Jeremy Colebrook-Soucie, Matthew Jones, Ray Qi
Permanent Geranium Lake
21 December, 2017

Functional Spec - Heap Trace Visualization

Overview

This project is designed to interpret and present useful information on heap traces. Specifically, it should translate a difficult to interpret heap trace into a few visualizations that show useful information. This involves two stages: preprocessing data from discrete events into a more easily displayed formatting and actually displaying that data.

The main part of preprocessing will handle two components:

- **Hierarchical groupings**: It will create groupings for class-based hierarchy and data-structure hierarchy. Ideally, these groupings will form a tree, so that we can move down the tree to increase the number of groups (with the whole heap at the root and all of the individual instances of objects at the leaves).
- **Processing data as a function of time**: In order to quickly facilitate the display of information as the user selects a different time frame, the preprocessing will need to create a collection of snapshots recorded at different times. This will allow us to not recompute the heap graph each time we load a new time.

Once preprocessing is complete, the program will display the information in three different charts. The stacked area line chart will show the size of some subsection of the heap as a function of time. The sunburst chart will show both size and frequency of access of different group. The network graph will visualize the heap graph in a condensed fashion. Both the sunburst chart and network graph show a view of the program at fixed point in time.

Goals vs. Non-Goals

This project will attempt to:

- Create a hierarchy within the heap graph to condense the graph and interpretation
- Allow the user to quickly look at the size and access rate of objects/groups at any time
- Allow the user to look at how the size of objects/groups changes as a function of time.
- Make heap graphs easier to read and understand for the user

This project will not attempt to:

- Interpret the data from the heap graph, e.g. what is a high access rate or object size? What size should certain groups have relative to others?
- Determine exactly what data structures are used in the heap graph.

Structural Overview

Our project will contain a few modular components. Specifically, it will contain a trace analyzer and a visualizer. The trace analyzer reads in traces, preprocesses the data, and serves that data to the visualizer using a local HTTP server. The visualizer is a HTML5 application that displays the visualizations.

Our trace analyzer will use a combination of programming languages, possibly including (but not limited to) Python, Haskell and Scala for data analysis and C++ for high-performance graph processing algorithms, such as tree detection and graph contraction. The visualizer frontend will be written in Javascript or a JavaScript-based language (such as TypeScript or Elm) and will render the preprocessed data visualizations using the visualization library d3.js.

Modularity is the key to successful software design. Our trace analyzer is divided into a few modular components, including: a trace reader, basic analysis tools (for simple analyses as determining size and use frequency of objects), a graph analyzer (for data structure detection and graph contraction), and an HTTP server. The architecture of the visualizer will be much simpler, and will consist of a d3.js application with AJAX calls.

Contents

Network Graph:

The Network Graph will show the relationship between groups of nodes in the heap graph. The nodes in the network graph will each represent a locally condensed set of nodes in the overall heap graph, where each nodes maps to a group in the hierarchy. Nodes in the network graph will be connected by an edge if and only if there is a connection between a node in each group in the actual heap graph. Also, we will modify the edge to appear thicker or more numerous if more connections exist between nodes in the groups.

The user will be able to filter the network graph in a number of ways. First, they can control whether we are grouping by class or data-structure hierarchy, to determine data organization at their discretion. Second, the user can control the number of groups they would like to see in the graph. This takes some of the load off of the program itself to decide what the size of a visually useful network graph is.

The third way the user can filter the graph is by time. The heap graph, and therefore the network graph, is only valid for a particular time. Since the network graph is just a snapshot at one time during the lifetime of the heap trace, we will allow a sliding scale which the user can manipulate to look at a different instance in time. One way to control this will be by a play button, which will move time smoothly throughout the whole program. As the time control is moved, the graph should change in a smooth fashion to reflect the changes with time.

Stacked Area Line Chart:

The Stacked Area Line Chart will show the size of hierarchical groups as a function of time. The *x*-axis will have time in the program, and the *y*-axis will have physical size in memory. At any time, the total size of the heap is given by the top of the chart at any time, and is subdivided by color. Each color represents the total size of a specific hierarchical group, and at any time the height of its region gives its size in memory.

