Heapviz: Interactive Heap Visualization for Program Understanding and Debugging

Abstract

Understanding the data structures in a program is crucial to understanding how the program works, or why it doesn't work. Inspecting the code that implements the data structures, however, is an arduous task and often fails to yield insights into the global organization of a program's data. Inspecting the actual contents of the heap solves these problems but presents a significant challenge of its own: finding an effective way to present the enormous number of objects it contains.

In this paper we present *Heapviz*, a tool for visualizing and exploring snapshots of the heap obtained from a running Java program. Unlike existing tools, such as traditional debuggers, Heapviz presents a global view of the program state as a graph, together with powerful interactive capabilities for navigating it. Our tool employs several key techniques that help manage the scale of the data. First, we reduce the size and complexity of the graph by using algorithms inspired by static shape analysis to aggregate the nodes that make up a data structure. Second, we implement a powerful visualization component whose interactive interface provides extensive support for exploring the graph. The user can search for objects based on type, connectivity, and field values; group objects; and color or hide and show each group. The user may also inspect individual objects to see their field values and neighbors in the graph. These interactive abilities help the user manage the complexity of these huge graphs.

By applying Heapviz to both constructed and real-world examples, we show that Heapviz provides programmers with a powerful and intuitive tool for exploring program behavior.

Keywords: Software visualization, program visualization, interactive graph visualization, interactive visualization, graph visualization, force directed layout

1 Introduction

Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.

– Fred Brooks[1]

Understanding modern software has become a significant challenge, even for expert programmers. Part of the problem is that today's programs are larger and more complex than their predecessors, in terms of static code base (lines of code), runtime behavior, and memory footprint. Another problem is that modern applications, such as web-based e-commerce and cloud computing platforms, are constructed by assembling reusable software components, ranging from simple container classes to huge middleware frameworks. In many cases, these components are instantiated dynamically and wired together using techniques such as reflection or bytecode rewriting. These features make it very challenging for any one programmer to obtain a global understanding of the program's state and behavior.

The size and complexity of software is also a major impediment to program understanding tools, particularly those based on static analysis of the code. The programming techniques described above often result in very imprecise information that is of little value to the programmer. Tools that analyze the dynamic behavior of programs have traditionally focused on identifying performance problems rather than on general program understanding[2,3,4]. The primary technique currently available for inspecting program state is the debugger, which is extremely painful to use for anything but the smallest data structures.

In this paper we present a new tool called *Heapviz* that is capable of effectively visualizing heap snapshots obtained from running Java programs. By visualizing the actual contents of the heap, we avoid the drawbacks of static analysis tools: the problems caused by dynamic software architectures and the inaccuracy of heap approximation. The main challenge of

our approach is the scale of the data: even a modest program can contain an enormous number of objects. We visualize the heap as a graph in which nodes represent objects and edges represent pointers (object references) between them. Our work leverages the Prefuse visualization toolkit[5], which provides a rich set of software tools for building interactive visualizations. Unlike traditional debuggers, Heapviz provides a global view of the data together with powerful interactive capabilities.

Our solution involves two techniques. First, we introduce algorithms for aggregating and abstracting individual objects to create a more succinct summary of the heap. For example, we might display all the elements of a large container using a single representative element. Second, we implement an interactive visualization that allows the user to (a) inspect individual objects and field values, (b) search for objects based on type and field values, (c) group objects and hide/show/color the groups, and (d) explore the connectivity of the object graph.

We demonstrate Heapviz on both constructed examples and real-world Java benchmark programs to evaluate its effectiveness as a tool for helping programmers visualize and navigate large, pointer-based data structures at both whole-program and individual data structure scales. This ability could greatly increase programmer productivity in many aspects of software construction and maintenance, including finding bugs and memory leaks, identifying opportunities to improve data structures, understanding the overall system architecture and interaction between software components, and helping new members on a development team come up to speed on the code quickly.

2 Related Work

Previous work on program analysis and understanding includes a number of techniques for visualizing the behavior of programs. A large body of prior research has focused primarily on

helping programmers navigate and visualize the *code*[6,7,8,9]. As many computing researchers and practitioners have observed, however, understanding the *data structures* of a program is often more valuable. Techniques for determining the structure of data in the heap fall into two main categories: static analysis and dynamic analysis. Static analysis algorithms, such as shape analysis[10,11], build a compile-time approximation of possible heap configurations. In many cases, however, these abstractions are too imprecise for detailed program understanding and debugging tasks.

Our work is most closely related to dynamic analysis tools that analyze the concrete heap (or a memory trace) in order to present the programmer with a graph or other visual representation of the actual state of the program. Since the main challenge for these tools is managing the scale of the data, the critical feature that distinguishes them is how they aggregate information for the user. Different choices lead to suitability for different tasks. Our specific goal for Heapviz is to help programmers understand the overall organization and structure of data.

Several existing tools provide programmers with an unabstracted graph representation of the concrete heap[12,13,14]. Without aggregation or interactive navigation, however, these visualizations do not scale well beyond a few hundred or thousand objects. Pheng and Verbrugge[15] present a tool with two visualizations of the heap. The first is a detailed graph of individual objects and pointers, with no abstraction. Nodes are displayed according to the shape to which they belong (list, tree, or DAG – from Ghiya and Hendren[10]). The second visualization consists of a line graph of the overall heap contents over time broken down by shape category. Their tool focuses on the evolution of the heap over time; Heapviz, on the other hand, aims to make a single snapshot of the heap comprehensible.

A number of existing heap visualization tools focus primarily on identifying memory utilization problems, such as memory bloat and memory leaks. The main difference between

Heapviz and these tools is that they give up much of the detail of the heap organization necessary to understand how the data structures work.

De Pauw et al.[16] present a tool aimed at finding spurious object references. The tool collapses the heap graph by aggregating groups of objects with similar reference patterns. It also supports interactive navigation, expanding and contracting of aggregated nodes. While similar in spirit, this tool is focused on finding spurious references and requires some programmer intervention to identify which references to record and display.

Several tools aggregate objects by ownership[17,18,19]. These tools first analyze the heap and build a model of object ownership, then aggregate objects that have similar patterns of ownership, type, or lifetime. The visualization typically presents the abstracted graph with annotations that summarize the properties of the aggregated nodes. The DYMEM memory visualization tool[20] shows the heap as a strict tree, duplicating subtrees as necessary, and uses an elaborate coloring scheme to indicate the amount of memory used and owned by groups of objects. These tools are often not well-suited for general program understanding, however, since they abstract away internal organization and individual objects.

Demsky and Rinard[21] present a heap visualization based on a dynamic analysis of object *roles*, or patterns of use in the program. The tool generates a role transition diagram, which focuses on object lifecycles, rather than the overall organization of the data. While this tool presents a unique view, scalability appears to be a concern for large programs.

Most closely related to Heapviz is the work of Marron et al.[22]. They process the heap graph using an abstract function previously developed for use in a sophisticated static pointer analysis. The analysis attempts to preserve information about internal structure (such as sharing) whenever nodes are collapsed.

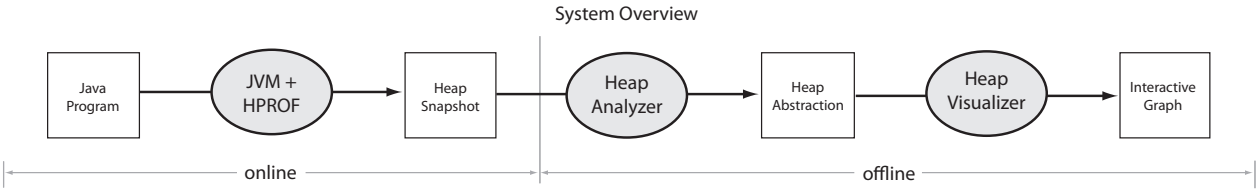

Figure 1: The Heapviz pipeline. The JVM and HPROF generate a heap snapshot from a running Java program. Our heap analyzer then parses this snapshot, builds a graph representation of the concrete heap, summarizes the graph, and outputs a heap abstraction. Our heap visualizer reads the heap abstraction and displays the graph, enabling the user to explore it interactively.

