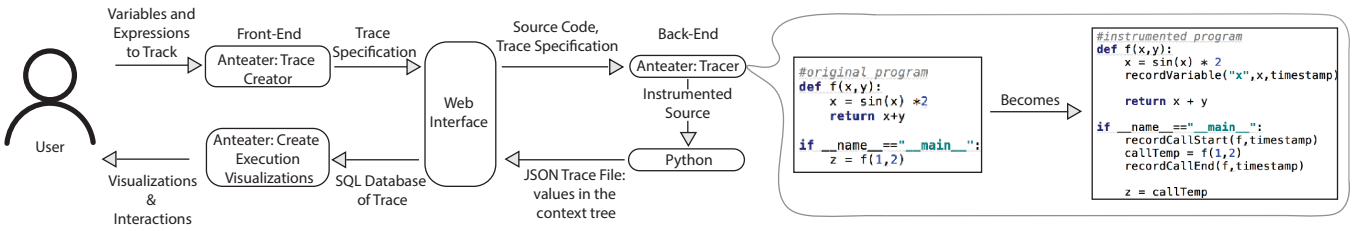

Fig. 3. An overview of the Anteater system. First a user chooses variables to track, defining the trace specification, using the Anteater interface. This trace specification is sent through the web interface to the python backend, along with the source code. The Anteater tracer then instruments the source code to collect execution information along with the specified values. The right side of the figure shows a simplified version of this instrumentation. After the code is instrumented, it is run using python to create the program trace. This trace is passed back through the web interface to the Anteater front end where it is visualized and presented to the user.

A system that supports all of these tasks needs to be able to track the execution structure of the program along with variable and expression values in the context of its execution. An execution trace is a natural fit for tracking the execution structure and can be modified to also collect values. Once this data is collected, it needs to be presented in a way that allows for easy navigation through it while supporting these tasks. We argue that visualization is the best way to present this information because it is known for providing overviews and context, highlighting relationships, and facilitating the filtering down to subsets of interesting information, all of which are needed to support these tasks. Anteater takes a visualization approach to program debugging and understanding that satisfies these goals through execution traces and visualizations. Anteater deals solely with single-threaded programs but we expect that this task analysis would need to be extended to satisfy our goals for multi-threaded programs.

5 Tracing Infrastructure and Data Organization

To support the goals and tasks defined in Sec. 4, an execution trace with accompanying variable and expression values must be collected. Anteater implements a tracer that automatically instruments the source code to collect the execution trace. Implemented in Python, the tracer relies solely on the Abstract Syntax Trees (AST) library to facilitate the transformation of the source code. While Anteater currently only works with Python programs, the same principles can be implemented in any language that has the ability to transform source code in a similar way. After transforming the source code, Anteater runs the program, creates the trace file, and organizes the data in a way that allows for easy creation of interactive visualizations. Fig. 3 illustrates how the system operates.

5.1 Tracing Programs

When a user chooses to create a trace, the Anteater backend is passed a trace specification containing a list of variables and expressions to track along with a list of functions and libraries to exclude from the trace (see Sec. 6 for additional detail). The tracer indexes through these lists and determines the scope in which each item resides to ensure that it only tracks/excludes the specified items. For example, if two functions both define variable x, the tracer will only track the one the user selected.

Once Anteater determines the scope of each item, the tracer uses the Python *ast* library to parse the source code into its AST. It then performs a series of traversals of the AST to collect information about the source code and transform the program to trace the execution and desired values.

In the first traversal through the AST, no transformations occur. Rather, Anteater collects information about functions, loops, and dependencies. For functions and loops, it collects the lines at which the function definition or loop begins and ends. This information enables more detailed linking between visualizations and source code. For dependencies, the tracer traverses through the code and, for each variable, stores functions and variables on which it directly depends (i.e., stores targets on the right side of an assignment statement). To find all dependencies for a variable, we access its dependency list, and, for each dependency in the list, we access their dependencies. This continues until Anteater builds a comprehensive list of all possible dependencies.

Once all of the static data has been retrieved from the source code, Anteater begins transforming it. An initial traversal through the AST transforms the code to separate all function calls from their respective expression statements and expand list comprehensions into `for` loops. Anteater pulls all function calls that do not stand alone out of their expressions and assigns them to a temporary variable that replaces the call in the original expression (e.g., $x = 2 * f()$ becomes $tempF = f()$; $x = 2 * tempF$). This allows Anteater to capture easily when and in what order functions are called.

Next, the tracer performs the main transformations to insert the instrumentation that collects the trace. As the tracer traverses the AST, it always pauses at assignment, call, and loop nodes. When it reaches an assignment node, it checks if the target variable needs to be tracked. If so, it inserts new nodes into the AST that record the value of the variable after assignment.

When the tracer reaches a call node, it first checks if the exclusion list includes the function. If not, the tracer generates AST nodes to record that the beginning of the call and inserts them before the call. The tracer then generates AST nodes that record when the call has returned and inserts them after the call. Because we move function calls into their own statements, a function call statement fully executes before the next function call starts. This allows all bookkeeping for a call to be completed before the next call executes. A simplified example of this transformation is shown in Fig. 3.

When the tracer reaches a loop, it creates a counter that counts the iterations of that loop and inserts new instrumentation to record the start of the loop. As it traverses the body of the loop, any time the tracer creates a new record, it records the iteration in which that record occurred. Tracking the iteration binds together groups of records in the trace and records that occurred in the same part of the execution. The tracer also checks if the iterator variables need to be tracked.

Lastly, the tracer transforms the program to record expressions. Expressions are more complicated because they could occur in a variety of nodes. As the tracer visits the nodes, it checks if the line containing the node also contains a tracked expression. If it does, the tracer extracts the expression from the line, assigns it to a temporary variable, and then replaces the expression in the original line with the temporary variable. This ensures that the expression only executes once and that the trace records its exact behavior during the execution of the program.

