Show me the difference!

Building Hierarchical Graph Difference Visualizations for Call Graphs

Alexandru Ianta
PhD Student
Faculty of Computer Science
University of Alberta
Edmonton, Alberta, Canada
ianta@ualberta.ca

Abstract

In static analysis, abstract constructions such as call graphs and control flow graphs model the inter-procedural and intra-procedural flow of a program. Even for simple 'hello world' programs the underlying analysis constructs can become exceptionally large, boasting over 13,500 vertices and upwards of 150,000 edges. Previous work has identified and delivered on the need to visualize such constructs to aide static analysis developers in debugging their work. This work presents the visualization of differences between call graphs as a possible enhancement to previous work. The visualizations achieved took approximately 20 to 30 seconds to compute and for sufficiently small differences show the potential to provide intuitive results.

Keywords: hierarchical graphs, visualization, static analysis

1 Introduction

Static analysis developers face a different set of challenges than developers of typical applications[13]. In addition to application code, abstract constructs such as control flow graphs, that model the flow of data in an function, are present in their work[13]. This motivated the development of a set of tools specifically designed to help static analysis developers debug their analysis[13]. The result of that work was VisuFlow, *a debugging environment for static data-flow analysis*[13].

VisuFlow is implemented as a plugin for Eclipse, a popular Java integrated development environment (IDE)[9, 13].The plugin provides static analysis developers with numerous dedicated debugging tools, amongst which is a GraphView which renders the control flow graph produced by the analysis active during debugging[13].

Such visualizations were one of the key asks of the surveyed static analysis developers[13]. The authors of VisuFlow express interest in exploring better visualizations of analysis constructs. Particular emphasis was made on enhancements regarding quick response time to user modifications in either the analysis or application code[13].

This work aims to explore the visualization of differences in call graphs (a related analysis construct depicting function calls, more focused on an inter-procedural view[1] of the program [7]) that would arise from a developer making a change to the underlying code. An general approach to graph difference visualization is used so the work may easily be extended to visualizing the control flow graphs that VisuFlow renders. We assume the utility of such a visualization in the context of debugging, positing that it would provide an effective 'sanity check' for developers that allows them to ensure that the change they made to the application or analysis code, in fact reflects the change they expect in the resulting call graph.

With that assumption made, we concern ourselves primarily with assessing the performance of computing such a visualization as well as remarking on the subjective quality of the resulting output in terms of visual clutter. While the latter would have to be evaluated through more rigorous means such as a user study to bear significance, providing examples of the produced visualizations provides useful context for future researchers.

Our contributions therefore consist of:

- An implementation of a graph difference visualization technique (discussed in section 2) applied to call graphs of programs exhibiting small changes.
- A preliminary evaluation of the performance of the visualization.
- Exemplars of the visualization's best and worst results.

2 Background

Hierarchy in graph theoretic terms is a recursive grouping placed on the vertices of a graph [5]. *Metanodes* are vertices which contain subsets of nodes and edges as selected according to some criterion[5]. Hierarchical graphs are useful in presenting information where the equivalent full graph produces too much noise to be useful, or is computationally difficult to compute[5]. Archambault, Munzner, and Auber provide an overview of these concepts in great detail while presenting GrouseFlocks, a software tool for exploring graph hierarchy spaces[5].

[1] https://stackoverflow.com/questions/9889118/practical-differences-between-control-flow-graph-and-call-flow-graph

Subsequent work by Archambault focused specifically on the application of hierarchical graphs to visualize structural differences between two graphs. The advantage of the approach again stemming from the technique's ability to abstract away uninformative complexity within metanodes so that structural differences between the the graphs can be identifiable within the produced visualizations[4]. The difference map of a two graphs $G1$ & $G2$ is the union of the node and edge sets of the two graphs[4]. In the case where a unique labeling exists for $G1$ & $G2$, such that a vertex or edge with the label $L1$ is considered to be the same vertex or edge in either $G1$ or $G2$ or both, it is possible to compute the difference map in linear time with respect to the sum number of vertices and edges in $G1$ & $G2$[4]. This difference map acts as the base for the visualization, the noise to be hidden inside metanodes are connected portions of the difference map that belong only to $G1$, or only to $G2$, or are identical in both $G1$ & $G2$[4]. The remaining portions of the graph will represent vertices and edges that interface with these metanodes. All paths between vertices from the original difference map are preserved by rerouting them to the metanode containing their source or target vertices should they cross metanode boundaries[4].