3 System Overview

In this section, we provide a brief overview of the architecture of Heapviz. We go into further detail in Sections 4 and 5. The full Heapviz pipeline (Figure 1) consists of three major parts:

1. **JVM + HPROF.** Generates a heap snapshot from a running Java program using the HPROF tool[4] provided by Sun with the Java Development Kit.

2. **Heap analyzer.** Parses the heap snapshot, builds a graph representation of the concrete heap, summarizes the graph, and outputs the resulting heap abstraction.

3. **Heap visualizer.** Reads the heap abstraction and displays the (possibly large) graph, allowing the user to explore it interactively.

Our *heap analyzer* parses the heap snapshot and builds a graph representation of the concrete heap. It then summarizes the concrete heap using the algorithm described in Section 4; we call this summarized graph a *heap abstraction*. Finally, the heap analyzer outputs the heap abstraction to GraphML[23], an XML-based graph format.

Our *heap visualizer* reads the heap abstraction from the GraphML file and displays the summarized graph. As described further in Section 5, it displays the graph using a force-directed layout and allows the user to explore the graph interactively, selecting nodes to view

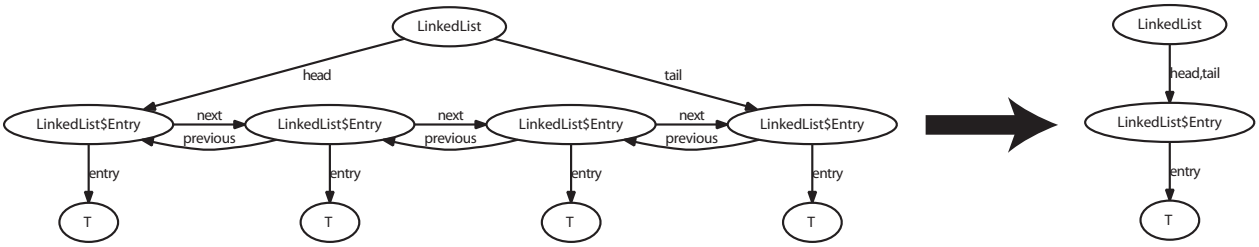

Figure 2: An example of applying our summarization algorithm to a linked list. Our algorithm first summarizes all `LinkedList$Entry` objects into a single node, then summarizes all `T` objects into a single node. Note that the summary would look the same regardless of the number of `T` objects in the linked list.

the contents of their fields, querying the graph for nodes of specific types or with certain field values, grouping nodes and applying visual filters to them, and displaying the connections among the objects in the heap.

4 Heap Analysis

4.1 Heap Snapshot

Our heap analysis starts with a heap snapshot obtained from a running Java program. The heap snapshot tells us which objects are currently in the heap, what their field values are, including the pointer values, and the values of all root references (static variables and stack references). In addition, the heap snapshot provides the runtime type of each object instance and the number of bytes needed to represent each object.

We use Sun's HPROF tool[4] to generate heap snapshots. HPROF is an agent that connects to a host Java virtual machine and uses the JVM Tool Interface[24] to enumerate all the objects in the program's heap. HPROF outputs the snapshot in a well-documented binary format that is supported by many third-party profiling and memory analysis tools. HPROF runs on top of any JVM that supports the JVM Tool Interface, so it is independent of JVM

implementation.

One of our goals is to allow the programmer to view the heap at any point in the program's execution. To support this capability we provide a mechanism for dumping a heap snapshot from within the program itself. Since the source to HPROF is provided with the Java Development Kit, we modify it to include a class `Dumper` with a static method `dumpHeap()` that generates the heap snapshot when called. The programmer adds a call to `dumpHeap()` in the source code at the point where a heap visualization is desired.

4.2 Heap Analyzer

4.2.1 Input Format

Our heap analyzer takes the heap snapshot from HPROF and parses it into a sequence of records. Of interest are the class records, which tell us details about the types of the objects in the heap snapshot, the object instance records, which give the types of these objects and their field values, and the root records, which tell us which heap objects are pointed to by root pointers (stack references and static references). Using these data, the heap analyzer builds a graph representation of the program heap, mapping object instances to graph vertices, pointer fields to graph edges, and roots to entry points in the graph.

4.2.2 Summarization Algorithm

Typical Java programs may contain 100,000, 1,000,000, or more live objects at any given point in program execution[25]; drawing all these objects would make the visualization too cluttered to comprehend and too slow to interact with. Our heap analyzer *summarizes* the graph to make visualization manageable.

Our summarization algorithm is designed to reduce the size of the graph while retaining the relationship among nodes. Each node in the summary graph represents a set of nodes in the concrete graph with the same runtime type and similar connectivity, and the edges in the summary graph represent sets of edges in the concrete graph. It works by merging nodes in the concrete graph according to a set of rules, and repeatedly applying those rules until it reaches a fixed point.

The rules for merging are:

1. **If there exists a reference from object o_1 to object o_2, and o_1 and o_2 are of the same type, merge o_1 and o_2.** This rule merges the recursive backbone of a data structure (e.g. the nodes of a linked list or the nodes in a tree).

2. **If objects o_1 and o_2 have the same set of predecessor objects (objects that point to o_1 or o_2) and are of the same type, merge o_1 and o_2.** This rule merges sets of objects that have the same type and the same connectivity (e.g. the objects contained by a data structure).

With these two simple rules, our system can compress very large graphs into more manageable ones. Consider the linked list data structure of Figure 2. This linked list contains four objects of type `T`. Our summarization algorithm first merges all the `LinkedList$Entry` objects into a single object using rule 1, and then all the `T` objects into a single object using rule 2. Note that the summarized graph looks the same no matter how many elements the linked list contains — whether 4, 400, or 40,000. However, if the linked list contains objects of different types, the summarized graph will contain separate nodes for the set of objects of each type.

4.2.3 Output Format

We write the summarized graph in GraphML format[23] for the heap visualizer to import and display.

5 Heap Visualization

Given a heap abstraction in GraphML format from the analyzer described in Section 4, Heapviz creates an interactive force-directed layout. The goals of the visualization are to create (1) an intuitive display of summarized heap data, and (2) an interactive environment where heap data can be easily explored. Heapviz builds upon the Prefuse toolkit[5]. Our implementation focuses on facilitating program understanding and debugging.

5.1 Graph Display

5.1.1 Force-Directed Layout

We selected force-direction as the layout algorithm because heap abstractions may be arbitrary graphs. The prior version of Heapviz [citation withheld for blind submission] used a radial layout and imposed a tree structure on the graph by computing a dominator tree. We found that such a layout was overly rigid and resulted in somewhat confusing graphs. Instead, force-direction minimizes edge crossings in the graph (i.e., makes it as planar as possible) to make the structure clear to the user. In our experience, heap abstractions tend to be planar, near-planar, or can become planar with minimal work from the user.

Efficient use of screen space is important, as even after summarization the graph may contain several hundred nodes. Force-direction strikes a balance between maximizing screen space

usage and preserving the structure of the heap abstraction.

5.1.2 Visual Encoding

Instead of relying on a dense visual encoding to convey information to the user, Heapviz displays the bare minimum of expected visual properties and relies on the user to modify the graph to suit his own individual needs through the many interactions that are provided.

Visual encoding of the graph focuses primarily on node size, labels, and color. The schemes that control each of these properties can be selected by the user. Node size may be constant, a function of the node's size in bytes, or a function of the number of represented instances. Labels display the type of each object. Color carries many different meanings, including differentiating user-defined node groups (see Section 5.2.2), selected nodes, and nodes being hovered over.

5.2 Interaction

Heapviz depends on rich interactions to make sense of complex heap data. Its interactions can be grouped into three major categories: graph display, groups, and query and selection. All three types provide a method to quickly examine the member variables of nodes through tooltips in their respective GUI elements.

