Data visualization

COSC 480B

Reyan Ahmed

rahmed1@colgate.edu

Lecture 10

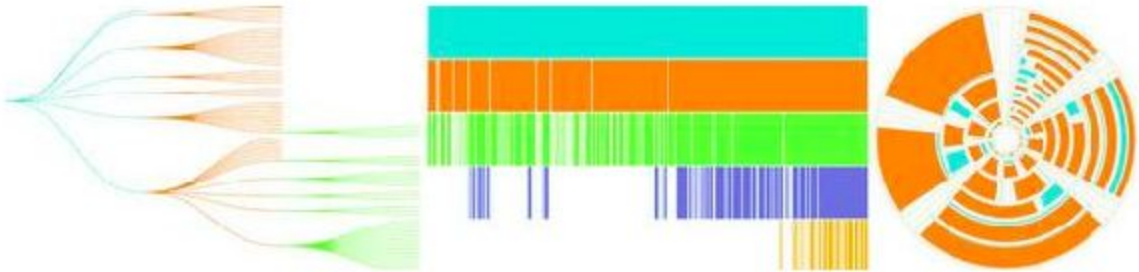
Hierarchical visualization

Overview

- Understanding hierarchical data principles
- Using dendrograms
- Learning about circle packs
- Working with treemaps
- Employing partitions

Overview

Some of the hierarchical diagrams we'll look at in this chapter. The dendrogram (left), the icicle chart (middle), and a treemap (right) showing a radial projection that's popular with hierarchical diagrams.

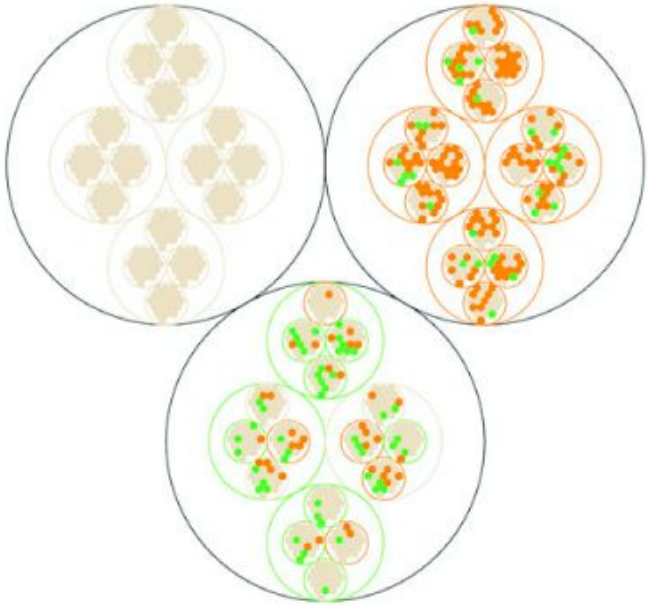


Hierarchical patterns

Typical A/B testing results in a tabular view, showing several metrics and the change versus the control cell. Positive changes are denoted with a plus symbol, and statistically significant changes are shown with green for a statistically significant positive change and orange for a statistically significant negative change.

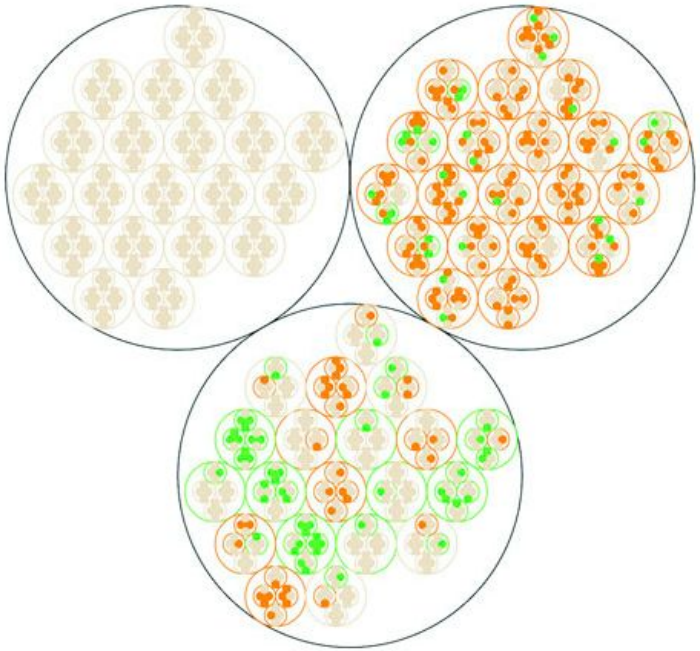
Hierarchical patterns

A circle pack of A/B testing results, showing the nested results by cell, metric, subscription level, and country. Green shows statistically significant wins, and orange shows statistically significant losses.