3 Methodology

The aim is to evaluate the computational cost of computing a hierarchical graph difference visualization as described by Archambault[4]. Specifically we are interested in considering the application of such a visualization as part of a suite of developer tools (like VisuFlow) that can aide in the understanding of changes in a call graph. In VisuFlow, the underlying control flow graphs are recomputed when developers make changes to the code they are working on[13].

3.1 Preparing Call Graph Inputs

We assume that this re-computation is most often triggered by small changes in the underlying code. Thus, we create a series of code exemplars that mimic the kind of small code deltas we'd expect to see between sequential recomputes.

Figures 1 to 5 show the exemplars used in this work. Figures 1 to 3 build upon each other sequentially. Where as figure 4 acts as the base exemplar to compare against figure 5.

For each exemplar we run apply a SPARK analysis using Soot to produce its call graph. We then write the type, source, and target of every edge in the call graph into a CSV file. This CSV is converted to a corresponding JSON file that structures the information into a JSON object with the following format (see Listing 1). The *nodes* array is simply an array of strings containing the unique values found in the *source* and *target* fields. These JSON files will act as input for our implementation of the graph difference visualization. Node and edge labels must be unique for Archambault's technique to be applicable[4]. The node labels are simply the toString() representations of source and target Soot's *Edge* object. For edge labels we concatenate the type, source and target strings to produce a unique labeling.

Listing 1. JSON structure of call graph produced by Soot with SPARK analysis.

```
{
    "nodes": [...],
    "links": [
        {
            "type": "STATIC",
            "source": "...",
            "target": "..."
        },...
    ]
}
```

Note that despite the simplicity of these exemplars, the resulting call graphs all have over 13,500 vertices and 150,000 edges, making these non-trivial inputs. The scripts to produce the call graphs and JSON files however, are simple, their exact implementation is not relevant to our work and therefore not included as part of our artifacts, though we provide the generated JSON files we used as input. However, any pair of graphs expressed in the format described in Listing 1, that also adhere to the unique labeling constraint will work.

With our call graph inputs in hand we are ready to evaluate Archambault's technique in visualizing their differences.

```
1
2 public class HelloWorld {
3
4     public static void main(String[] args) {
5
6         System.out.println("Hello from Java 8!");
7
8     }
9
10 }
11
```

Figure 1. The hello world code exemplar.

```
1
2 public class HelloWorld {
3
4     public static void main(String[] args) {
5
6         System.out.println("Hello from Java 8!");
7         System.out.println(System.getProperty("java.version"));
8     }
9
10 }
11
```

Figure 2. The hello world code exemplar (Fig. 1) modified to also print out the 'java.version' system property.

```java
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class HelloWorld {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Hello from Java 8!");
        System.out.println(System.getProperty("java.version"));

        readFile("test.file");
    }

    public static void readFile(String fName) {
        try(FileReader fr = new FileReader(new File(fName));
                BufferedReader br = new BufferedReader(fr)){

            String line = br.readLine();
            while (line != null) {
                line = br.readLine();
            }

            br.close();

        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

```

Figure 3. The hello world with system property exemplar (Fig. 2) modified to include a call to a static method reading a file.

```java
import java.util.Random;
import java.util.stream.Stream;

public class HelloWorld {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Hello from Java 8!");
        System.out.println(System.getProperty("java.version"));

        Stream.generate(new Random()::nextInt)
            .limit(500)
            .filter(n->n>100)
            .map(n->{
                return "The random number was bigger than 100!\n";
            })
            .forEach(System.out::println);
    }


}
```

Figure 4. A base exemplar implementing a java stream that emits a string if a randomly generated number is greater than 100.

3.2 Computing the Difference Visualization

To do this, a simple javascript front end application was was built using Vue.js, BootstrapVue and, Three.js. The purpose of the application was to compute the hierarchical difference graph (hGraph) of two graphs and render the resulting output. It would also measure the amount of time it took to compute the hGraph in milliseconds and display it on the screen.