Once Anteater completes the instrumentation, it compiles the AST into an executable program, which generates the trace as it executes.

5.2 Data Organization

After Anteater instruments the source code, it runs the modified program and creates the trace file. Anteater writes the raw trace as a simple JSON file. This allows it to easily capture the hierarchical structure of the execution as well as record data about program blocks as attributes in the corresponding JSON block. Anteater then passes the trace to the front-end. While convenient for collecting the trace, JSON is less convenient and flexible for querying the trace which limits the range of possible visualizations and interactions. Extracting instances of variables from the JSON structure requires traversing the entire structure. This makes even the most basic queries, simply selecting data from the structure, inefficient. Furthermore, more complex interactions such

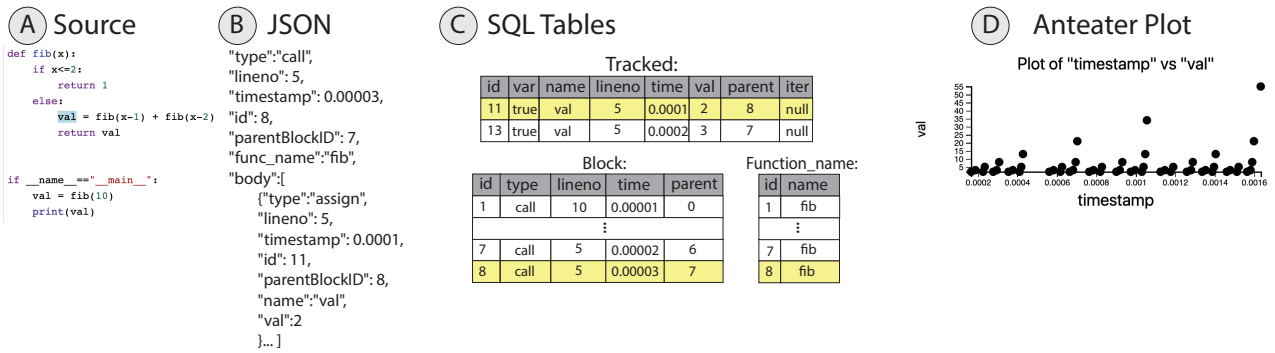

Fig. 4. An overview of how Anteater goes from source code to visualization. (A) shows the initial source code. We are going to track the variable "val" After instrumenting the source code, as demonstrated in Fig. 3. The instrumented program creates a trace cell as shown in (B). Anteater then puts the JSON into a SQL table as shown in (C). From there, Anteater queries the table to select all points from "Tracked" that have the name "val" and creates a scatterplot of those points over time.

as filtering a variable or joining two variables, result in complicated traversals of the JSON structure that further decrease the efficiency of queries. To support more complex visualizations and interactions, Anteater converts the JSON trace into a SQL database.

Anteater converts the JSON trace into several SQL tables, the primary two being "block" and "tracked". The "block" table stores information about all execution blocks that do not correspond to tracked values. It includes columns storing the id, type, line number, timestamp, and parent id of the block. Fig 4 demonstrates how to convert a JSON call block into its corresponding SQL tables. Anteater uses these blocks to create the generalized context tree, as shown in Fig. 2. Using the parent id of each block, we could build the hierarchical structure used to create the generalized context tree by starting at the root, querying the SQL database to find all of the blocks that have the root as their parent, and then visiting each child node and repeating this process.

The other primary table, the "tracked" table, stores the occurrences of the variables and expressions the user tracked. This table stores the id, name, line number, timestamp, value, parent id, iteration number, and a boolean indicating if it is a variable (as opposed to an expression). Anteater queries this table to build the visualizations. For basic, unfiltered visualizations, Anteater simply queries this table for all records with a certain name. For filtered queries, it either specifies a range or a specific set of id's that the record can take on using the `WHERE` clause. Fig 4 describes how to go from the source code, to JSON, to SQL, to visualizations.

Additional tables exist, such as "function_name" and "for_loop" that store additional information about certain types of blocks. The "custom" table stores the values of custom expressions that are collected alongside the variables and expressions selected in the source code.

Converting the trace to SQL yields several advantages. First, querying becomes much simpler. For basic visualizations, we now must simply write a `SELECT` statement to gather all instances of a tracked variable. To filter instances, we can simply add `FILTER ON` to the SQL statement. Similarly, joining two variables becomes much simpler through the use of `JOIN`. Fig. 5 shows a table of visualizations supported by Anteater and the corresponding SQL query keywords used to collect the data.

Second, Anteater supports any visualization for which there exists a SQL query to select the appropriate data. In other words, forming the proper query becomes the only restriction to the range of possible visualizations. While the current implementation only supports a few visualizations, we could easily extend it to support others.

The last advantage comes from the decoupling of the visualizations and the data representation. The specification of the visualizations does not inherently depend on the representation of the data. A SQL query simply returns a list of datapoints for Anteater to use in the visualization. Because of this, new visualization implementations can be plugged in with minimal effort to adapt them to fit into Anteater. This further increases the extensibility and flexibility of Anteater.

Data Type	Plot Type	Query
Q	Histogram	`SELECT`
N	Bar plot	`SELECT`
QxQ	Scatter	`SELECT, JOIN`
QxQxQ...	Parallel Coordinates	SELECT, JOIN
N, Q, QxQ	Small Multiples	`SELECT, JOIN, SORT ON`

Fig. 5. The above table shows the current visualizations supported and the SQL queries used to create these visualizations. We use "Q" for quantitative data and "N" for nominal.

6 Anteater's Visualization Design

Anteater presents a new way of exploring and interacting with program executions helping users to gain a deeper understanding of the inner-workings of their programs that they cannot get from traditional tools. In the previous section, we discussed how Anteater creates the execution trace. Here, we describe the visualization design of Anteater and the features that facilitate the exploration of the execution trace. As we walk through the design, we will describe the features in context of a simple Python program that runs a recursive Fibonacci function. In addition, we use Yi et al.'s categories of interactions [41] to classify our interactions and further validate our design.