Hierarchical patterns

Hierarchical viz of A/B testing results ordered by cell, country, subscription, and then metric



Working with hierarchical data

Consider the tweeter dataset:

```
var root = {id: "All Tweets", children: [
{id: "Al's Tweets", children: [{id: "tweet1"}, {id: "tweet2"}]},
{id: "Roy's Tweets", children: [{id: "tweet1"}, {id: "tweet2"}]}
...
```

To construct the hierarchical structure use the following code:

```
var root = d3.hierarchy(root, function (d) {return d.children})
```

Working with hierarchical data

Hierarchical JSON and hierarchical objects:

- There is a root node/object
- The root does not have a parent, or the parent is null
- And the root has children
- The other object looks similar to the root
- They have parent
- Also there are leaf nodes
- Leaf nodes don’t have children

Working with hierarchical data

Let’s say our dataset is the following:

```
{
   cell: "gifs everywhere",
   country:"Norway",
   metric:"recommendations",
   significance:0.07408233813305311,
   subscription:"deluxe",
   value:0.4472930460902817
}
```

Working with hierarchical data

Then to create a nested hierarchical javascript object we run the following:

```
var nestedDataset = d3.nest()
  .key(d => d.cell)
  .key(d => d.metric)
  .key(d => d.subscription)
  .key(d => d.country)
  .entries(dataset)
```

Working with hierarchical data

For tabular format the procedure is different. Consider the following tabular data:

```
child, parent
you, your mom
you, your dad
your dad, your paternal grandfather
your dad, your maternal grandmother
your mom, your maternal grandfather
your mom, your maternal grandmother
, you
```

Working with hierarchical data

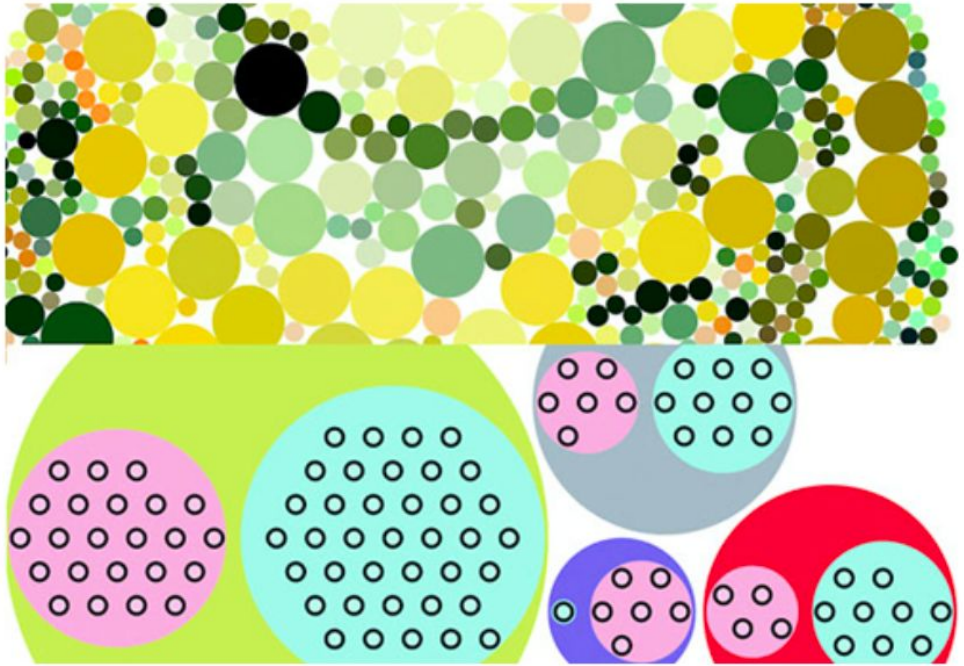
To create the hierarchical object we run the following:

```
d3.stratify()
  .parentID(d => d.child)
  .id(d => d.parent)
```

Note that we assigned child as parent, because the data is in reverse order.