Vue.js is a popular application framework whose highly modular nature facilitates rapid prototyping [2]. BootstrapVue provides a collection of standard UI components for Vue applications to use[1]. Three.js is a javascript 3d visualization

```java
import java.util.Random;
import java.util.stream.Stream;

public class HelloWorld {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Hello from Java 8!");
        System.out.println(System.getProperty("java.version"));

        Stream.generate(new Random()::nextInt)
            .limit(500)
            .filter(n->n>100)
            .map(n->{
                StringBuilder sb = new StringBuilder();
                sb.append("The random number was bigger than 100!\n");
                sb.append("it was " + n + "!");
                return sb.toString();
            })
            .forEach(System.out::println);
    }


}
```

Figure 5. A modification of the stream exemplar (Fig. 4) that uses StringBuilder build the emitted string.

library[3]. Finally we leverage a web component developed by Vasco Asturiano called 3D Force-Directed Graph built on Three.js to render a graph provided a set of *nodes* and *links* [6].

The Vue application, along side our results, and instructions on how to reproduce them are available on GitHub [2].

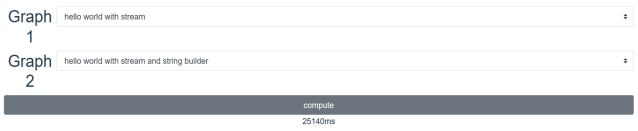

Figure 6. A screenshot of the cg-diff Vue application built for this work. The two dropdown boxes allow the user to select input call graphs for Graph 1 (G1) and Graph 2 (G2). The dark grey *compute* button underneath commences the computation of the hierarchical difference graph (hGraph). The compute time of the hGraph is displayed underneath the button in milliseconds.

The application is comprised of two components: *App* and *GraphRender*. *GraphRender* is simply responsible for initializing and displaying a 3D force-directed graph given a graph object containing nodes and links.

The *App* component is responsible for loading in the five call graphs discussed in section 3.1. It defines a function *makeHGraph(g1,g2)* which contains the auxiliary functions required to compute the hGraph. These are defined inside *makeHGraph(g1,g2)* because we want the computationally intensive work to happen in a WebWorker. WebWorkers, allow front-end applications to perform tasks in background threads that do not block the UI thread of the browser. Using web workers involves passing in a specified function to execute in a separate thread, the worker is not able to resolve

[2]https://github.com/aianta/cg-diff

references outside the passed function [11]. Thus we have the *makeHGraph(g1,g2)* wrapper function that contains all other necessary functions for hGraph computation within it.

makeHGraph(g1,g2) first computes the graph difference between *g1 & g2*, then decomposes the resulting graph by edges, and then by nodes as described by Archambault's work in sections 3.1 (Difference Map Computation) and 3.2 (Hierarchy-Based Visualization of the Difference Map)[4]. Finally, it takes the decomposed clusters of nodes and re-links them together using the edge set from the original graph difference. Edges going into or coming out of vertices now contained in a cluster (metanode) are remapped to the vertices representing the containing cluster.

The computational cost of computing the graph difference is linear with respect to the sum number of vertices and edges in *g1 & g2* [4]. The decomposition steps make use of breadth-first search(bfs) which also runs in $O(|V| + |E|)$ [8] and is applied only upon vertices and edges which have not yet been visited by previous passes of bfs. Thus the computational cost scales linearly with the size of the initial inputs and the number of decomposed clusters found in the difference graph.

4 Evaluation

To evaluate the performance of the visualization we compute and render hGraph for the pairs of *g1 & g2* that logically correspond to small changes in the underlying source code. These are displayed in rows 1 through 3 of Table 1. As an additional test on how the implementation behaves when the delta between *g1 & g2* is large, we compute the visualization where *g1* is set to the hello world exemplar (Figure 1) and *g2* is set to the base stream exemplar (Figure 4).

For each configuration we compute hGraph fives times and then calculate the mean, median and standard deviation of the recorded execution time in milliseconds. All experiments were performed on a machine equipped with an Intel i9-9900K processor @5.1GHz with 64GB of RAM. The exact sizes of the input graphs, as well as the recorded execution times are available in our artifact repository.