The supplemental video demonstrates how the user can interact with Heapviz. Because our work relies on the user's being able to explore the graph interactively, we recommend that the reader view the video to have a better understanding of how Heapviz works and how it can be used.

5.2.1 Graph Display

Heapviz supports many of the interactions one would expect from an interactive node-edge display in order to make using Heapviz as easy and fluid as possible. Among these are the ability to pan, zoom, drag nodes and bring up context menus. We support node selection in a variety of ways, including selecting singly, in an area, or based on reachability from given source nodes. The user may tweak the physics simulation to suit the nature of the graph by modifying the simulation parameters through sliders. Edges currently do not support any interactions.

5.2.2 Groups

Once selected, nodes can be put into groups. This helps the user classify nodes in a meaningful way, extended existing interaction modes with further semantic information and allowing many new interactions. For each group, the user can hide it (to reduce clutter), color it (for easy identification) or combine it with other groups (via union and intersection). Groups provide a familiar and robust framework for adding new interactions or augmenting existing ones.

5.2.3 Query and Selection

Nodes carry a large amount of information on the examined program's state. Heapviz provides a simple, yet powerful query language to harness that information, helping the user find and select nodes that satisfy the query. The language supports querying based on structure alone (e.g., `degree() > 3`), member variables of the represented objects (e.g., `my_boolean == false`), type or package information of the objects (e.g., `type() == "HashMap"` or `package() == "java.lang"`), or group information (e.g., `ingroup("group1")`). Queries can be chained

together with Boolean connectors to create more complex ones, allowing the user the flexibility and power to pinpoint the nodes of interest and to create any desired intermediate groups.

6 Case Studies

We now present the results of visualizing data structures in several Java programs and use these as a basis for discussion of Heapviz's strengths and weaknesses. First, we show three constructed examples built using standard Java container classes. Second, we explore three real programs: _209_db[26], SPEC JBB 2000[27], and Heapviz itself.

6.1 Constructed Examples

We first consider three examples constructed from standard data structures from the Java class library. In the first example, Heapviz reveals a difficult-to-find performance bug in a hash table with little effort on the user's part. In the second example, the user takes advantage of Heapviz's interactive abilities to explore the effect of insertion order on the structure of a red-black tree. In the final example, Heapviz's summarization algorithm makes clear the sharing of elements contained in three distinct data structures.

6.1.1 Hash Table

Hash tables are widely used in production software, and their average case time complexity is well understood by programmers. Furthermore, programmers understand in the abstract that using a poor hash function will cause too many elements to hash into the same bin, resulting in $O(n)$ time complexity in the worst case. But suppose a programmer is debugging a program

that uses a hash table, and she wants to determine whether that hashtable has elements distributed evenly into bins. Using a debugger to do this would be extremely tedious, and it would be difficult to instrument library hash table code. Instead, we can use Heapviz to visualize the hash table and see whether elements are evenly distributed among bins.

Figures 3 and 4 show Heapviz visualizations of two different Java `HashMap`s containing 100 elements each. The `HashMap` in Figure 3 contains elements that use a good hash function, and the one in Figure 4 contains elements that use a poor hash function. We can clearly see in the bad `HashMap` that many elements have hashed into the same bucket, and this has produced a long hash chain in that bucket. Thus, if the program performs an insert or lookup on an element that hashes into that bucket, it will pay an $O(n)$ cost instead of the expected $O(1)$ cost. This is obvious from a glance at the Heapviz visualization. Indeed, by examining the field values of the `HashMap$Entry` objects in the long chain, the programmer can easily determine which bucket is experiencing excess collisions and adjust the hash function accordingly.

6.1.2 Red-Black Tree

A red-black tree is a type of self-balancing binary search tree that is often used to implement a map data structure. To maintain the balanced property, a set of *tree rotations* may be performed when inserting or removing a node from the tree. In a red-black tree, internal tree nodes are colored either red or black, and every path from a given node to a leaf must contain the same number of black nodes. Thus, the worst case height of the tree is $2 \log n$, which occurs when all nodes are black except for those along one path of alternating red and black nodes.

Suppose a student wants to learn about red-black trees, specifically how to construct a tree in both the best and worst cases. It would be impractical to examine manually a

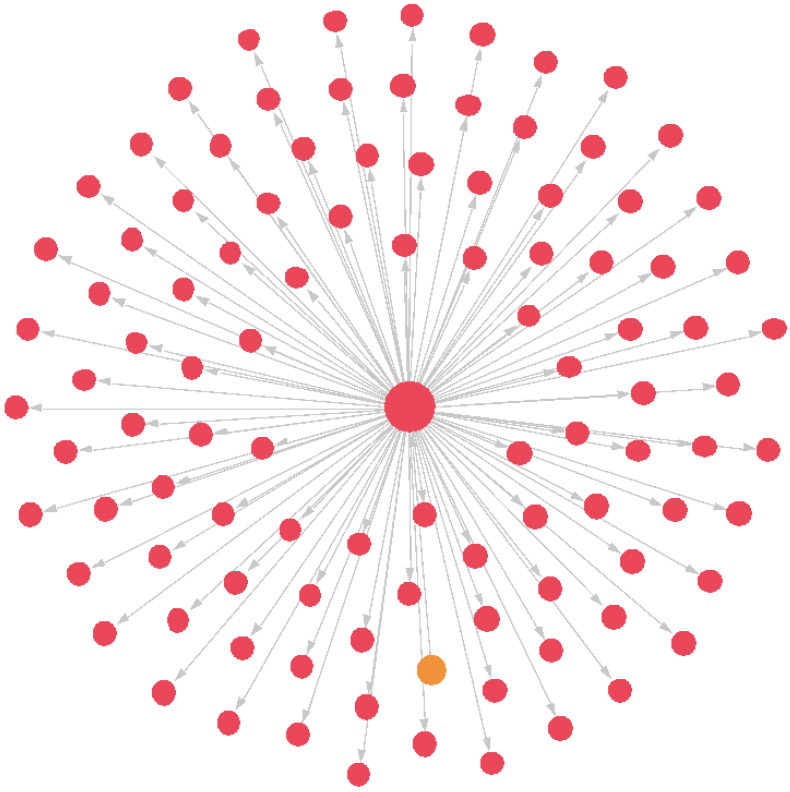

Figure 3: A hash table with a good hashing function. The orange node is the `HashMap` object, the large node at the center is the array of buckets, and the red nodes on the perimeter are the internal `HashMap$Entry` objects used to implement the hash chains. All the `HashMap$Entry` nodes are a short distance from the array of buckets, indicating that the hash function has distributed keys evenly.

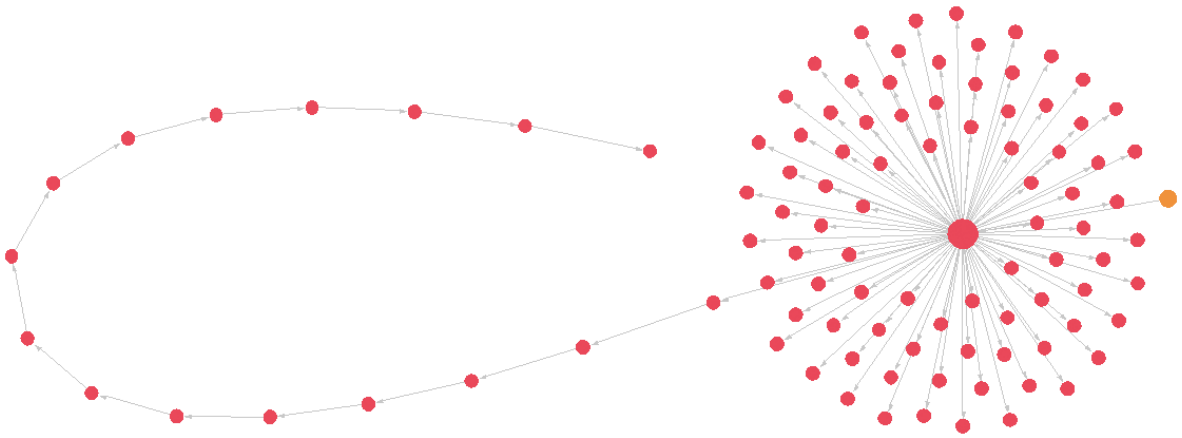

Figure 4: A hash table with a bad hashing function. As in Figure 3, the orange node is the `HashMap` object, the large node at the center is the array of buckets, and the red nodes on the perimeter are the internal `HashMap$Entry` objects used to implement the hash chains. One hash chain is much longer than the others, indicating an excess of collisions into that bucket.

reasonably-sized tree using a debugger, but a visualization of the tree with the nodes colored appropriately would make it obvious whether we have a "good" or "bad" tree. Heapviz's interactive tools make it easy to create such a visualization.