6.1 Creating a Program Trace

To fulfill T1, Anteater needs to allow a user to define which variables and expressions to trace. The first page a user sees after they load their source code allows them to set the specifications of the trace by highlighting text in their source code and right clicking to specify the action (track or exclude). Users can also define additional expressions on their chosen variables and expressions to track during execution. Anteater best supports numerical values but has limited support for strings and booleans. While lists and matrices cannot directly be visualized, information about either structure can be tracked using custom expressions. Once the users complete the trace specification, Anteater passes it to the tracer in the backend for processing. Allowing users to select what data they desire to see falls into Yi et al.'s "select" interaction category.

The tracer will only collect variables and expressions that the user specifies. This was an explicit design choice, because the entirety of data associated with every single variable is massive. Collecting all variables would also record unnecessary data. Many variables residing in a program have little importance in describing how code is behaving. Thus, we allow the user to select the important variables to track.

Similarly, collecting all function calls leads to a large collection of unimportant information. To help reduce clutter from unimportant function calls, we add a predefined list of functions and libraries to ignore (e.g., math, numpy, print, len) and allow users to add functions of their own to exclude from the trace. This allows users to reduce clutter and remove uninteresting/unimportant structures from the trace to better highlight the important structures.

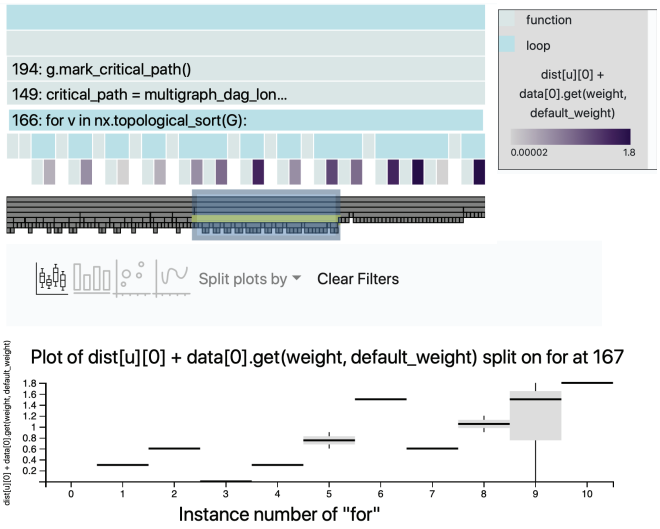

Fig. 6. An example of Anteater splitting the data by a structural element. Anteater splits the data by instances of a for loop at line 167, which corresponds to iterations of the loop at line 166 (the selected block in the generalized context tree). The plot shows one boxplot per loop instance.

6.2 Visualizing Trace Data

Once the tracer returns the execution trace, Anteater generates interactive visualizations. Two types of visualizations are provided: a view of the execution structure, which we call the generalized context tree after Boehme et al. [8], and a visualization of the variable values.

6.2.1 Generalized Context Tree

The generalized context tree, shown on the right side of Fig. 2-A and in Fig. 2-B, provides an overview of the execution structure. The visualization has its origins in flame graphs and icicle plots. We chose this type of visualization because it is well known for visualizing traces and well understood. In our setting, each rectangular block in the plot represents one of three things: a function call, a loop, or a variable assignment. The icicle plot shows the hierarchy so that for a given block, everything that is within its bounds below it, is a child. For example, in Fig. 2-A, the block in the second row labeled "10: val = ..." is the initial call into the Fibonacci function and everything below that happens within that call. The generalized context tree can be used to determine which functions executed and when, fulfilling T2.

As we move from left to right in the plot, we are increasing in time; everything to the left of a block was executed before that block. This allows users to easily read the visualization and understand when blocks are executed relative to other blocks.

In the generalized context tree, a single variable is highlighted, selected by the user (in the upper right corner of Fig. 2-A). When users select a variable, all blocks in the tree corresponding to the variable (which reside at the leaf level) are colored by the value of that instance. Positive values range from white (low) to purple (high), while negative values range from white (least negative) to orange (most negative). In Fig. 2-A, Anteater colors the leaf nodes representing the variable "val" with varying shades of purple. Deeper leaves are shaded much lighter, which indicates small values at those instances; this corresponds to the deepest Fibonacci calls returning the smallest values. Coloring blocks in this way shows the behavior of values in the context of the whole execution. Every other variable or expression that appears in the trace still appears in the generalized context tree; Anteater colors them gray.

6.2.2 Variable Value Plots

The second visualization provided by Anteater, is a plot of tracked variables. Users drag variables and expressions from the source code into the plot to visualize them. Anteater then automatically determines the visualization options available for that datatype (Fig. 5 shows the supported plots). Users can click on the icons above the plot to switch between the different plot types available for that datatype. This falls into Yi et al.'s "encode" category.

To support T5, Anteater allows the user to drag multiple variables into the plot. If the variables are compatible, Anteater plots them against each other in either a scatterplot or parallel coordinates (depending on the number of variables), allowing the user to observe their relationship. Compatible variables share a common ancestor and have 1-1 instances within that ancestor. Allowing users to change the variable on each axis falls into Yi et al.'s "reconfigure" category.

Anteater also provides grouping capabilities that allow the user to split the data into groups and create either a box and whisker plot or small multiples of plots. The data can be split on either a variable/-expression from the trace that takes on sufficiently few values (i.e., will not create dozens of plots) or a repeated structure in the execution, such as a loop, where each instance of the structure contains multiple instances of the tracked variables/expressions. For example, in Fig. 6, Anteater splits the plot on the outer loop and creates a box and whisker plot for each instance of the inner loop. Splitting points into groups falls into Yi et al.'s "explore" category.

