Graft: A Debugging Tool For Apache Giraph

Semih Salihoglu, Jaeho Shin, Vikesh Khanna, Ba Quan Truong, Jennifer Widom
Stanford University
{semih, jaeho.shin, vikesh, bqtruong, widom}@cs.stanford.edu

ABSTRACT
We address the problem of debugging programs written for Pregel-like systems. After interviewing Giraph and GPS users, we developed Graft. Graft supports the debugging cycle that users typically go through: (1) Users describe programmatically which vertices they are interested in inspecting. During execution, Graft captures the context information of these vertices across supersteps. (2) Using Graft’s GUI, users visualize how the values and messages of the captured vertices change from superstep to superstep, narrowing in suspicious vertices and supersteps. (3) Users replay the exact lines of the vertex.compute() function that executed for the suspicious vertices and supersteps, by copying code that Graft generates into their development environments’ line-by-line debuggers. Graft also has features to construct end-to-end tests for Giraph programs. Graft is open-source and fully integrated into Apache Giraph’s main code base.

1. INTRODUCTION
The Pregel distributed graph-processing engine [18] and its open source versions, such as Apache Giraph [7], Apache Hama [11], and GPS [24], are being adopted by a growing number of applications for processing large-scale graphs. For example, Facebook is using Apache Giraph in production for its Graph Search application [22] and its recommendation algorithms, and PayPal is using Giraph for fraud detection and user credit risk [27]. Like MapReduce [3] and Hadoop [10] for record-oriented data, Pregel-like systems offer transparent scalability, automatic fault-tolerance, and a simple programming interface based around implementing a small set of functions.

The computational framework introduced by Pregel is based on the Bulk Synchronous Parallel (BSP) computation model [28]. At the beginning of the computation, the vertices of the graph are distributed across Worker tasks running on different compute nodes. Computation is broken down into iterations called supersteps, and all workers synchronize at the end of each superstep. Algorithms are implemented in a vertex-centric fashion inside a vertex.compute() function, which gets called on each vertex exactly once in every superstep. Inside vertex.compute(), vertices receive messages from the previous superstep, update their local values, and send messages to other vertices. In Giraph [7] and GPS [24], an optional master.compute() function is executed by the Master task between supersteps.

We have tackled the challenge of debugging programs written for Pregel-like systems. Despite being a core component of programmers’ development cycles, very little work has been done on debugging in these systems. We interviewed several Giraph and GPS programmers (hereafter referred to as “users”) and studied how they currently debug their vertex.compute() functions. We found that the following three steps were common across users: (1) Users add print statements to their code to capture information about a select set of potentially “buggy” vertices, e.g., vertices that are assigned incorrect values, send incorrect messages, or throw exceptions. The captured set of vertices is typically quite small, sometimes containing as little as a single vertex with its neighbors, because it is slow to log, and difficult to inspect the information of a large number of vertices. (2) Then, users inspect the captured vertex information and mentally “replay” their graph algorithms superstep by superstep, until they narrow in on the most suspicious vertices and supersteps. (3) Finally, they return to their code and try to identify the part of vertex.compute() that must have executed on the suspicious vertices and supersteps, hoping to find the bug.

Based on our observations, we designed and developed Graft, a new replay-style debugger that is tailored specifically for the needs of Giraph users. Existing replay debuggers, e.g. [2, 6], capture and replay all low-level system calls made by a distributed application, such as memory reads and writes to the network drivers, which are usually not relevant for diagnosing bugs inside vertex.compute() functions. They also do not provide any replay functionality specific to Pregel’s vertex-centric graph computaions. Graft’s approach is motivated by the three manual steps we observed in users’ current debugging cycles, which we call capture, visualize, and reproduce, respectively:

- **Capture**: Users describe programmatically which vertices they are interested in capturing (details in Section 3.1). Graft captures the entire context information for these vertices, across all supersteps or a user-defined selection of supersteps. It is expected that the selected set of vertices will be relatively small, and the rich API encourages applying selective criteria.

- **Visualize**: Graft includes a graph-specific and superstep-based visual interface for users to replay the algorithm’s effects on the vertices whose contexts have been captured. Users can see how the values and messages of these vertices change from superstep to superstep, narrowing in on suspicious values, messages, or exceptions.