Below, we list the five steps a Heapviz user would perform to transform an unsummarized heap graph into an effective diagram of the contained red-black tree. A figure accompanies each step, and our supplemental video demonstrates this procedure. The process takes no more than 90 seconds for an experienced user.

1. The user loads the graph into Heapviz. Default colors are assigned to connected subgraphs to make separating them easier. Notice the main graph component separating into two subsections. (Figure 5)

2. To make navigation easier, the user hides the components that are not connected to the main component. She then hovers over the large `Integer` array node, highlighting its neighbors. The connectedness of the tree and the highlighted `Integer` objects tells us that the tree's data is these `Integer`s. (Figure 6)

3. Being interested in only the structure of the tree, the user selects all the `Integer`s at once and hides them. Now, we see a central node of type `java.lang.Object` holding the graph together. Red-black tree leaf nodes cannot contain data, and many implementations use a single sentinel node to represent all leaves. The user deduces that this central node is the sentinel node and is not important to the structure of the tree. (Figure 7)

4. The user hides the sentinel node, and the graph lays itself out. Immediately the tree structure becomes obvious. Since all the nodes are the same type, the user has opted to use circles to represent nodes instead of labels to reduce clutter. (Figure 8)

5. With two simple queries and a few clicks, the user colors each internal node red or black according to a field value. Now the tree looks like we would expect a red-black tree to look. (Figure 9)

Now that we can visualize a red-black tree, we can answer our question about the best and worst cases. Figure 9 shows a tree that was constructed by inserting keys in random order. The red nodes are evenly distributed throughout the tree, and all paths from the root to a leaf have approximately the same length. This is the best case. Figure 10 shows a tree that was constructed by inserting keys in ascending order. The nodes in this tree are all black except for those along one path in which nodes alternate between red and black. This is the worst case because the length of the alternating path is $2 \log n$, while all the others are $\log n$. Thus, by exploring with Heapviz we have learned that inserting keys in order produces the worst case red-black tree.

6.1.3 Overlapping Elements

Consider a program that contains multiple data structures. These data structures each contain some subset of the data objects in the program, and these subsets may overlap. That is, of the objects in data structure x, some are pointed to only by x, some by x and y, and some by x and z. Heapviz can explain this situation.

Figure 11 shows such a program. This program has three data structures, a `TreeMap`, a `HashSet`, and a `LinkedList`, using the standard implementations in the Java class library. These data structures contain `IntBox` objects, some of which are shared by two or all three data structures. The unsummarized graph on the left in Figure 11 gives too much detail to discover this sharing, but the summarized graph makes the sharing clear. Each `IntBox` node in the summarized graph represents a set of concrete nodes with the same set of concrete predecessor nodes. Thus the summarized graph explains the sharing of `IntBox` nodes among

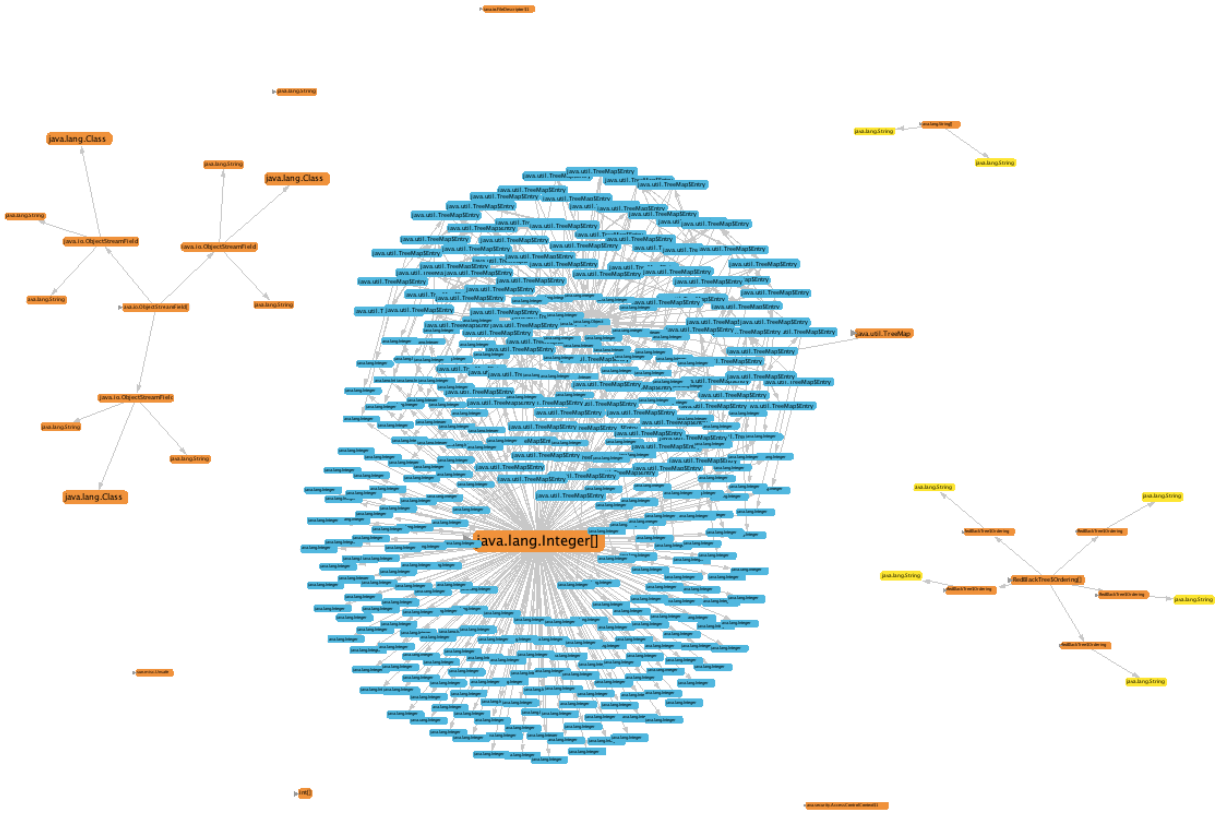

Figure 5: The initial state of a Heapviz visualization of a red-black tree. Notice the main graph component separating into two subsections.