6.3 Interacting with the Trace Visualizations

Anteater's interactions are key in helping users get a better understanding of their program. Anteater offers several interactions that afford the user the capabilities to 1) maintain context with the source code, 2) inspect the dependencies of a certain instance of a variable or expression, 3) link between the value plots and the generalized context tree, and 4) filter the visualizations to narrow the focus to an interesting/important subset. The last capability enables users to better understand the relationship between specific values and the structure of the execution.

6.3.1 Maintaining Source Code Context

When exploring the execution, it is important to link back to the source code to maintain the context of the execution. On its own, the generalized context tree is fairly abstract. To provide necessary context, when the user selects a block in the generalized context tree, the source code jumps to, and highlights, the corresponding section of the code. If it corresponds to a user-defined function call, it also highlights the corresponding function. This interaction, paired with a preview of the corresponding source code on the blocks, supports T6 by allowing users to navigate the execution trace without forgetting their place in the source code. It also falls into Yi et al.'s "connect" category.

6.3.2 Inspecting Dependencies

To support T3, Anteater determines what dependencies could exist for any instance of a variable (as discussed earlier) and then uses context from the execution trace to eliminate some possibilities and present the remainder to the user. When a user selects a block representing a variable in the generalized context tree, Anteater checks if the prior siblings as well as the siblings of any ancestors of that block are in the list of possible dependencies. If they are, their blocks are highlighted in red to show the user on which parts of the context tree that block depends. This allows the user to quickly get an idea of which entities contribute to that specific instance. In Fig. 2-A, the selected instance of "val" depends on the prior two calls to "fib". This interaction falls into Yi et al.'s "explore" category.

6.3.3 Linking visualizations

Anteater provides interactions on the plots and the generalized context tree to link the two together. When a user selects a block in the generalized context tree, the values shown in the plot filter down to include all values in the subtree rooted at the selected block. In addition, to provide global context, the plot shows the values from the subtree rooted at the parent of the selected block. As shown in the histogram in Fig. 9-B, Anteater colors the bar representing the selected instance(s) black while the coloring rest of the bars gray for context. In the scatterplot, it colors the points representing selected instances black while coloring the rest gray. We also provide linking from the plot back to the generalized context tree. In the histogram, selecting a bar highlights the corresponding blocks in the tree, as shown in Fig. 9-A. In the scatterplot, brushing over a set of points highlights the corresponding blocks in the trees, as shown in Fig. 2-B where the red blocks in the tree correspond to the

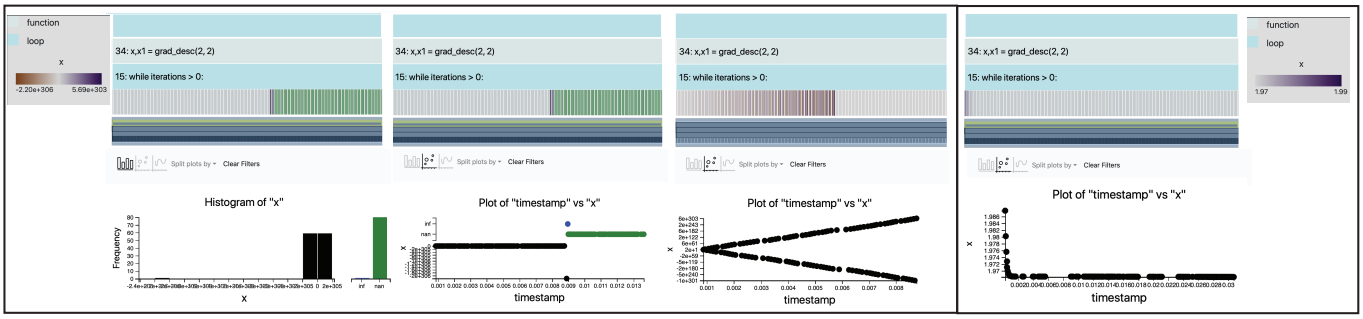

Fig. 7. Debugging Gradient Descent with Anteater. In (A) it is immediately apparent in both the generalized context tree and the histogram that there is a bug causing NaN's. In (B), we switch to the scatterplot view to see how the values behave before they become NaN. The values are mostly centered around zero before becoming an extremely small negative, then going to infinity and becoming NaN. We suspect that the values centered around zero are not actually zeros so we filter the values in the scatterplot to allow us to zoom in on them and switch to a symmetric log scale, shown in (C). Now we see that the values are oscillating which suggests the problem of exploding gradients caused by a training rate that is too large. In (D) we change the gradient and see that the value quickly converges as it should.

brushed points. These interactions support T4 by allowing the user to pinpoint interesting values in the plots and locate them in the execution. They also fall into Yi et al.'s "select" and "connect" category.

6.3.4 Filtering

Anteater supports a few types of filtering on the plot and the generalized context tree to help users filter out unimportant information and emphasize important parts of the execution, which helps support T4. The first type of filtering was mentioned above where clicking on deeper nodes in the context tree filters the value plots. Through this interaction, users can filter down the plot to interesting subsets of the data. This falls into Yi et al.'s "explore" and "filter" category.

In the scatterplot, users can brush over a subset of points, right click, and select to filter out the values not in their brush. Anteater then removes all other points from the plot, effectively zooming in on selected points, and colors the blocks gray in the generalized context tree corresponding to the filtered out points. An example of this can be seen in Fig. 7(C) and Fig. 9-C. This falls into Yi et al.'s "filter" category.

One last way users can filter the visualization is by hiding parts of the generalized context tree. Right clicking on a block in the tree will expand the block to take up the entire width of the interface, increasing the size of all of its children and thus making them easier to see. However, in doing this, users might lose context of where they are exploring with respect to the execution. To retain this context, we add a smaller, grayscale version of the generalized context tree with a highlighter bar over it. When the user zooms in on a block, the highlighter narrows to indicate its place in the overall context tree. It also highlights the selected block in yellow, as well as any other blocks that are highlighted in the generalized context tree (dependencies and brushed values). This allows users to see them even if they are outside of the visible portion of it. In Fig. 6, we zoomed in on the loop at 166, and we see our location in the execution in the context bar. This interaction falls into Yi et al.'s "abstract/elaborate" category.