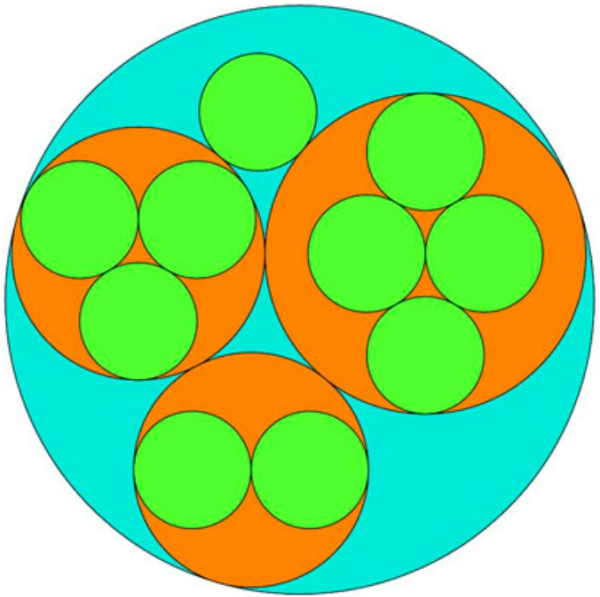
Pack layouts

Pack layouts are useful for representing nested data. They can be flattened (top) or they can visually represent hierarchy (bottom).



Pack layouts

Each tweet is represented by a green circle nested inside an orange circle that represents the user who made the tweet. One of those green circles is exactly the same size as its parent orange circle, which we address below. The users are all nested inside a blue circle that represents our “root” node.



Pack layouts

Circle packing of nested tweets data

```
d3.json("tweets.json", viz)

function viz(data) {
var depthScale = d3.scaleOrdinal()
  .range(["#5EAFC6", "#FE9922", "#93c464", "#75739F"])

var nestedTweets = d3.nest()
  .key(d => d.user)
  .entries(data.tweets);
var packableTweets = {id: "All Tweets", values: nestedTweets};        1
var packChart = d3.pack();                                            2

packChart.size([500,500])                                             3
```

1 Puts the array that d3.nest creates inside a “root” object that acts as the top-level parent
2 Initialize the pack layout
3 Sets the size of the circle-packing chart

Pack layouts

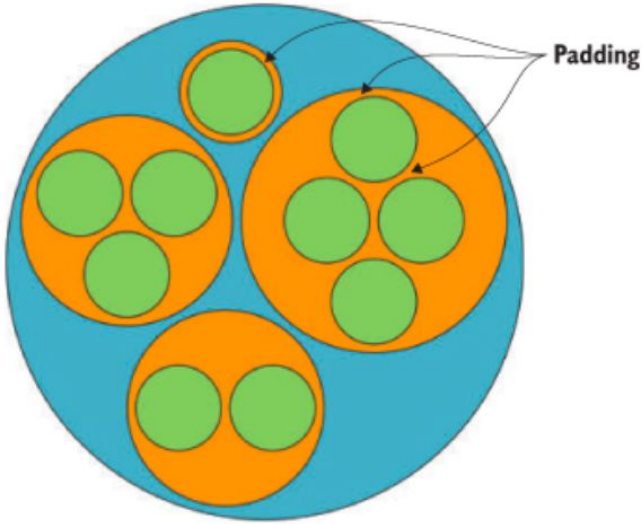
```
var root = d3.hierarchy(packableTweets, d => d.values)            4
  .sum(() => 1)                                                   5

d3.select("svg")
  .append("g")
  .attr("transform", "translate(100,20)")
    selectAll("circle")
    .data(packChart(root).descendants())                          6
    .enter()
    .append("circle")
    .attr("r", d => d.r)                                          7
    .attr("cx", d => d.x)
    .attr("cy", d => d.y)
    .style("fill", d => depthScale(d.depth))                      8
    .style("stroke", "black")
}
```