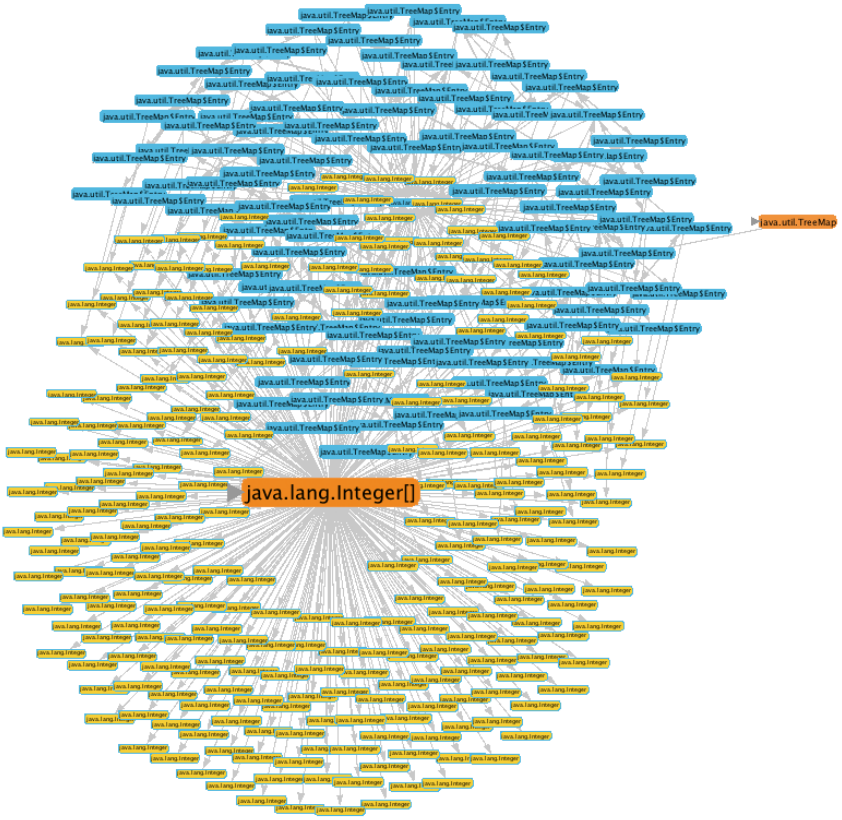

Figure 6: A Heapviz visualization of a red-black tree while a hover action highlights the immediate neighbors of the large `Integer` array at the center of the graph. This shows that the tree's data is in the `Integer`s.

Figure 7: A Heapviz visualization of a red-black tree after hiding the `Integer` objects (to focus on the structure of the tree). A single sentinel node of type `java.lang.Object` ties the graph together.

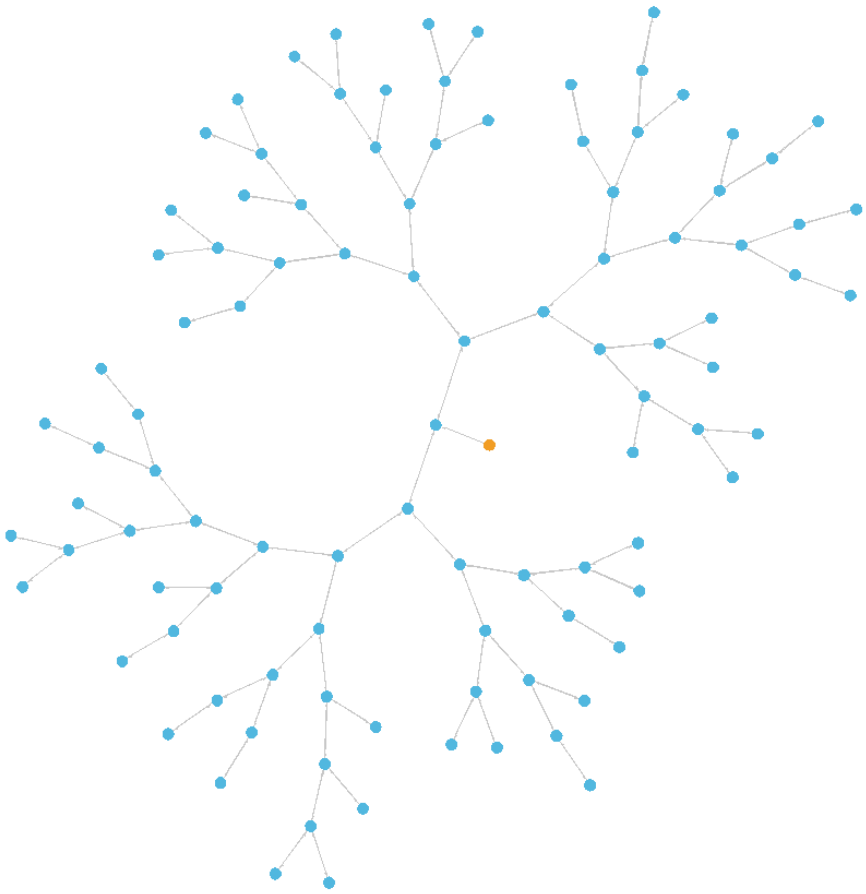

Figure 8: A Heapviz visualization of a red-black tree after hiding the sentinel node and setting the node display style to circles instead of labels. The tree structure is now obvious.

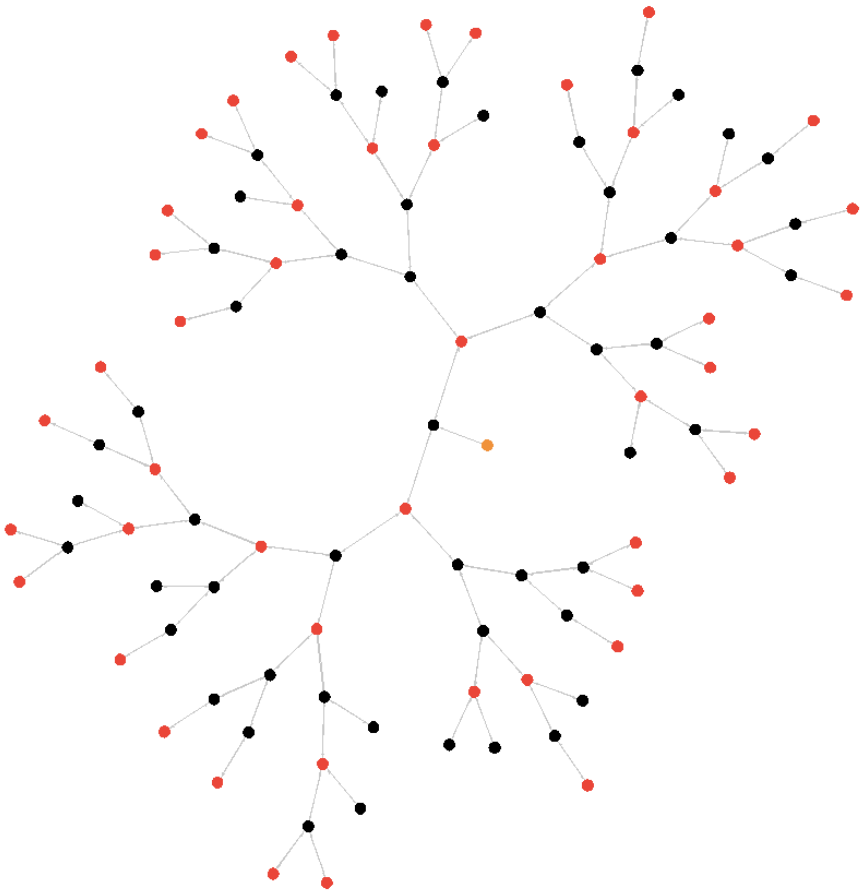

Figure 9: A Heapviz visualization of a red-black tree after coloring the nodes according to a field that indicates whether each node is red or black. Now the graph looks just as we would expect a red-black tree to look. Note that the red nodes are evenly distributed in the tree. This red-black tree was created by inserting keys in random order.