7 Evaluation

We evaluated the efficacy of Anteater's framework through a user study (Sec. 7.1) and a series of case studies (Sec. 7.2).

7.1 Pair Analytics User Study

User affordances offered by and the development status of a visualization prototype are key factors to steer the design of a user evaluation study [15]. In the case of Anteater, we do not intend to validate the scalability or usability of its interface and architecture (see Section 8). Rather, it is more appropriate to validate Anteater's visualization design principles and the user exploration processes that Anteater facilitates. Hence, we chose *pair analytics* [4] an appropriate user evaluation protocol.

Pair analytics offers a "think-aloud" protocol that helps generate verbal data by capturing the natural interaction between study participants and the proctor using the visualization interface as a communication anchor. Using the pair analytics method, a team is formed between a study proctor (or a visualization expert) and a subject matter expert. This approach allows the subject matter expert to focus less on the nuances of the visualization interface (e.g., interaction types, loading data, etc) and more on exploration and question-answering processes.

7.1.1 Methodology

For each study, we recorded screen capture data along with audio recordings of each interview. Due to the current policies in place in the U.S., all sessions were held online rather than in person, as would typically be done. Anteater's primary developer served as the study proctor and students from a Principles of Machine Learning course served as the subject matter experts aiming to gain a deeper understanding of their Python programs. The study proctor guided the interactions within Anteater based on the verbal commands from the subject matter expects. Akin to other pair analytics evaluation studies, the proctor freely asked questions to promote exploratory thinking. In effect, participants' answering of such questions helped distill internal cognitive process that were qualitatively analyzed.

We recruited total of 5 participants from the class, which had a total of 20 students. However, only 3 of the studies were carried out to completion. One of the studies was discarded since the participant provided a program with a known bug that they thought might be interesting to re-discover with Anteater. While the subject matter expert's program was appropriate for the user study, we thought the a prior knowledge of the participant would bias the study's outcome. As a result, we promoted this program to a case study and discuss it in Sec. 7.2. We discarded another study because the programs presented by the participant were not a good fit for the study. They do, however, highlight some of the limitations of Anteater and will be discussed in more detail later in 7.1.2. For the participants who completed the study, sessions lasted between 60 and 90 minutes. Approximately the first 30 minutes of each session was spent introducing the subject matter experts to Anteater and getting Anteater set up to run properly on their machines. All of the participants use Python as their primary language in their work and consider themselves to be experts in Python.

7.1.2 Results

From this study, we found that, even in its imperfect prototype state, Anteater was useful to participants for debugging and achieving a better understanding of their programs. For the sake of confidentiality, we cannot give specifics about the programs used by participants. However, we try to give some context in the form of general concepts found in data analysis programs. Participants provided their own Python programs for inspection with Anteater. All of the programs performed some form of data analysis.

The first participant (P1) knew a bug existed in their program causing it to run incorrectly, but had yet to find it. With the proctor's guidance,

P1 leveraged Anteater to identify and fix the bug. Through the use of the timeline plot and the ability to track custom expressions on more complex data structures, P1 found the bug, fixed it, and then verified that the revised program ran properly. During the exploration process, P1 discovered that there was something unusual about the training dataset, denoted as the *whole* dataset, which is split into *left* and *right*, vital to the proper execution of the program. P1 correctly noticed the problem since "the right dataset and the [whole] dataset cannot be the same" even though the scatter plot showed them as identical. Upon further investigation of the captured values in each dataset, P1 explained that the "right dataset ... points to [the] class dataset" which causes them to overwrite the whole training set with only the *right* one. After modifying the code, a new trace was run and P1 validated the proper behavior of the code. After being asked if they were "able to gain new insight into [their] program using Anteater," P1 answered that "the scatter graph and also the tracking values [were] very helpful."

The other two participants (P2 and P3) presented more open-ended cases. P2 and P3 did not have known bugs, rather non-trivial data analysis programs whose execution was not fully understood. In both cases, the timeline view of certain variables over time was crucial. P2 heavily relied on the timeline and filtering capabilities of Anteater to verify that their program was converging as expected. P2 also used the timeline and filtering feature to inspect if their program was reaching the extremes of their search space. Using Anteater, P2 discovered that the program did not search the entire space in one direction and searched beyond the bounds in the other direction. After completing their exploration, P2 commented that understanding "why the values are so far off from the [search space] is a good next thing to look at."

P3 also heavily relied on the timeline view. They used it to understand the behavior of a set of weights in their analysis program. Before their use of Anteater, P3 had little idea of how the weights behaved throughout their program's execution. Anteater allowed them to track and visualize the weights over time to see how they evolved as the program ran. After they inspected the weights, the participant commented that "[Anteater] completely helped [them] understand sort of the underlying domain thing of what was going on with the weights." P3 further explained that Anteater was able to show that the program "is converging on one particular feature as an important weight and the rest [are seen as] super unimportant." Through the use of Anteater, P3 was able to understand the behavior of the weights relative to the domain for which the program addressed, specifically through the use of the visualizations.

After participants finished their exploration with Anteater, they were asked a set of questions about their experience with it. The participants were all able to gain new insight into their programs: P1 by finding their bug and P2 and P3 through understanding the behavior of certain values. When asked what they did not like or find useful, most of their responses were related to usability problems with the current implementation of Anteater, such as confusion about when to right click or left click and when they can alter the source code to have it propagate to the backend. We consider all of their concerns to be minor implementation problems, rather than problems with the underlying system design. P2 found that for their problem, the histograms were not very useful but could see how such plots could help others. All participants expressed that, if a polished and optimized version were available, they would like to use a system like Anteater for future problems.