4 Process the hierarchy with an accessor function for child elements to look for values, which matches the data created by d3.nest
5 Creates a function that returns 1 when determining the size of leaf nodes
6 Processes the hierarchy with the pack layout and then flattens it using descendants to get a flattened array
7 Radius and xy coordinates are all computed by the pack layout
8 The pack layout also gives each node a depth attribute that we can use to color them distinctly by depth

Pack layouts

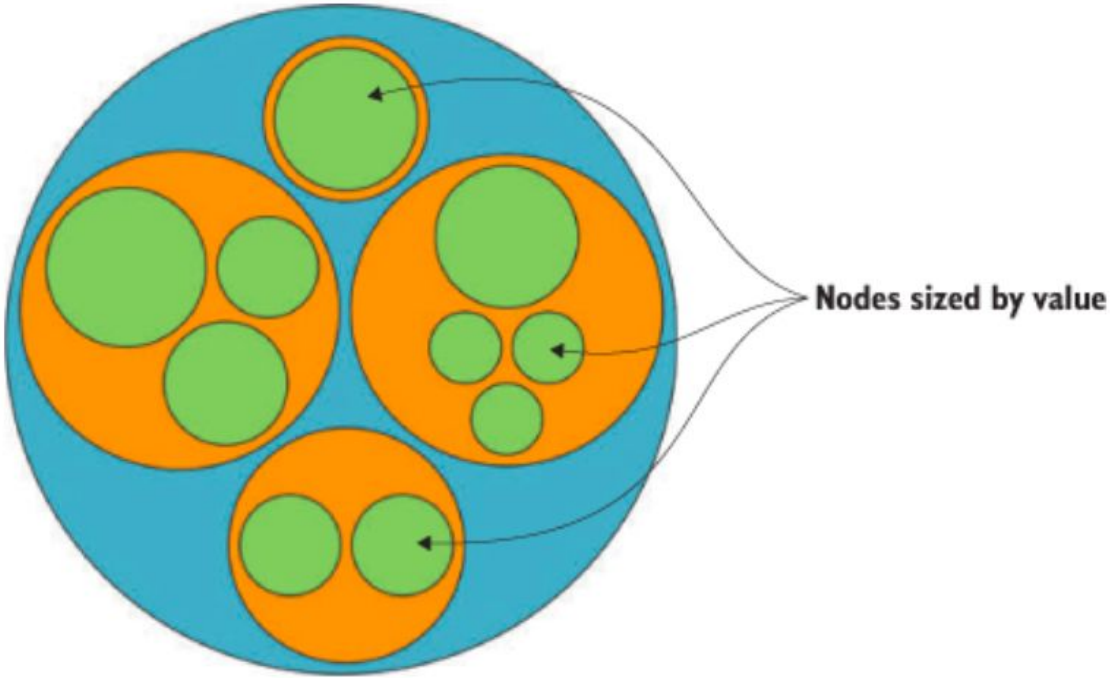
```
packChart.padding(10)
```

An example of a fixed margin based on hierarchical depth. We can create this by reducing the circle size of each node based on its computed depth value.



Pack layouts

A circle-packing layout with the size of the leaf nodes set to the impact factor of those nodes



Pack layouts

```
d3.hierarchy(packableTweets, d => d.values)
.sum(d => d.retweets ? d.retweets.length +
d.favorites.length + 1 : undefined)                1
```

1 Adds 1 so that tweets with no retweets or favorites still have a value greater than zero and are displayed along with checking to make sure it has a retweets property