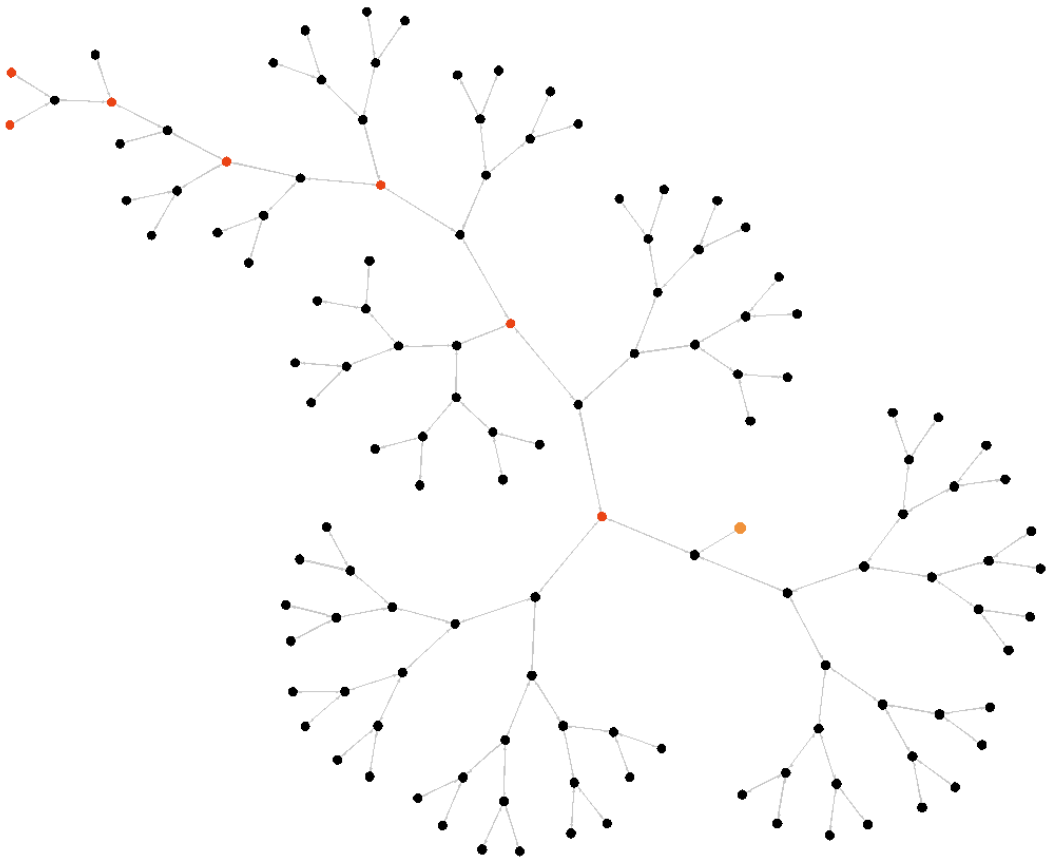

Figure 10: A Heapviz visualization of a red-black tree where keys were inserted in ascending order. All nodes are black except for those along one path, where red and black nodes alternate—the worst case.

the different data structures. For example, we can see that the `IntBox` objects represented by node a are pointed to only by the `HashMap`, but the ones represented by nodes c and d are pointed to by both the `HashMap` and the `LinkedList`. Thus we can determine that all `IntBox` objects in the `LinkedList` are also in the `HashMap`, but not all of the ones in the `HashMap` are in the `LinkedList`. In the interactive visualization, the user can hover over nodes to see exactly how many `IntBox` nodes are shared among the data structures, and how many are only in the `HashMap`.

From the nodes and edges in Figure 11, without looking at the source code of the program, we can determine that:

1. There are three data structures in the program: a `TreeMap`, a `HashSet`, and a `LinkedList`.
2. Each data structure contains objects of type `IntBox`.
3. Some `IntBox` objects (the ones represented by node a) are pointed to by only the `HashMap`. The user could find the exact number by hovering over node a.
4. Some `IntBox` objects are shared by all three data structures (node c), and others are shared by only two (nodes b and d). Again, the exact number can be found by hovering over the appropriate summary node.
5. The `HashMap` contains all `IntBox` objects in the program. The other data structures each contain a strict subset of the `IntBox` objects.

Heapviz makes clear the sharing among different data structures without requiring the user to look at the program source code.

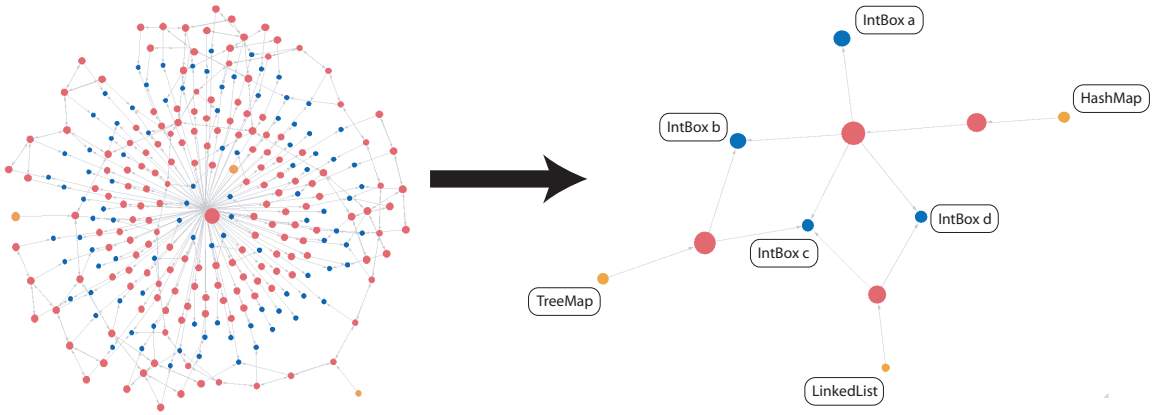

Figure 11: A Heapviz visualization of three data structures: a `TreeMap`, a `HashSet`, and a `LinkedList`. Each data structure contains a subset of `IntBox` objects. The unsummarized graph is on the left, and the summarized graph is on the right. Heapviz reveals the sharing among different data structures without requiring the user to look at the program source code.

6.2 Real Examples

Now we assess Heapviz's visualization of three real-world programs, _209_db[26], SPEC JBB 2000[27], and Heapviz itself. We consider how Heapviz is successful in helping us understand the data structures in _209_db, why it is less successful in summarizing the graph from SPEC JBB 2000, and how it can be used to diagnose its own excessive memory use.

6.2.1 _209_db

_209_db is a database benchmark from the SPEC JVM 98 benchmark suite[26]. It performs multiple database operations on an in-memory database. We took a heap snapshot of an execution of _209_db just after the database is built from an input file and before any operations are performed on the database. Figure 12 shows the summarized visualization on the top left, with the inset zoomed to show the program's primary data structure.

Before summarization, the graph contained 294,002 objects; after summarization, it contained

254. Because nodes in the visualization are sized by the amount of data they represent, it is easy to pick out visually the primary data structure in the program: an object of type `spec.benchmarks._209_db.Database`. This database contains two `Vector` objects (`Vector`s are growable arrays of objects), which each contain strings or database entries. The node that represents all database entries in this database summarizes 15,332 nodes. The database entries then each contain a `Vector` of strings; the total number of these strings in the database is 122,656.

By inspecting the source code of _209_db, we can explain what these data mean. One of the two `Vector` objects pointed to by the database is used to store a format string that describes the records included in each entry. The other `Vector` holds the database entries. Each database entry uses a `Vector` to hold strings for each record: name, address, city, state, and so on. Though we had to look at the source code to understand this program, the visualization quickly showed us the primary data structure in the program and its high-level structure.

6.2.2 SPEC JBB 2000

The SPEC JBB 2000[27] benchmark emulates a three-tier client-server system, with the database replaced by an in-memory tree and clients replaced by driver threads. The system models a wholesale company, with warehouses serving different districts and customers placing orders. We took a heap snapshot of SPEC JBB 2000 during the `destroy()` method of the `District` class in order to understand the sharing of `Order` objects stored by the `District`.

A full visualization of the SPEC JBB 2000 heap snapshot is shown in Figure 13. From the 117,819 objects in the concrete heap at this point in program execution, we produce a summarized graph of 7578 nodes, a reduction of 93.5%. Although some large data structures

are visible after application of the summarization algorithm, the graph is still too visually complex.

The characteristics of this data provide some clues as to limitations of the summarization algorithm. SPEC JBB 2000 represents an extreme form of the simple program discussed in Section 6.1.3, with a dense web of connections among what we could consider the "leaves" of the data structures—objects that represent the individual elements of the program, such as customers and orders. In SPEC JBB, many of these "leaves" point to other "leaves" and have a one-to-one relationship. This one-to-one pointer mapping gives each of these "leaves" a unique predecessor set, preventing them from being summarized.

For example, consider the `Order` objects in SPEC JBB. An `Order` object points to the `Customer` who made that `Order`, and a `Customer` object points to the last `Order` the `Customer` made. Thus, all `Customer` objects point to different `Order` objects, which then point back to the `Customer` object. This situation is shown in Figure 14.

