Assignment 6: Treemaps

Date Posted: April 6, 2024

Due Date: April 19, 2024 11:59pm

Description

In this assignment, you will implement a three variations of a treemap algorithm.

Submission Requirements

Provided Template

In the GitHub repository, you will find six files: `index.html`, `plot.js`, `flare.js`, `d3.v7.min.js`, `README.md`, and `spec.pdf`. The `index.html` provides a minimal HTML outline to set up your webpage and structure your responses. The `plot.js` file contains skeleton code for your JavaScript code and is where you will write your plotting code. You must build on the existing skeleton code. The `flare.js` file contains the tree that you will visualize. This is the same tree used in many of the examples from tree lecture. `d3.v7.min.js` is a copy of the d3 library. You will need to load this before any other scripts that use d3. `README.md` provides a template to use for documenting your repository as described below. Finally, `spec.pdf` contains a copy of the assignment specification (this document).

Submitted Files

You will create your visualizations in `index.html`, it should be the only HTML file in your submission. You should write your main plotting code in `plot.js`, but may add any additional JavaScript and CSS files as you see fit. You must add appropriate documentation via commenting to all JavaScript code. Finally, you should include a `README.md` file (using the given template) that provides a text description of what is in your repository, how to run your program and any parameters that you used. Also, document any idiosyncrasies, behaviors, or bugs of note that you want us to be aware of. **When updating the README, please remove/update the instructive text in parentheses - that is only there to guide you.** When you have completed your assignment, please submit the link to your GitHub repository on Canvas.

Implementing Treemaps in D3 (100pts)

Using the provided template, you will implement a basic treemap algorithm along with two variations on that algorithm. You will use the Flare library dataset that we saw in class, which describes the hierarchy of the file structure for the flare software library. This is stored in the `flare.js` file.

Note, d3 offers native support for treemap visualizations. However, the goal of this assignment is for you to actually write the main part of this algorithm and its variations.As such, you are **not allowed** to use the `d3.layout.treemap` nor are you allowed to existing code that computes treemaps. You must use build on the code provided.

While your visualizations will ultimately graded using the `flare.js` tree, I encourage you to create a smaller test tree, where you can easily manually create the treemap, to test your alogorithms.

Step 1: Implement the Basic Treemapping Algorithm

You will start by implementing a basic treemap. Treemapping algorithms draw a tree by progressively splitting a rectangle along vertical and horizontal lines such that the area of the rectangle is proportional to the specified value (see here for the original paper). The most basic version of this algorithm alternates the direction it splits the rectangles at each level (depth). You will start by splitting the first rectangle along the x-direction, because our computer screens are wider than they are tall. Then, the second level splits will be along the y-direction, the third level splits will be along the x-direction, and so on.

To implement this variant, in the skeleton code you will need to finish implementing three components. First, you will implement the function `setTreeDepth()` which should add the attribute `depth` to each node of the tree, as well as return the maximum depth in the tree. This function will look similar to the `setTreeSize()` and `setTreeCount()` functions, both of which add attributes to the tree. You will use the depth of a node to decide if it is a horizontal or vertical split.

Then you will need to finish the implementation of `setRectangles()`. This function sets the attribute `rect` for each node in the tree. This attribute stores the coordinates of the upper left corner of the rectangle (x1,y1) and the lower right corner (x2,y2). Note, that `setRectangles()` is coded generically to compute the area of the rectangles using either the count (the number of leaves that stem from a node) or the size (the cumulative size of all leaves that stem from a node), via the accessor `attrFun()`.

Finally, you will finish implementing the function `setAttrs()`, which sets the attributes for all of the `rect` svg elements, using the selection of rects bound to the nodes of the tree.

Step 2: Improving the Visualization

After implementing the basic algorithm, you will add the following three improvements:

- Add margins to each rectangle to better depict the nested hierarchy. The margins should not be added between rectangles at the same level, only around the edge of the bounding (parent) rectangle. There is a variable `border` in `setRectangles()` that you may want to use.
- Color the nodes to encode their depth in the tree. This also makes it easier to see the hierarchy.
- Draw the nodes in each level in descending order of value (and thus area). This makes it easier to make comparisons between nodes in the same level.

Step 3: Cutting Each Layer in the "Best" Direction

The basic algorithm chooses a fixed direction to split the nodes for each depth. However, this causes relatively uneven shapes and makes it harder to read areas (e.g., it creates very narrow and tall vertical slices). A natural improvement is to instead have the algorithm split along the longest axis for each individual level. If, for a given level, the parent rectangle is wider than it is tall, we will split along the x-axis. If the parent rectangle is taller than it is wide, we will split along the y-axis.

You should implement this approach by modifying the `setRectangles()` function. However, `setRectangles()` should also still support the basic algorithm, so you will need track which version of the algorithm should be used (e.g., via a parameter). You will add two new buttons to `index.html` to transition the rectangles to this new layout, using the size attribute and the count attribute (similar to the existing buttons). You should now have four buttons that switch between the two attributes for each of the two algorithms.

Step 4: Adaptively Cutting in the "Best" Direction for Each Node

Both of the previous algorithms cut each layer in a single direction, which may still not most efficiently use the space. A natural next improvement to try is to determine the "best" direction for each individual node in a layer, rather than the single "best" direction for the whole layer. For each node in a layer, you will need to check which direction has the most remaining space and split it in that direction. This will require tracking some extra variables, such as the remaining width and height in the current rectangle and the current minimum x and y values (i.e., the smallest x and y values that we could draw at without overlapping an existing rectangle).

You can implement this approach by either modifying the `setRectangles()` function again or adding in a new, similar function. Again, you will add two new buttons to `index.html` to transition the rectangles to this new layout, using the size attribute and the count attribute (similar to the existing buttons). You will now have six buttons that switch between the two attributes for each of the three algorithms.

Grading

Your submission will be graded on how well it fulfills the specified criteria. Additionally, points will be deducted for lack of documentation (i.e. commenting your JS code and filling out the README.md file). The below rubric describes the point values for individual components.

Correctly computing the tree node depth (Step 1)	10
Correctly computing the rects in setRectangle() in (Step 1)	20
Correctly setting the the svg rect attributes in setAttr() (Step 1)	10
Correctly adding margins between the parent rectangle and the children rectangles (Step 2)	10
Correctly setting the color based on depth (Step 2)	10
Correctly drawing the nodes in descending order by value (Step 2)	5
Correctly implementing the best direction cutting approach (Step 3)	15
Correctly implementing the adaptive best direction cutting approach (Step 4)	15
Proper documentation (code comments and README.md)	5