As mentioned earlier, we discarded one evaluation study, because the programs provided were not ideally suited for the objective of the evaluation. The participant initially brought a large, machine learning program that took approximately a week to run. This program was not a good fit since we do not aim to study the interaction between trace size and applicability of our approach, but rather the utility of our approach to real Python programmers. A program that takes a week to run will generate a trace too large to handle by the current implementation of Anteater. Admittedly, this does limit the generalizability of Anteater, but we consider the study of how to scale a system like Anteater for future work. The participant then provided several small-scale programs that we were also not ideally fit for the study. This study aimed to use programs similar to those that would be found in a real data analysis setting. The small-scale programs were simply not sufficiently realistic for the study. They were more in-line with programs found in an educational setting. As a result, rather than asking them to provide additional programs, we omitted their case from the study.

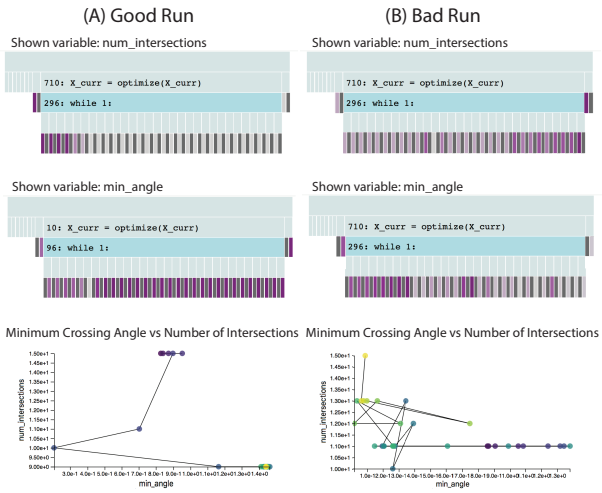

Fig. 8. Using Anteater to compare two runs of gradient descent that should maximize the minimum crossing angle while minimizing edge crossings. The generalized context trees in (A) show that the number of intersections rapidly decreases (the color changes from dark purple to white) while the minimum angle increases. The scatterplot shows that the descent spends its first few steps at a bad solution and takes approximately three big steps before converging on a good solution. In contrast, in (B) the number of intersections increases throughout the descent while the minimum angle decreases. The scatterplot shows that, in general, as the number of intersections grows, the minimum angle shrinks and lands at a bad solution.

7.2 Case Studies

Here, we present several, real-world cases, showcasing how Anteater derives insight into debugging and program understanding.

7.2.1 Gradient Descent

The first case study we present inspects a program performing gradient descent. This program was collected from a question on Stack Overflow [1]. The programmer struggled to figure out why the resulting values were NaNs. We will walk through how to use Anteater to understand the bug and correct it.

First, we run the program with Anteater to track the misbehaving variable, "x." Fig. 7-A shows the resulting execution tree and histogram. The histogram shows that much of the descent generates NaNs.

As a natural next step, we look at these values over time. We switch the plot type to "scatterplot" which shows a plot of the variable "x" over time, shown in Fig. 7-B. Now, we clearly see that the value of "x" stays around zero, before becoming a very small negative, then going to infinity after which we hit the NaNs. However, there is something strange in the values staying around zero and then suddenly becoming a very small negative. To investigate this, we filter the values into to show only those points staying close to zero. We also switch to a symmetric log scale because we suspect that the values may not actually lie that close to zero. The resulting visualizations are shown in Fig. 7-C. We see that the value oscillates between increasingly large positives and negatives until it reaches infinity.

Now that we know the problem, we try to fix it. The oscillating values suggest that the gradients is exploding due to a training rate that is too large. In Fig. 7-D, after lowering the training rate and re-running the trace, the value quickly converges, as expected.

Using Anteater, we quickly and easily track the variable "x" and see its behavior throughout the execution. In a traditional debugger, detecting this behavior would have required stepping through several iterations to view the values. After lowering the training rate, we repeat this process to determine if that fixed the problem. This involves significantly more interaction with the debugger than when using Anteater.

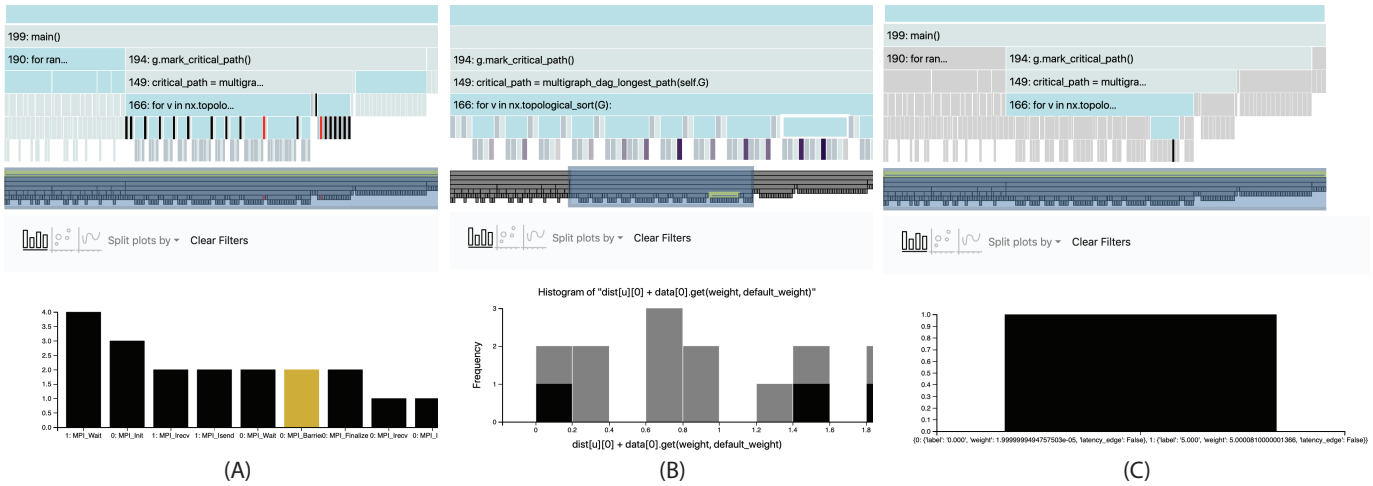

Fig. 9. (A) shows how we can use the bar plot to find where the barrier node occurs in the execution plot. (B) shows the weights of the paths to the barrier node calculated in the program. We see that none of them are 5, as we expect from looking at the MPI call graph. (C) shows the data used to calculate the path weight that should include the edge of weight 5. We see that the data does include this edge but the algorithm does not look at it. For more details, refer to the supplemental video.