Because `Customer` objects point to different `Order` objects, any `Order` pointed to by a `Customer` has a unique predecessor set and will not be merged with any other `Order`, unless the `Customer` objects are merged first. But because these `Order` objects point back to different `Customer` objects, those `Customer` objects also have unique predecessor sets and will not be merged. As a result, our algorithm cannot merge nodes that exhibit this one-to-one structure. We see this exact behavior in our graph: None of the three hundred `Order` objects are summarized because each of them points to and is pointed to by a different `Customer` object. We discuss possible solutions to this problem in Section 7.

6.2.3 Heapviz on Heapviz

Our early prototypes of the Heapviz heap analyzer component required too much memory to summarize large graphs, such as those for _209_db and SPEC JBB 2000. Having constructed

a debugging tool and finding ourselves needing to debug it, we decided to apply Heapviz to itself. We produced a heap snapshot of the analyzer working on our "Overlapping Elements" example from Section 6.1.3, immediately after the analyzer had generated the summarized graph. The resulting visualization of the summarized graph is shown in Figure 15.

To debug our excessive memory usage problem, we first looked at nodes that represent a large amount of memory. Heapviz sizes nodes by the total volume of data summarized by that node, so it is visually obvious which nodes represent the most memory. In this graph, we immediately see a large node of type `Value`, occupying 2.1 MB. Another node of the same type occupies 291 KB. We conclude that reducing the size and/or number of `Value` objects will reduce memory usage.

Examining the code for the `Value` class, we find this:

```
public class Value {

  public Type type;

  private long objVal;
  private boolean boolVal;
  private char charVal;
  private float floatVal;
  private double doubleVal;
  private byte byteVal;
  private short shortVal;
  private int intVal;
  private long longVal;
  ...
}
```

`Value` is being used to represent Java values of various types. Here it is essentially being

used as a tagged union type; sometimes it represents an object value, sometimes an boolean value, etc., and which kind of value it is representing is signified by the `type` field. With this representation, each instance of `Value` must reserve space for every possible type of value that might be represented. For example, if a particular `Value` instance represents an object value, it still stores empty fields for a boolean value, a char value, a float value, and so on. This instance uses 38 bytes of memory even though only 8 bytes are required. With our heap containing 58,456 instances of type `Value`, this inefficient representation incurs a significant memory overhead.

Based on this finding, we refactored the `Value` class into an abstract parent class `Value` and concrete subclasses for each type of value, each containing only the necessary fields for that type. Re-running Heapviz on the refactored version results in Figure 16. Here we can see that `Value` has been broken up into several subclasses, and by probing the nodes we can quickly determine that all of these instances combined now use 177 KB, down from over 2 MB.

7 Future Work

We have demonstrated Heapviz's performance on large programs; however, our summarization algorithm cannot greatly compress the size of graphs for programs containing a large number of nodes that have unique predecessor sets. With highly-connected graphs, such as the SPEC JBB 2000 benchmark (Figure 13), it is difficult to see the results of a query, even with highlighting of the relevant nodes. Our current summarization algorithm applies the same set of rules for merging nodes to all heap graphs, regardless of complexity. We plan to experiment with adjustable levels of detail, in which the summarization algorithm applies increasingly-powerful abstraction rules to a graph until it reaches a certain thresh-

old of complexity, measured in number of nodes and edges. For example, Heapviz currently preserves sharing information among nodes, but for complex graphs like SPEC JBB 2000 it could discard sharing information to produce a more manageable visualization.

In addition to adjustable levels of detail, we plan to augment our visualization component to allow the user to dynamically summarize and unsummarize portions of the graph. For example, if the user initially asks for a summarized graph, but then locates a data structure of interest and wants to see its unsummarized structure, Heapviz should allow her to expand the summarized nodes in the graph through some interaction with the visualization tool.

Another direction of future work is to enable viewing of the evolution of a heap graph over time. Currently, Heapviz supports visualizing only a single static snapshot of the program heap. It might be useful to also support an animation facility showing how the heap changes as the program executes. The major challenge with this would be to design a summarization algorithm that remains stable as nodes and edges are added to and removed from the graph.

Finally, because our long-term goal with Heapviz is to produce a tool that will be useful to developers, we intend to release Heapviz to the developer community. In order to reach this goal, we plan to conduct formal user studies to determine what improvements Heapviz requires before it can be released.

8 Conclusions

We have presented a tool for helping programmers analyze, visualize, and navigate heap snapshots from running Java programs. Heapviz enables users to navigate large, pointer-based data structures at a whole-program scale. We have introduced a heap analyzer, which parses a heap snapshot, builds a graph representation, and applies algorithms to create a summarized heap abstraction. We have demonstrated how to navigate this abstraction with

a heap visualizer which supports several interaction styles, including detailed field view, search, and node grouping.

Heapviz builds on a body of prior work on tools for debugging, static analysis, and data structure visualization. Our system makes several key contributions. Unlike traditional debuggers, we provide both an overview and detail; Heapviz provides the global view of the actual heap contents, as well as the ability to examine detail on demand with the field view. This variable level of detail supports programmer productivity in many common usage scenarios, including finding bugs and memory leaks, identifying data structures that could be improved, and understanding the overall system architecture. We hope that further exploration of domain-specific usage scenarios will spark the development of new analysis and visualization techniques.

Funding Acknowledgement

This work was supported by the National Science Foundation under grant CCF-1018038.

References

[1] Frederick P. Brooks, Jr. *The Mythical Man-Month.* Addison-Wesley, 1975.

[2] Quest. JProbe Memory Debugger. `http://www.quest.com/jprobe/`, Retrieved July 22, 2010.

[3] Gary Sevitsky, Wim De Pauw, and Ravi Konuru. An information exploration tool for performance analysis of java programs. In *TOOLS '01: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 85, Washington, DC, 2001. IEEE Computer Society.

[4] Kelly O'Hair. HPROF: A heap/CPU profiling tool in J2SE 5.0, 2004. `http://java.sun.com/developer/technicalArticles/Programming/HPROF.html`, Retrieved April 30, 2010.

[5] Jeffrey Heer. Prefuse: a toolkit for interactive information visualization. In *CHI '05: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 421–430, 2005.

[6] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 326–337, 1993.

[7] M.-A. D. Storey and H. A. Muller. Manipulating and documenting software structures using shrimp views. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*, page 275, Washington, DC, 1995. IEEE Computer Society.

[8] Wim De Pauw and John M. Vlissides. Visualizing object-oriented programs with Jinsight. In *The European Conference on Object-Oriented Programming*, pages 541–542, 1998.

[9] Steven P. Reiss and Manos Renieris. Jove: Java as it happens. In *ACM Symposium on Software Visualization*, pages 115–124, 2005.

[10] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *ACM Symposium on the Principles of Programming Languages*, pages 1–15, 1996.

[11] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM Symposium on the Principles of Programming Languages*, pages 105–118, 1999.

[12] Thomas Zimmerman and Andreas Zeller. Visualizing memory graphs. *Lecture Notes in Computer Science - Software Visualization*, 2269:533–537, 2002.

[13] Ali S. Erkan, T. J. VanSlyke, and Timothy M. Scaffidi. Data structure visualization with LaTeX and Prefuse. In *ITiCSE '07: Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 301–305, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-610-3. doi: http://doi.acm.org/10.1145/1268784.1268870.

[14] Jaishankar Sundararaman and Godmar Back. HDPV: Interactive, faithful, in-vivo runtime state visualization for C/C++ and Java. In *SoftVis '08: Proceedings of the 4th ACM Symposium on Software Visualization*, pages 47–56, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-112-5. doi: http://doi.acm.org/10.1145/1409720.1409729.

[15] Sokhom Pheng and Clark Verbrugge. Dynamic data structure analysis for Java programs. In *Proceedings of IEEE International Conference on Program Comprehension*, pages 191–201, 2006.