While a user study would be required to assess the utility of the visualization to developers. We include a screenshot of the hGraph between the hello world with system property (Figure 2) exemplar and the read file exemplar (Figure 3). It can be seen in Figure 7.

We also include a screenshot (see Figure 8) of our 'large delta' configuration (row 4 in Table 1) to illustrate how the visualization, in addition to taking significantly longer (82 seconds), becomes cluttered and likely meaningless under such conditions.

Our results in Table 1 show that for sufficiently small code changes on our machine hGraphs can be computed in between 20 and 30 seconds. This is well beyond even the most generous 10 second limit, defined by Jakob Neilson

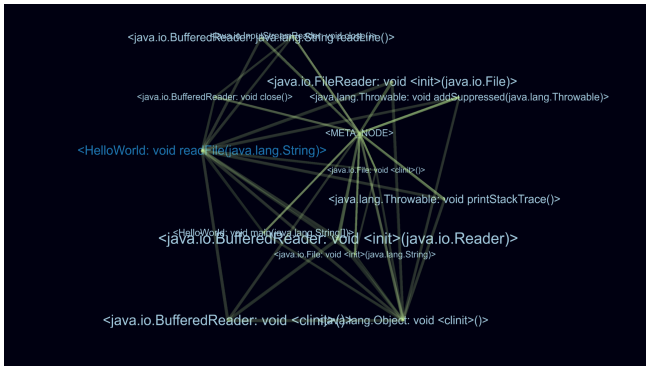

Figure 7. A screenshot from the cg-diff Vue application rendering the hGraph forthe hello world with system property (Figure 2) exemplar and the read file exemplar (Figure 3). While exhibiting some clutter, the visualized nodes clearly represent a change involving calls to FileReader, BufferedReader, IOException, and File. all of which align with our expectation from the exemplars.

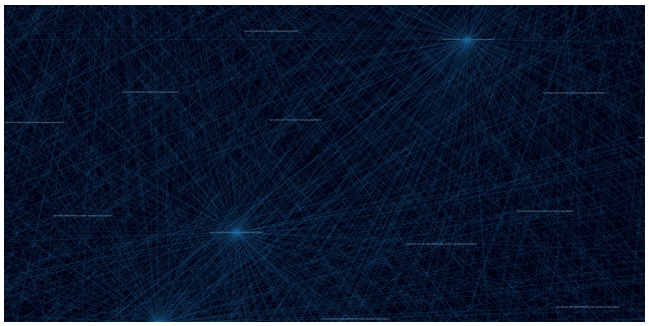

Figure 8. A screenshot from the cg-diff Vue application rendering the hGraph forthe hello world (Figure 1) exemplar and the stream exemplar (Figure 4). With the inclusion of the various underlying APIs involved in Java Streams that are missing from g1, the visualization becomes exceptionally cluttered.

in his book *Usability Engineering*, for an interval of time in which the user's attention remains focused on the task at hand [12]. Thus it seems unlikely that this visualization technique can be effectively incorporated into an interactive developer tool.

5 Related Work

As described in section 1, VisuFlow provides visualization of control flow graphs. However it does not offer any visualizations tailored to highlighting the differences between control flow graphs.

Archambault's own work introducing the technique we applied here, was never tested on static analysis constructs. His evaluation used datasets containing discussion thread history on online forums as well as scans of internet websites[4].

Table 1. Compute time to produce the hierarchical difference graph for G1 & G2. Mean and median of five executions shown, σ shows the standard deviation. ΔV shows the difference in vertices between G1 & G2. ΔE shows the difference in edges between G1 & G2. Note that execution time does not include rendering time as rendering time appeared to be nearly instant and thus not a significant factor.