- **Reproduce**: The last step involves code inspection, for which we rely on the user’s integrated development environment (IDE), such as Eclipse [4] or IntelliJ [12]. The context that Graft cap-
2. BACKGROUND: GIRAPH API

The Giraph API consists of the four classes that were described in the original API of Pregel, and an optional Master class, which was introduced by GPS [24]. For the purposes of describing Graft, the important components are:

- **vertex.compute()**: Users subclass the vertex class and code the vertex-centric logic of the computation by implementing the vertex.compute() function. Inside vertex.compute(), a vertex has access to five pieces of data: (1) the vertex ID; (2) its outgoing edges; (3) its incoming messages; (4) a set of aggregators (see below); and (5) default global data consisting of the current superstep number and the total number of vertices and edges in the graph. Each vertex also has an active/inactive flag; a vertex declares itself inactive by calling the voteToHalt() function in the API. The system terminates computation when all vertices become inactive or the Master class instructs the system to terminate. We note that in Giraph, the vertex.compute() function is not part of the Vertex class but a class called Computation. For simplicity of presentation, we will ignore this difference in class names.

- **Aggregators**: Global objects visible to all vertices; used for coordination, data sharing, and statistics aggregation. When multiple vertices update their local copy of an aggregator during a superstep, the system merges the updates at the end of the superstep using a user-specific merge function.

- **master.compute()**: Users can optionally subclass the Master class and implement the master.compute() function, which gets called at the beginning of each superstep. master.compute() can update the aggregators before they

3. THE GRAFT DEBUGGING TOOL

Figure 1 gives an overview of Graft’s architecture. In the following subsections we explain the architecture and components in terms of the capture, visualize, and reproduce functionalities they implement.

3.1 Capture: The DebugConfig File and Graft Instrumenter

Users extend and implement a DebugConfig class to specify the vertex values they are interested in capturing. Users can instruct Graft to capture all vertices in five categories: (1) vertices specified by their IDs, and optionally their neighbors; (2) a random set of a given number of vertices, and optionally their neighbors; (3) vertices that violate a specified constraint on vertex values; (4) vertices that send a message value that violates a specified constraint; and (5) vertices that raise exceptions. Alternatively, a user may specify that all active vertices should be captured. Users can also limit in which supersteps Graft captures vertices; by default Graft captures vertices in each superstep. For example, the DebugConfig shown in Figure 2 instructs Graft to capture 5 random vertices and their neighbors, and all vertices that send negative-valued messages, across all supersteps.

The Graft Instrumenter takes as input the user’s DebugConfig file and vertex.compute() function. It uses Javassist [15] to wrap the vertex.compute() around a new instrumented one, which is the final program that is submitted to Giraph. When Giraph calls vertex.compute() on the instrumented code of a vertex v, the code calls the user’s original vertex.compute() function, intercepting messages and value updates so it can check constraints. After the user’s vertex.compute() function returns, the instrumented function checks whether v should be captured: (1) if v is in one of the five possible categories of DebugConfig (above); or (2) if the user instructed Graft to capture all active vertices. To capture v, the instrumented code logs the context of v, along with the messages that v sent, to a trace file in the Hadoop Distributed File System (HDFS)—the distributed file system that is used by Giraph [10]. The context of a vertex.compute() function consists of the five pieces of data that the Giraph API exposes to a vertex (recall Section 2). We note that there are some limitations to capturing only this small set of data, which we discuss in Section 7. We also note that the design of our debugger assumes that the captured set of vertices is relatively small, both for usability and performance. As a “safety net”, we have built in an adjustable threshold, specifying a maximum number of captures, after which Graft stops capturing.

3.2 Visualize: Graft GUI

Users inspect the captured vertices through the Graft GUI in their browsers; screenshots are shown in Figures 3-5. Figures 3 and 4 are from a graph coloring algorithm, so the vertex values are color assignments. Figure 5 is from a connected components algorithm,
where the values are vertex IDs. The GUI’s Node-link View is for visualizing a small number of selected vertices and their graph structure. The Tabular View is for visualizing larger numbers of selected vertices. The Violations and Exceptions View is for inspecting those vertices that violate constraints or raise exceptions.

We explain these views in detail, and explain the Reproduce Context Buttons.

- **Node-link View (Figure 3):** Shows the vertices that were captured by ID or by random selection as a node-link diagram. The IDs and values of vertices are displayed on the nodes, and edge values (if any) are displayed on the links. If a vertex is inactive in the displayed superstep, its color is dimmed. For example, in Figure 3, vertex 567890 is active in superstep 231, while vertex 567891 is inactive. If u is a neighbor of a captured vertex v but u is not captured, then u is shown as a small node and only u’s ID is displayed. By clicking on a captured vertex, users can further see the incoming and outgoing messages of the vertex (omitted in the figure). In the upper-right corner, we see the aggregator values and default global data for the displayed superstep.