7.2.2 Graph Edge Crossing Angle Maximization

In this case study, we investigate a program that tries to balance the number of edge crossings in a graph with the size of the minimum crossing angle. The program searches for the layout that minimizes the number of edge crossings while maximizing the size of the minimum crossing angle. In this case study, we inspect the stability of the gradient descent method on this problem.

To inspect the stability, we ran the gradient descent multiple times, tracking the minimum angle and number of intersections at each iteration of the gradient descent. We found that in most cases, the gradient descent returns a good solution, as demonstrated in Fig. 8-A, where it immediately begins moving toward a good solution and never turns back. However, instances occur, as shown in Fig. 8-B, where the gradient descent starts moving towards a bad solution, and never recovers. Therefore, we can conclude that although the majority of the time it produces a good solution, this method suffers from stability issues.

7.2.3 Longest Weighted Path Calculation

This case study was presented to us by a prospective participant in the user study. While the program was not a good fit for the study, because the participant already knew where the bug was, it presents a good example of the utility of Anteater on real problems. This program aims to find the critical path, i.e., the longest weighted path, from the ”Init” to ”Finalize” nodes in an MPI call graph. It uses the networkx library to build a multiDAG and calculate the longest (weighted) path. We were given this program with the knowledge that this bug existed and which methods were affected but no other information on how to fix it. We then found and fixed the bug using only Anteater. Below, we explain how we found the bug.

To begin, we loaded the program and data files into Anteater. We know from inspecting the test graph manually that the algorithm overlooks one of the edges (of weight 5) from the ”Init” node into the ”Barrier” node. To build the longest path, the algorithm topologically sorts the nodes and iterates over them. For each node, it iterates over all of the predecessor nodes. To find the bug, we first want to find where the barrier node occurs. We do this by collecting the node label in each iteration. By inspecting the node labels in the bar plot, as shown in Fig 9-A, we find the point in the execution tree where the loop reaches the ”barrier” node. Once we find the barrier node, we select it to view the other values in that specific iteration. We then switch variables to look at the path weights for each predecessor, as shown in Fig. 9-B. We see that none of the path weights reach 5, which indicates that the algorithm misses the edge of weight 5 into the “Barrier” node. Next, we look at the data used to calculate the path weights. We notice that one of the “Init” predecessors has two weights associated with it, as shown in the filtered bar in Fig. 9-C. There are two keys in the dictionary, one for each edge from “Init” to “Barrier”. Looking back at the algorithm, we see that it only looks at the first key which causes it to miss the edge of weight 5 and report an incorrect longest weighted path. To fix this, we simply find the edge with the highest weight over all of the edges from the predecessor to current node.

8 Discussion and Limitations

Omnicode vs. Anteater While Omnicode and Anteater both intend to help programmers debug and understand their programs, the two systems differ in their target audience. Omnicode aims to help novice users create mental-models to reason about their program’s execution and debug unexpected behavior. The size and complexity of programs it needs to support for this audience is quite small. Thus, Omnicode only supports programs of around 10 variables and 100 execution steps. Anteater aims to help programmers in general. Therefore it needs to support different types of programs.

While Anteater cannot support large software-systems as they produce an unmanageable amount of data, it can support much larger programs than those written by novices, such as those programs written by data scientists. Most of the differences between Omnicode and Anteater stem from the fact that they are geared toward different audiences. Omnicode supports a live programming environment because its target programs are small whereas a static environment makes more sense for Anteater. Similarly, Omnicode tracks every variable in the program which is infeasible for the larger programs Anteater supports.

Limitations Anteater will not scale to large traces. In these programs, the traces become too large and the visualizations unreadable. Research exists on collecting the entire trace of large programs [29]; future work is needed to evaluate if Anteater works well with this method. We note that our visualizations operate on relational data, and there is a growing number of techniques to support interactive visualizations on large datasets [17]. A full investigation of their impact on program visualization, however, is out of present scope. In addition, Anteater works best with numerical data and has limited support for other datatypes. While it can present numbers, strings, and booleans, it does not support compound objects directly. Information about variables of these datatypes can still be visualized through the use of custom expressions, but we leave first-class support for more datatypes for future work. Finally, Anteater assumes a sequential programming model and does not support parallel programs. Work exists in automatic tracing of parallel programs in the traditional sense (without values) but applying and extending these traces to our system is left for future work.

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.

Acknowledgments This work is partially supported by the NIST Graduate Student Measurement and Engineering Fellowship, through a grant with the NPSC (Award #70NANB16H141), and NSF awards IIS-1815238. We thank Tim Zimmerman, Chee Yee Tang, and Rick Candell from NIST for their helpful introduction to the Tennessee Eastman challenge problem.

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.

[2] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer. Heapviz: Interactive heap visualization for program understanding and debugging. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS, pp. 53–62. ACM, New York, NY, USA, 2010. doi: 10.1145/1879211.1879222

[3] 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.

[4] R. Arias-Hernandez, L. T. Kaastra, T. M. Green, and B. Fisher. Pair analytics: Capturing reasoning processes in collaborative visual analytics. In *2011 44th Hawaii international conference on system sciences*, pp. 1–10. IEEE, 2011.

[5] F. Beck, F. Hollerich, S. Diehl, and D. Weiskopf. Visual monitoring of numeric variables embedded in source code. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 1–4. IEEE, 2013.