G1	G2	σ	Mean	Median	Δ V	Δ E
hello world (Fig. 1)	with system property (Fig. 2)	290	21,253ms	21,395ms	0	6
with system property (Fig. 2)	read file (Fig. 3)	478	31,038ms	31,316ms	1	25
with stream (Fig. 4)	with stream and string builder (Fig. 5)	522	26,432ms	26,322ms	0	8
hello world (Fig. 1)	with stream (Fig. 4)	1,632	83,608ms	82,888ms	849	5766

Work done by Toshihirio Kamiya explores the visualization of differences in call trees between two revisions of the same software product. Their work leverages execution traces, and various heuristics to prune resulting graphs keeping them legible. The process described is semi-automatic and requires a some fine tuning by users to produce the best results. In a preliminary experiment the technique took upwards of a minute to produce results. Similar to our work, Kamiya set out to produce a proof-of-concept that will require additional effort to be applied at scale[10].

However, because our work leverages a general graph difference visualization technique we are not limited to visualizing the differences between call graphs exclusively. Control-flow graphs, exploded super graphs, as well as any other sort of graph structures which provide unique labeling for vertices and edges are compatible with the application demonstrated in this work.

To the best of our knowledge no other work attempts to apply a generic graph difference visualization technique to raw call graphs or other related program analysis graph structures, specifically aimed at visualizing small changes in the underlying source code for the purpose of assessing the viability of inclusion in interactive developer tool sets.

6 Conclusion

Providing static analysis developers with better visualizations of abstract structures such as call graphs remains a challenging task. In this work we've shown how hierarchical graph difference (hGraph) visualizations may be used to highlight changes in call graphs between code exemplars containing small changes. The technique, proposed by David Archambault, highlights differences between graphs by hiding 'uninteresting' parts of the difference map. The technique and its implementation run in linear time with respect to the provided inputs, however even so, computation still takes about 20 to 30 seconds. This disqualifies it from integration into an interactive user experience. For cases where the code difference between the two input call graphs is small, the produced visualization does, subjectively, align with expectations. A full user study could be done in future work to assess the utility of such outputs in understanding code changes.

Unfortunately, if the two input call graphs differ significantly, the resulting visualization not only takes longer to compute, but also suffers from visual cluttering that renders it, subjectively, useless. Archambault's work describes additional enhancements that could be applied such as: coarsening degree one vertices, and betweenness centrality (a measure of the number of times a vertex appears on a shortest path between other vertices[4]) coarsening which could reduce the visual clutter. But these improvements to cluttering would also increase computational cost[4].

Another optimization could aim to reduce the size of the input call graphs by increasing precision. We used SPARK analysis, but a more precise technique might yield call graphs small enough to response interactively to user input.

In any case the challenges in visually materializing the evolution of large abstract structures like call graphs in a fast, applicable manner, continue.

References

[1] [n.d.]. BootsrapVue. https://bootstrap-vue.org/

[2] [n.d.]. Introduction - vue.js. https://vuejs.org/v2/guide/

[3] [n.d.]. three.js. https://threejs.org/

[4] Daniel Archambault. 2009. Structural Differences between Two Graphs through Hierarchies. In *Proceedings of Graphics Interface 2009* (Kelowna, British Columbia, Canada) *(GI '09)*. Canadian Information Processing Society, CAN, 87–94.

[5] Daniel Archambault, Tamara Munzner, and David Auber. 2008. Grouse-Flocks: Steerable Exploration of Graph Hierarchy Space. *IEEE transactions on visualization and computer graphics* 14 (07 2008), 900–13. https://doi.org/10.1109/TVCG.2008.34

[6] Vasco Asturiano. 2021. 3D Force-Directed Graph. https://github.com/vasturiano/3d-force-graph

[7] D. Callahan, A. Carle, M.W. Hall, and K. Kennedy. 1990. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering* 16, 4 (1990), 483–487. https://doi.org/10.1109/32.54302

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[9] Eclipse Foundation. [n.d.]. IDE. https://www.eclipse.org/ide/

[10] Toshihiro Kamiya. 2018. Code difference visualization by a call tree. In *2018 IEEE 12th International Workshop on Software Clones (IWSC)*. 60–63. https://doi.org/10.1109/IWSC.2018.8327321

[11] Mozilla and individual contributors. [n.d.]. Using Web Workers. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

[12] Jakob Nielsen. 1994. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[13] Lisa Nguyen Quang Do, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden. 2018. VisuFlow: A Debugging Environment for Static Analyses. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. 89–92.