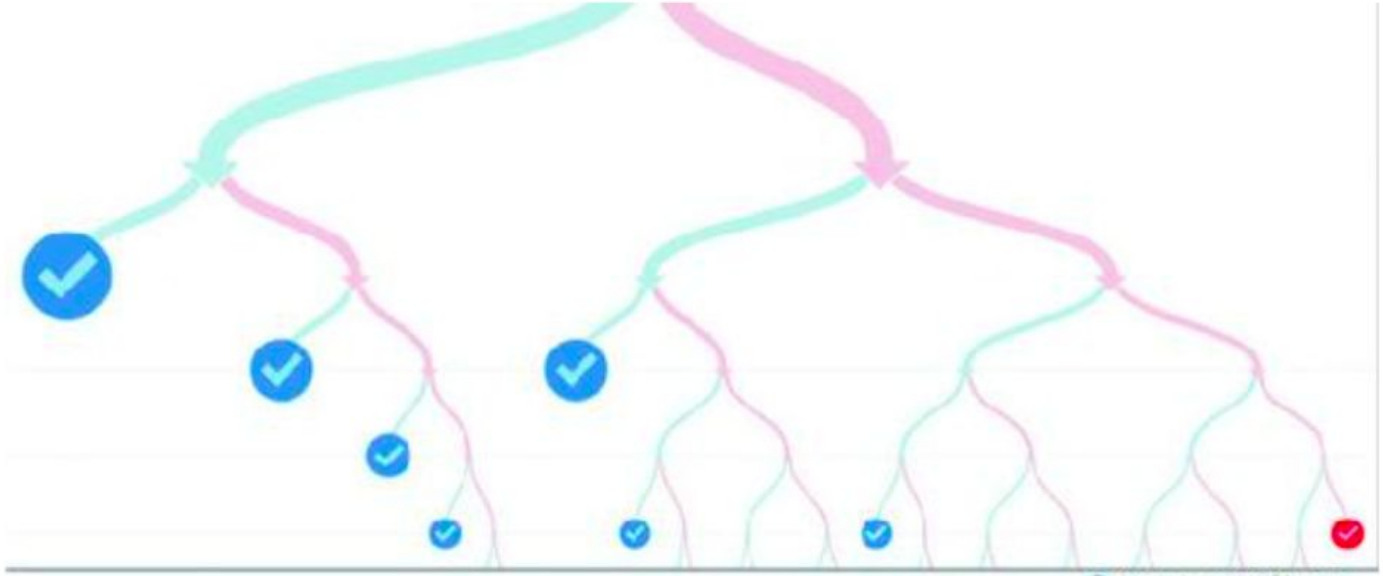
Pack layouts

When to use circle packing

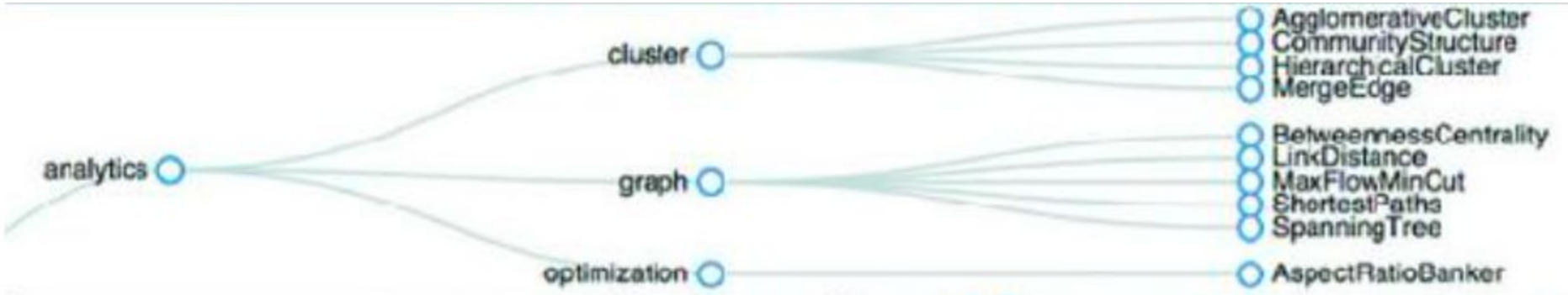
- Circle packs don’t use space efficiently
- Tree maps based on rectangles uses spaces more efficiently
- Circles are not good in presenting magnitudes
- Hence, it is more reasonable to keep circles same size
- If you really need to represent sizes, use other visualization