The user can also click on an individual group (color), and the graph will restructure to show only that group, allowing the user to only look at a single hierarchical group at a time. Optionally, the user may also choose to include or exclude each group (color) individually, to only look at a specific subset of the groups.

The user may also select from a list of filters, which will control:

- the minimum/maximum size of objects interpreted in the chart.
- The number of groups in the chart, either by number or by size

This is accomplished using NVD3's stacked area chart, which can be viewed at http://nvd3.org/examples/stackedArea.html. This gives us a way to quickly modify the user's view of the chart on a preprocessed data set. We will preprocess the data to different filter settings from the user, so that when the user alters the filter settings we can quickly jump to a new data set without delay.

Sunburst:

The Sunburst chart shows both sizes of groups and access rates of groups at any point in time. The sunburst chart is divided into angular sections. The innermost layer of the sunburst represents the highest-level groups, and the layers outside of those represent subgroups of that angular group. So, we can not only see how the groups compose the heap, in terms of size, but also how large each subgroup is as well.

If the user clicks on an angular group, that group is then treated as the whole graph. It takes up the center of the chart, and all of its subgroups are scaled accordingly. Clicking on the center of the chart instead moves to the next highest group, essentially zooming out towards the whole graph. The user can use these controls to move throughout different groups/levels of the entire heap graph. The user can also hover over any angular group at any time to find basic information (name and size) about that group.

There is also a button which the user may select to view access rates. If the user does not select this button, the angular groups are colored by their highest-level group such that the colors of each such group are unique. If the user selects this button, the colors are changed to reflect relative access rates. Light blue will be used to represent the lowest access rates (about 0), and red will be used to represent the highest access rates.

Below the chart, a slider will be used to control time. Since the chart only holds information for one instance in time, but the user should be able to view the information for any time during the run of the program, the user can move along the slider and the chart will smoothly transition as time goes on. Or, the user can click at any point on the slider, and the chart will jump to that time. Alternatively, the slider will have a play button, which will move the slider automatically to simulate the program run.

This is accomplished using D3's sunburst chart, which is exemplified at https://bl.ocks.org/kerryrodden/7090426. Modifying this gives us smooth transitions between layers, and allow us to quickly recolor the whole chart in order to show the access rates.

Open Issues

There are currently a few open issues, largely due to the fact that this project is still in early stages of design:

- How much should we hierarchically condense data on our own? This seems to vary depending on the output we are choosing:
    - In the case of the sunburst chart, it helps to condense as much as possible. It is fine for us to show the innermost layer as only a few hierarchical groups, because details will become more apparent as we move towards the outer layers.
    - In the case of the stacked area line chart, we would like to condense a fair amount so that the user is looking at a few groups, not tens or even hundreds of groups. A possible solution here would to pick a reasonable goal for the number of groups (e.g. 10-15), and let the user modify that number in some way to reach a reasonable number of groups at their discretion.
    - In the case of the network graph, this is where the user should have the most control over the number of groups. The user may want to see how a few large groups interact with each other, or they may want to see how many smaller groups interact with each other to get a bigger picture of the heap structure.
- As we condense data, what do we call the new hierarchical groups? This is trivial for class hierarchies, since we can just use the type of the object, or the class name, but for data structure hierarchies this is not so simple.
- What existing tools will we use to build the network graph?
- Since we do not have a continuous set of functions for size, but rather jump discontinuities at allocations and deallocations, we do not need to store time as a

continuous scale. If we do this, how do we move from the discrete scale to a continuous scale the user can slide through?

- A large set of technical issues:
    - How do we decide on a non-class (data structure-based) hierarchy? Specifically, how do we try to identify these in a graph in a way that is not NP-hard?
    - How should we store the processed input data, so that the amount of information is not overwhelming, but can also be called upon quickly to be displayed to the user? Specifically, what can we afford (in space) to store, and what can we afford (in time) to compute on the spot?
    - How should access rate be computed. A big issue that can only be reasoned through by some sort of testing.