Users can replay how vertex values, active/inactive states, incoming and outgoing messages, and aggregators change superstep by superstep by pressing the Next and Previous superstep buttons. On the left side, there are three boxes labeled as M for message value constraint, V for vertex value constraint, and E for exception. Green labels indicate that no violation or exception has occurred on any vertex during the superstep that is being displayed, and red labels indicate otherwise. When a label is red, users can click on the label and go to the Violations and Exceptions View to see the vertices that violated the constraint or raised an exception (see below).

- **Tabular View (Figure 4):** If the user is debugging a large number of vertices, then the node-link diagram becomes difficult to use. The Graft GUI alternatively provides a tabular view of vertices, shown in Figure 4, where each row displays the summary of a captured vertex and can be expanded to see the entire context. This view also provides a simple search feature to find vertices by their IDs or their neighbors’ IDs, their values, or messages they have sent and/or received. Similar to the Node-link View, the Tabular View allows stepping through vertex contexts superstep by superstep.

  - **Violations and Exceptions View (Figure 5):** Tabular view of vertices that have violated a vertex value or message constraint, or have raised an exception. Shows the constraint-violating vertex or message value, or the error message and stack trace of the exception.

  - **Reproduce Context Buttons:** When the user wants to further investigate a vertex v, he clicks the “Reproduce Vertex Context” button that is available in all of the views. The GUI displays a piece of Java code the user can copy into his IDE to replay line-by-line which lines of vertex.compute() executed for the selected vertex. We will explain the details of this Java code in the next section.

### 3.3 Reproduce: Context Reproducer

When a user clicks on the Reproduce Vertex Context button for a vertex v in superstep i, Graft’s Context Reproducer takes the trace file of v from superstep i and generates a JUnit test file [16], with the help of Apache Velocity’s template-based code generation [29].

The generated code replicates the context of v through mock objects, then calls v.compute(). The user can copy this code into his IDE and use the IDE’s line-by-line debugger to see exactly which lines of v.compute() executed in superstep i. Using Mockito [19], the mock objects emulate the behavior of the different objects that the vertex.compute() code depends on, such as v’s data and ID, aggregators in superstep i, or the default global data exposed to v, thus replicating the context under which v.compute() executed in the cluster. (We will discuss a limitation of this approach in Section 7.)

Figure 6 shows an example JUnit test file replicating the context of a vertex with ID 672 in superstep 41 for a program called GCVertex (graph coloring). In lines 10-12, the mock GraphState object emulates the default global data that Giraph exposed to vertex 672, which consists of the superstep number (41), the total number of vertices (1 billion), and the total number of edges (3 billion). Lines 14-18 emulate the aggregators that were exposed to vertex 672. Lines 14-18 emulate the value, outgoing edges, and incoming messages of the vertex. Finally line 30 calls the user’s GCVertex.compute() function. We believe the ability to replay the line-by-line execution of vertex.compute() for a specific vertex and superstep is a very powerful feature of Graft.

Although JUnit is a standard testing framework to unit-test pieces of a vertex value constraint, and
4. DEBUGGING SCENARIOS

To illustrate Graft’s debugging capabilities, our demo runs through three debugging scenarios that cover different features of Graft. Two of the scenarios demonstrate how Graft helps users debug their Java code, our primary use of JUnit is not to unit-test vertex.compute(), but to reproduce the context of vertex.compute() for a particular vertex in a particular superstep. However, users can edit the JUnit test code generated by Graft and turn it into a real unit test.

### 3.4 Other Features

**Debugging Master.compute()**

If the Giraph program submitted to Graft contains a master.compute() function, Graft automatically captures its context—just the aggregator values—in every superstep. By clicking on the “Reproduce Master Context” button in the GUI, users can generate a JUnit test file reproducing master.compute() execution in a particular superstep. In our experience, the most common bug inside master.compute() is setting the phase of the computation incorrectly (recall Section 2), which generally leads to infinite superstep executions or premature termination.