Trees

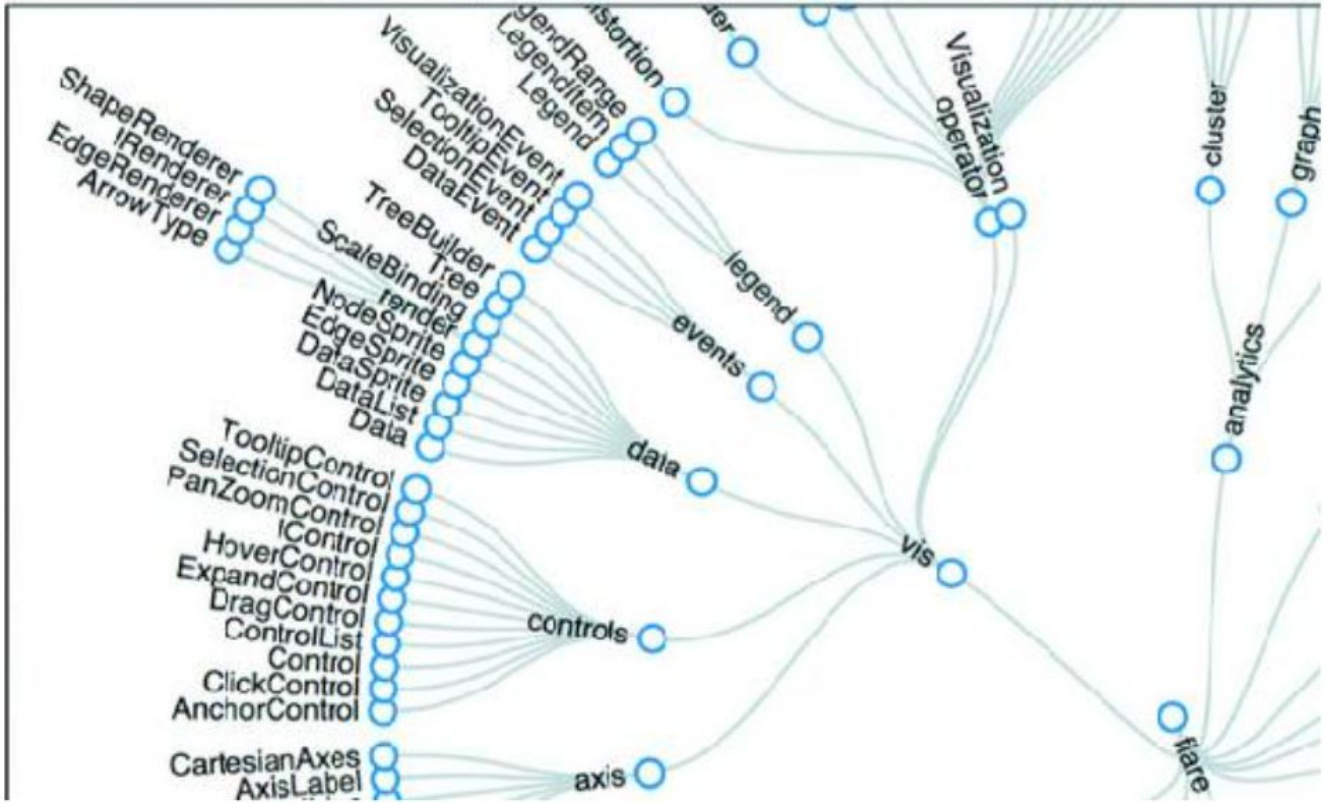
Tree layouts are another useful method for expressing hierarchical relationships and are often laid out vertically, horizontally, or radially. (Examples from Mike Bostock at d3js.org.)



Trees


analytics
cluster
graph
optimization
AgglomerativeCluster
CommunityStructure
HierarchicalCluster
MergeEdge
BetweennessCentrality
LinkDistance
MaxFlowMinCut
ShortestPaths
SpanningTree
AspectRatioBanker

Trees


ShapeRenderer
IRenderer
EdgeRenderer
ArrowType
VisualizationEvent
TooltipEvent
SelectionEvent
DataEvent
LegendRange
LegendItem
Legend
TreeBuilder
Tree
ScaleBinding
render
NodeSprite
EdgeSprite
DataSprite
DataList
Data
legend
events
data
TooltipControl
SelectionControl
PanZoomControl
IControl
HoverControl
ExpandControl
DragControl
ControlList
Control
ClickControl
AnchorControl
controls
CartesianAxes
AxisLabel
axis
vis
Visualization
operator
cluster
graph
analytics
flare