[6] J. Bertin, W. J. Berg, and H. Wainer. *Semiology of graphics: diagrams, networks, maps*, vol. 1. University of Wisconsin press Madison, 1983.

[7] C.-P. Bezemer, J. Pouwelse, and B. Gregg. Understanding software performance regressions using differential flame graphs. In *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering*, SANER, pp. 535–539, mar, 2015. doi: 10.1109/SANER.2015.7081872

[8] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz. Caliper: Performance introspection for hpc software stacks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pp. 47:1–47:11. IEEE Press, Piscataway, NJ, USA, 2016.

[9] B. Borkholder. Mirur, 2014. `https://mirur.io`, last visited on 2020-04-30.

[10] M. S. T. Carpendale. Considering visual variables as a basis for information visualisation. 2003.

[11] 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

[12] A. Coleman. How to quickly and effectively read other peoples code. "https://selftaughtcoders.com/how-to-quickly-and-effectively-read-other-peoples-code/".

[13] S. Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.

[14] L. N. Q. Do, S. Krüger, P. Hill, K. Ali, and E. Bodden. Visuflow: A debugging environment for static analyses. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pp. 89–92. ACM, New York, NY, USA, 2018. doi: 10.1145/3183440.3183470

[15] N. Elmqvist and J. S. Yi. Patterns for visualization evaluation. *Information Visualization*, 14(3):250–269, 2015.

[16] P. Gestwicki and B. Jayaraman. Methodology and architecture of jive. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, pp. 95–104. ACM, New York, NY, USA, 2005. doi: 10.1145/1056018.1056032

[17] P. Godfrey, J. Gryz, and P. Lasek. Interactive visualization of large data sets. *IEEE Transactions on Knowledge and Data Engineering*, 28(8):2142–2157, 2016.

[18] P. Gralka, C. Schulz, G. Reina, D. Weiskopf, and T. Ertl. Visual exploration of memory traces and call stacks. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 54–63. IEEE, 2017.

[19] J. Hoffswell, A. Satyanarayan, and J. Heer. Augmenting code with in situ visualizations to aid program understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, p. 532. ACM, 2018.

[20] S. Jarman. Tips for navigating large and unfamiliar codebases. "https://hackernoon.com/tips-for-navigating-large-and-unfamiliar-codebases-f5426cea552c", 2017.

[21] JetBrains. Pycharm, the python ide for professional developers, 2000. `https://www.jetbrains.com/pycharm/`, last visited on 2020-04-30.

[22] H. Kang and P. J. Guo. Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pp. 737–745. ACM, 2017.

[23] B. Karran, J. Trümper, and J. Döllner. Synctrace: Visual thread-interplay analysis. In *Proceedings of the 1st Working Conference on Software Visualization*, VISSOFT, p. 10. IEEE Computer Society, 2013. doi: 10.1109/VISSOFT.2013.6650534

[24] H. Lam, M. Tory, and T. Munzner. Bridging from goals to tasks with design study analysis reports. *IEEE transactions on visualization and computer graphics*, 24(1):435–445, 2018.

[25] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 185–194. ACM, 2010.

[26] H. Lieberman and C. Fry. Bridging the gulf between code and behavior in programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pp. 480–486. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995. doi: 10.1145/223904.223969

[27] S. Litvinov, M. Mingazov, V. Myachikov, V. Ivanov, Y. Palamarchuk, P. Sozonov, and G. Succi. A tool for visualizing the execution of programs and stack traces especially suited for novice programmers. *arXiv preprint arXiv:1711.11377*, 2017.

[28] T. Ohmann, R. Stanley, I. Beschastnikh, and Y. Brun. Visually reasoning about system and resource behavior. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pp. 601–604. ACM, New York, NY, USA, 2016. doi: 10.1145/2889160.2889166

[29] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pp. 535–552. ACM, New York, NY, USA, 2007. doi: 10.1145/1297027.1297067

[30] S. P. Reiss. The challenge of helping the programmer during debugging. In *2014 Second IEEE Working Conference on Software Visualization*, pp. 112–116, Sep. 2014. doi: 10.1109/VISSOFT.2014.27

[31] M. Renieris and S. P. Reiss. ALMOST: Exploring program traces. In *1999 Workshop on New Paradigms in Information Visualization and Manipulation*, pp. 70–77. ACM, 1999. doi: 10.1145/331770.331788

[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.

[33] R. Schulz, F. Beck, J. W. C. Felipez, and A. Bergel. Visually exploring object mutation. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 21–25, Oct 2016. doi: 10.1109/VISSOFT.2016.21

[34] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *The Craft of Information Visualization*, pp. 364–371. Elsevier, 2003.

[35] D. Socha, M. L. Bailey, and D. Notkin. Voyeur: Graphical views of parallel programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, PADD, pp. 206–215. ACM, New York, NY, USA, 1988. doi: 10.1145/68210.69235

[36] Sourcegraph. Sourcegraph: Search, navigate, and review code, 2018. `https://about.sourcegraph.com/`, last visited on 2020-04-30.

[37] J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price. *Software Visualization*. MIT Press, 1998.

[38] J. Sundararaman and G. Back. Hdpv: Interactive, faithful, in-vivo runtime state visualization for c/c++ and java. In *Proceedings of the 4th ACM Symposium on Software Visualization*, SoftVis '08, pp. 47–56. ACM, New York, NY, USA, 2008. doi: 10.1145/1409720.1409729

[39] R. Tiarks and T. Roehm. Challenges in program comprehension. *Softwaretechnik-Trends*, 32(2):19–20, 2012.

[40] J. Trümper, J. Bohnet, and J. Döllner. Understanding complex multithreaded software systems by using trace visualization. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS, pp. 133–142. ACM, New York, NY, USA, 2010. doi: 10.1145/1879211.1879232

[41] J. S. Yi, Y. ah Kang, J. T. Stasko, and J. A. Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1224–1231, 2007.