**Small Graph Construction and End-To-End Tests**

We discussed that the JUnit test files Graft produces can be retained as real unit tests. In addition to unit tests, which test compute() functions in a single superstep, users may wish to write “end-to-end” tests. In an end-to-end test, a user may construct a small graph, then execute the program locally from first superstep until termination, to verify that Giraph’s final output is correct on the small graph. The Node-link View of the Graft GUI has an “off-line” mode to help users construct small graphs for testing purposes. Users can add vertices and draw edges between vertices, and edit the values of the vertices and edges in an easy fashion. Users can also select premade graphs from a menu. After constructing a graph in the GUI, the user can obtain either a text file that contains the adjacency list representation of the graph and use it in an end-to-end test, or can obtain an end-to-end test code template, which contains code that constructs the graph programmatically.

### 4.1 Graph Coloring Scenario

Our first scenario is based on a report from an actual Graft user and illustrates the feature of constructing the graph programmatically. Our implementation of GC contains a bug that incorrectly puts vertex 672 into the MIS. During line-by-line replay inside an IDE, we identify the buggy line; in this case, line 26.

#### Table 1: Graph datasets for demonstration.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Vertices</th>
<th>Edges</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>web-BS</td>
<td>685K</td>
<td>7.6M</td>
<td>A web graph from 2002</td>
</tr>
<tr>
<td>soc-Epinions</td>
<td>76K</td>
<td>500K</td>
<td>Epinions.com “who trusts whom” network</td>
</tr>
<tr>
<td>bipartite-1M-3M</td>
<td>1M</td>
<td>6M</td>
<td>A 3-regular bipartite graph</td>
</tr>
</tbody>
</table>

[Figure 6: JUnit test case generated by Graft Context Reproducer.]
the counters by one for each of its walkers, then sends the counters as messages to its neighbors.

To optimize the memory and network I/O, our implementation declares the counters and messages as 16-bit short primitive types. However, if a vertex \( u \) has a large number of walkers and the number of walkers from \( u \) to \( v \) exceeds Java’s max short value of 32767, then \( u \) might send \( v \) a negative number of walkers. To detect this bug using Graft, we run RW on the web-BS graph with a simple message value constraint that messages are non-negative. After the run we see that the message value constraint icon is red in some supersets, and in the Violations and Exceptions View we identify which vertices are sending negative messages. We generate a JUnit test case from a vertex \( v \) that has sent a negative message, and detect that the bug is due to overflowing of the short type counters.

### 4.3 Maximum-Weight Matching Scenario

Our third scenario illustrates that Graft can also be used to find errors in the input graph. We use an approximate maximum-weight matching algorithm from reference [23], which we call MWM. In this algorithm, the input is an undirected graph with edge weights. In each iteration of MWM, vertices select their maximum-weight neighbors. If \( u \) and \( v \) select each other, the edge \((u, v)\) is added to \( M \), and \( u \) and \( v \) (along with all edges incident to them) are removed from the graph. The iterations continue until there are no vertices left in the graph.

We run MWM on a weighted version of the soc-Epinions graph, which is encoded as undirected by having symmetric directed edges between every pair of adjacent vertices. However, a small fraction of the edges incorrectly have different weights on their symmetric edges. We run MWM on our erroneous soc-Epinions graph and see that it enters an infinite loop. We then run MWM with Graft and capture all active vertices after superstep 500, by which point the active graph is fairly small. We notice that some of the edge weights in the small remaining graph are asymmetric, which is the cause of the algorithm not converging.

### 5. PERFORMANCE

To evaluate the performance overhead of Graft, we ran the three algorithms from our scenarios with and without Graft, over large data sets. The graphs and the DebugConfig configurations we used are shown in Tables 2 and 3 respectively. We used a cluster of 36 machines, each with 30 GB of RAM, running Red Hat Linux OS and Giraph 1.0.0. We used the 3X software tool to automate and manage our experiments [26]. Our 3X repository, which contains the exact configurations of our experiments and can be used to reproduce them, is available to download [9].

Figure 7 shows a sample of our experiments. In the figure, each cluster of bars corresponds to one algorithm and dataset as indicated on the x-axis. Individual bars of the cluster show the overhead of running the algorithm and dataset on a particular DebugConfig or without Graft (“no-debug” bars). The heights of the bars indicate the relative total run-time of a specific experiment against the no-debug configuration (normalized to 1.0 for each experiment). The number on each bar indicates the total number of vertex captures in the experiment. For example, the last bar of the RW-tw cluster indicates that the overhead of running RW on the twitter graph with the DC-full configuration was 26% compared to running without Graft, and Graft captured 24213 vertices during the experiment. We ran each experiment five times and report the average runtimes for each experiment. The variances across runs were small and are shown by the error bars.