[16] Wim De Pauw and Gary Sevitsky. Visualizing reference patterns for solving memory leaks in Java. In *The European Conference on Object-Oriented Programming*, pages 116–134, 1999.

[17] Derek Rayside, Lucy Mendel, and Daniel Jackson. A dynamic analysis for revealing object ownership and sharing. In *WODA '06: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, pages 57–64, 2006.

[18] Trent Hill, James Noble, and John Potter. Scalable visualizations of object-oriented systems with ownership trees. *Journal of Visual Languages and Computing*, pages 319–339, June 2002.

[19] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. Making sense of large heaps. In *The European Conference on Object-Oriented Programming*, pages 77–97, 2009.

[20] Steven Reiss. Visualizing the Java heap to detect memory problems. In *VISSOFT '09: Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 73–80, 2009.

[21] Brian Demsky and Martin Rinard. Automatic extraction of heap reference properties in object-oriented programs. *IEEE Transactions on Software Engineering*, pages 305–324, 2009.

[22] Mark Marron, Cesar Sanchez, and Zhendong Su. High-level heap abstractions for debugging programs. 2010. `http://software.imdea.org/~marron/papers/publications.html`, Retrieved April 30, 2010.

[23] Graph Drawing Steering Committee. The GraphML file format, 2010. `http://graphml.graphdrawing.org`, Retrieved April 30, 2010.

[24] Sun Microsystems. JVM tool interface, 2010. `http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html`, Retrieved April 30, 2010.

[25] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, 2006.

[26] *SPECjvm98 Documentation*. Standard Performance Evaluation Corporation, release 1.03 edition, 1999.

[27] *SPECjbb2000 Documentation*. Standard Performance Evaluation Corporation, release 1.01 edition, 2001.

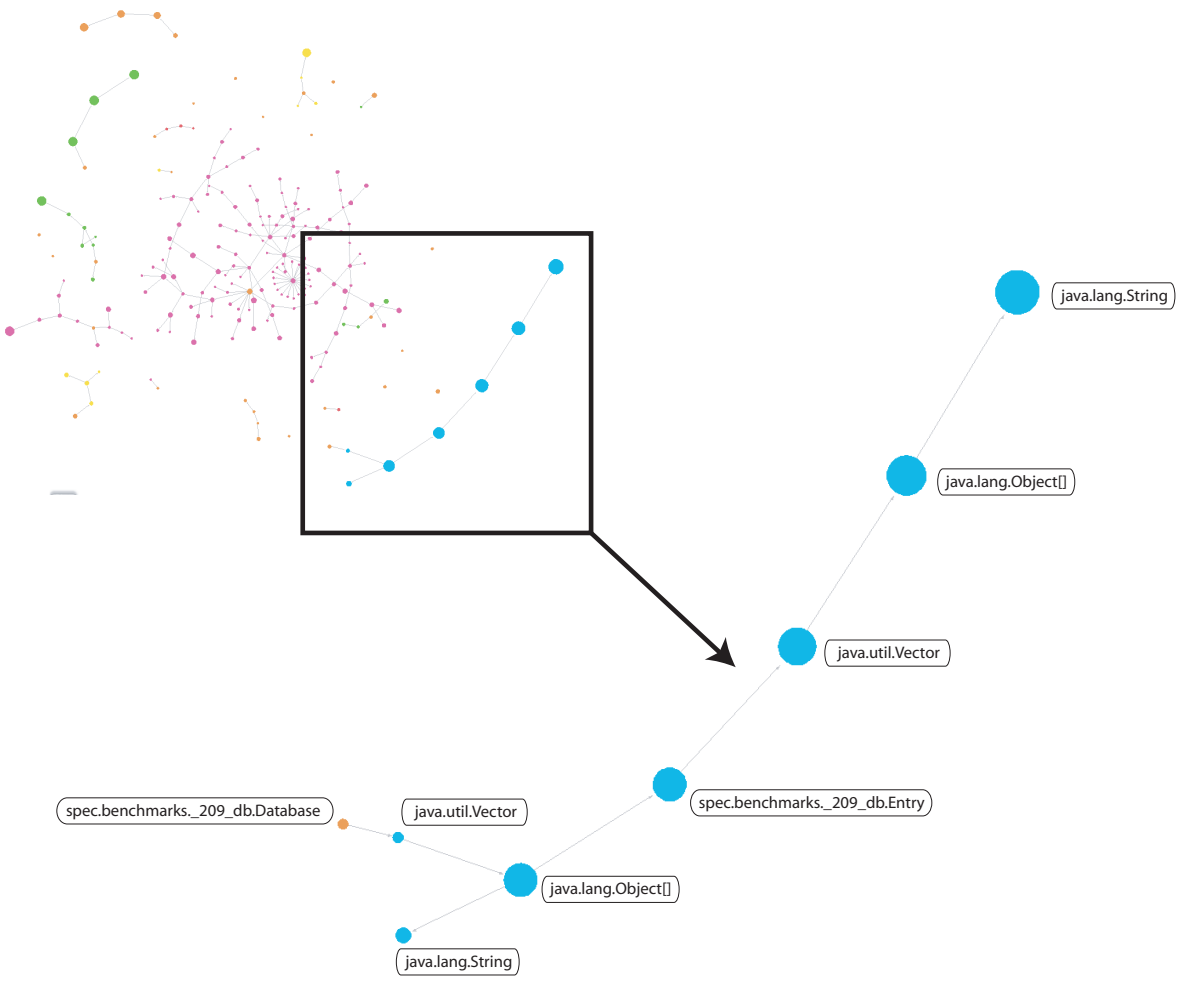

Figure 12: A summarized visualization of the _209_db benchmark. The top left shows the full graph, with the inset zoomed to show the program's primary data structure, a database. This database contains two `Vector` objects, each containing strings or database entries. The node representing all entries in this database summarizes 15,332 nodes. Each database entry contain a `Vector` of strings; the total number of these strings in the database is 122,656.

Figure 13: A summarized visualization of the SPEC JBB 2000 benchmark, which contains 117,819 objects in the concrete heap at this point in program execution. The summarized graph contains 7578 nodes. Although some large data structures are visible after this significant reduction, the graph is still cluttered.

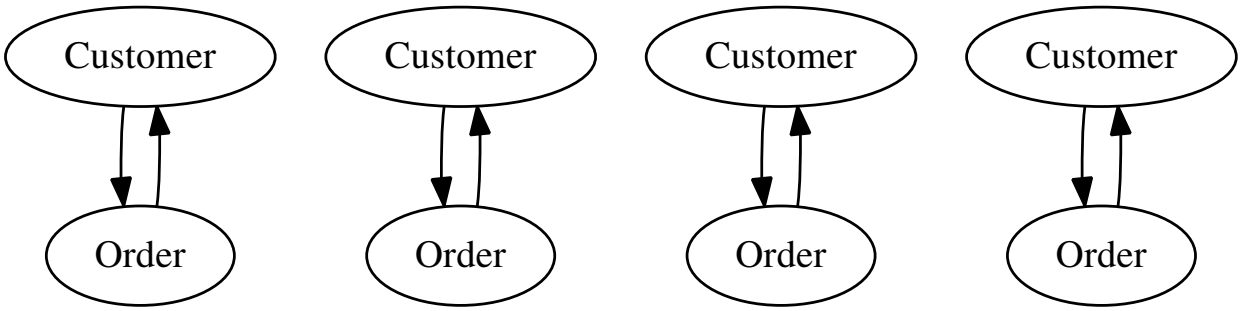

Figure 14: An example of the one-to-one mapping between "leaf" elements that we observed in SPEC JBB 2000. No matter what other objects point to these objects, they will not be summarized because the one-to-one structure guarantees that each has a unique predecessor set.

Figure 15: A Heapviz visualization of an early version of its own heap analyzer component. In the upper right, we see two large nodes of type `Value` (shown in green) that represent over 2 MB of memory usage.

Figure 16: A Heapviz visualization of the refactored version of its own heap analyzer component. The two large `Value` nodes have been replaced by several smaller nodes of types specific to their content (shown in green).