Overall, Graft’s overhead when capturing a set of 5 vertices randomly or by their IDs was less than 16% (DC-sp), and less than 17% when also capturing the neighbors of these vertices (DC-sp+nbr), across all of our experiments. The overhead was less than 20% when checking message and vertex value constraints (DC-vv and DC-msg). Finally, Graft’s overhead when capturing a specified set of vertices and their neighbors, and also the vertices that violate a message and vertex value constraint was less than 29% (DC-full). The number of captures varied between 1 and 1,246,151.

### 6. RELATED WORK

No previous work we know of proposes and implements a debugger designed specifically for the needs of Pregel-like systems, or any other distributed graph-processing system. We review some existing debuggers and explain why they cannot be used or adapted easily for Pregel-like systems.

#### 6.1 Remote Debuggers

Remote debuggers in existing development environments, e.g., [4][12], attach to pieces of code running on a specific port of a specific machine—information that is usually unknown to the users before launching their Giraph programs. Thus, it is difficult or impos-
sible to attach to a specific `vertex.compute()` function that the user is interested in. In addition, remote debuggers are interactive debuggers that halt the execution of the processes that they attach to. Thus, if a user puts a breakpoint into `vertex.compute()` and attaches to a Giraph worker, the worker halts its execution. This behavior may have two side effects. First, because the attached worker is not making progress, the master worker may assume that the attached worker has failed and switch to fault-recovery mode. Second, in Giraph and other Pregel-like systems, a pause of a single worker in a particular superstep causes all other workers to pause as well. Therefore as the user debugs his `vertex.compute()` code, all workers of Giraph are idle and keep consuming resources across the cluster. If there are multiple users of the cluster, those resources will not available to the other users during the entire debugging process.

### 6.2 Replay Debuggers

Replay debuggers provide offline debugging functionality. As we mentioned in Sections 1 and 6 existing replay debuggers, e.g., [2, 6], record low-level information such as all OS calls, network messages, and memory accesses throughout the execution of a distributed application. Then, using this recorded information, the user can replay the execution of any group of processes offline to find bugs in the distributed application. Some replay debuggers are designed to debug any distributed program and thus could be used to debug Giraph programs as well. However, when debugging Giraph programs, existing replay debuggers would incur unnecessarily high overhead recording a high volume of low-level information, most or all of which is irrelevant for bug-finding inside `vertex.compute()`. Also, existing replay debuggers do not provide specific visualizations for graphs, which we believe is very important for debugging in Pregel-like systems.

Graft can be thought of as a very lightweight replay debugger tailored for the needs of Giraph users. Instead of replaying the line-by-line executions of different Giraph workers, which existing replay debuggers would do, Graft can replay specific `vertex.compute()` calls. Thus, Graft only needs to capture a small amount of data, often in the kilobytes, even when debugging a computation across hundreds of workers. In addition, the Graft GUI is tailored to visualize graphs as opposed to generic distributed applications.

### 6.3 Debuggers of Dataflow Systems

Recently, two debuggers [1, 14] and one framework [20] for building debuggers for distributed dataflow systems have been introduced. We briefly review each of these systems, then explain why they could not be adapted easily for Giraph:

- **Inspector Gadget (IG):** IG [20] is a framework for building debugging tools for Pig [21], a dataflow system on top of Hadoop [10]. Some of the tools that have been programmed by IG allow users to trace input data records across operators to inspect how they are modified, build histograms over the output records of operators, and receive alerts when a record takes a long time to process inside operators. The IG framework is not designed to build tools that enable users to do line-by-line debugging within operators.

- **Daphne:** Daphne [14] is a set of debugging tools for DryadLINQ [30], a dataflow system on top of the Dryad [13] data-processing system. Daphne allows users to visualize and profile the resource consumption of different operators of their workflows. It also allows users to replay and line-by-line debug the processes that crash on their local machines.

- **Arthur:** Arthur [1] is a new debugger for Spark [31] and Hadoop [10] programs. Similar to some IG tools, Arthur lets users trace records across operators. Users can also use a line-by-line debugger to replay any process that they are interested in on their local machines, and users can inspect intermediate output of their dataflow programs. When a user issues a query over an intermediate output, Arthur first re-runs part or all of the dataflow program from the beginning to compute the relevant part of the intermediate output for the query and then runs the query. The query is a Spark program written by the user.

The functionalities provided by these debuggers are not a good fit for the needs of programmers of Pregel-like systems for several reasons. First, users of Pregel-like systems do not cast their graph computations as a workflow of operators that modify a set of input records. Second, `vertex.compute()` functions can be quite complex, and debugging them necessitates line-by-line debugging of specific `vertex.compute()` functions that users are interested in. Line-by-line debugging is either not provided by these tools, or is only provided at the worker level, as opposed to vertex level. Finally, similar to replay debuggers, debuggers of dataflow systems do not provide any specific visualizations for understanding graph algorithms.

### 7. SUMMARY, LIMITATION, AND FUTURE WORK

We presented Graft, a debugging tool for Apache Giraph. Graft helps users debug their `vertex.compute()` functions in three steps: first capturing the contexts of a specified set of vertices, then replaying the contexts of these vertices superstep by superstep until a bug is noticed, and finally reproducing exactly which lines of `vertex.compute()` ran on the buggy vertex and superstep inside a line-by-line debugger. Graft can also be used to debug `master.compute()` functions, find errors in input graphs, and generate end-to-end tests for Giraph programs.

Because the APIs of Pregel-like systems are implemented in standard programming languages such as Java or C++, users can, in principle, write `compute()` functions that depend on “external” data beyond the five pieces of data that are exposed by the API. For example, programs can read external data from an HDFS file in Giraph, or vertices that are executed on the same Giraph worker thread can share information amongst themselves through static Java objects. If a user’s `vertex.compute()` code has such external data dependencies, then the limited context information that Graft captures will not be enough to reproduce exactly what happened, which is a limitation of our approach. That said, we believe that such external data dependencies should be avoided in general, and every algorithm we have seen implemented in the code repositories of Pregel-like systems either had no external dependencies or could be implemented without such dependencies. There are ways, however, to capture external dependencies: we could, for example, employ techniques in some replay debuggers [2, 6] that capture enough context information to replay the entire execution of processes, which includes all function calls. For now, Graft is intended to be a very lightweight debugger, and employing techniques to capture and reproduce the complete data dependencies of `compute()` functions is beyond the scope of the initial system.

We outline three broad directions for future work.

- **More complex constraints:** During our interviews with Giraph users, several users expressed interest in more sophisticated vertex value and message constraints than what Graft currently offers. For example, in our GC algorithm, a user might wish to specify as a constraint that no two adjacent vertices should be assigned the same color. As another example, users
might want to express message constraints that depend on the value of the destination vertex, which Graft does not currently allow—message constraints can only depend on the ID of the destination vertex. Extending DebugConfig and the Graft Instrumenter to specify and verify such constraints should not be overly difficult and could be a useful extension.

- **Debugging other vertex-centric systems:** With only minor modifications the Graft code could be used for other Pregel-like systems, such as Apache Hama [11] or GPS [23], which contain only minor API differences from Giraph. Some vertex-centric graph-processing systems, such as PowerGraph [8] and GraphChi [17], support both synchronous and asynchronous computations. We believe a Graft-like debugger could also be helpful in debugging synchronous programs written for these systems. However, asynchronous computations have no clear superclass by superclass divisions; it would be a challenging and interesting future direction to design intuitive GUIs to visualize asynchronous computations.

- **Extending Graft’s unit-testing functionality:** During our interviews with Giraph users, we observed a prevalence of end-to-end tests, which verify the final output of their programs on small graphs, as opposed to unit tests, which verify the behavior of a vertex in a particular superclass and context. One reason is that it is much easier to construct a small input test graph, which is the input to an end-to-end test, than the entire context of a vertex in a particular superclass (mock vertex values, aggregators, global data, input messages, and neighbors of a vertex). However, our interviewees also recognized that at scale, `vertex.compute()` functions can be exposed to many contexts that are not tested during end-to-end tests on small graphs. We hope to extend the Graft GUI’s offline mode to make it easy to generate unit tests that can test `vertex.compute()` functions under contexts that emerge at large scale.

8. **ACKNOWLEDGEMENTS**

We are grateful to Pavan Athivarapu, Avery Ching, Sergey Edunov, Nishant Gandhi, and Maja Kabilijo for numerous interviews, discussions, and valuable feedback. We also thank Andrej Krevl for his assistance with a variety of system issues.

9. **REFERENCES**