Trees

Callback function to draw a dendrogram

```
var treeChart = d3.tree();
treeChart.size([500,500])

  var treeData = treeChart(root).descendants()

d3.select("svg")
  .append("g")
  .attr("id", "treeG")
  .attr("transform", "translate(60,20)")
  .selectAll("g")
  .data(treeData)
  .enter()
  .append("g")
  .attr("class", "node")
  .attr("transform", d => `translate(${d.x},${d.y})`)          1
```

1 Draw a <g> element for each node so we can put a circle in it now and add a label later

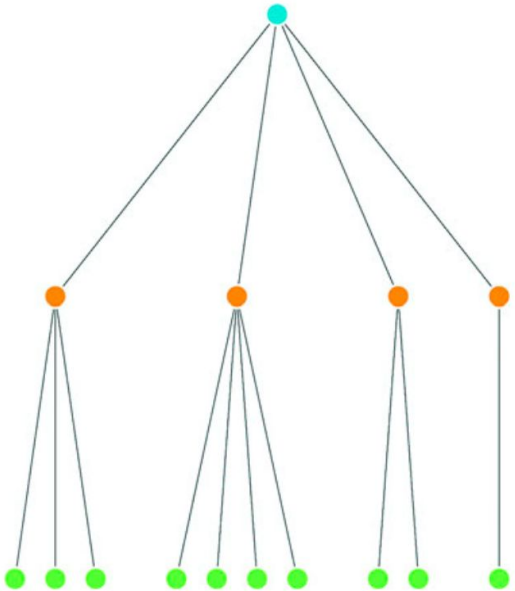
Trees

```
d3.selectAll("g.node")
  .append("circle")
  .attr("r", 10)
  .style("fill", d => depthScale(d.depth))                    2
  .style("stroke", "white")
  .style("stroke-width", "2px");                          3

d3.select("#treeG").selectAll("line")
  .data(treeData.filter(d => d.parent))                       4
  .enter().insert("line","g")
  .attr("x1", d => d.parent.x)                          5
  .attr("y1", d => d.parent.y)
  .attr("x2", d => d.x)                               6
  .attr("y2", d => d.y)
  .style("stroke", "black")
```

2 Fill based on the depth calculated by d3.hierarchy
3 Add a white halo around each node to give the connecting lines an offset appearance
4 Draw links using the same data except filter out any nodes that don’t have parents (which won’t have links)
5 Draw the link ending at the child node location
6 Draw the link starting at the parent node location

Trees

A dendrogram laid out vertically using data from tweets.json. The level 0 “root” node (which we created to contain the users) is in blue, the level 1 nodes (which represent users) are in orange, and the level 2 “leaf” nodes (which represent tweets) are in green.



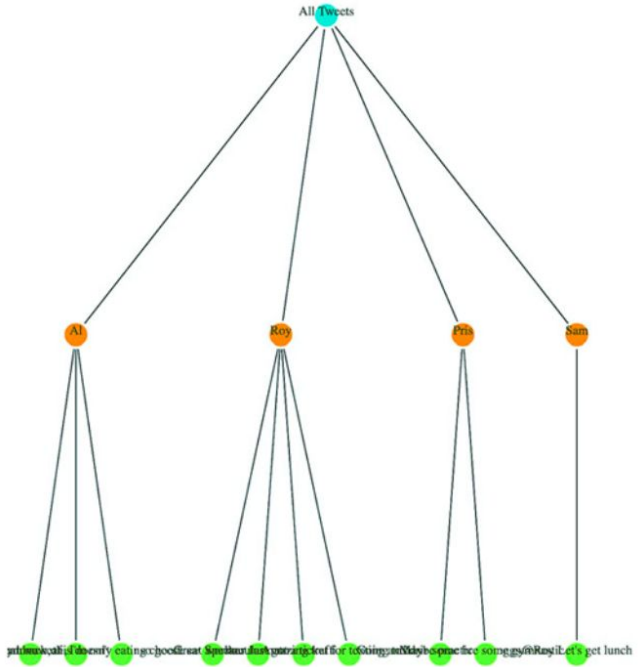
Trees

To add labels:

```
d3.selectAll("g.node")
        .append("text")
        .style("text-anchor", "middle")
        .style("fill", "#4f442b")
        .text(d => d.data.id || d.data.key || d.data.content)
```

Trees

A dendrogram with labels for each of the nodes



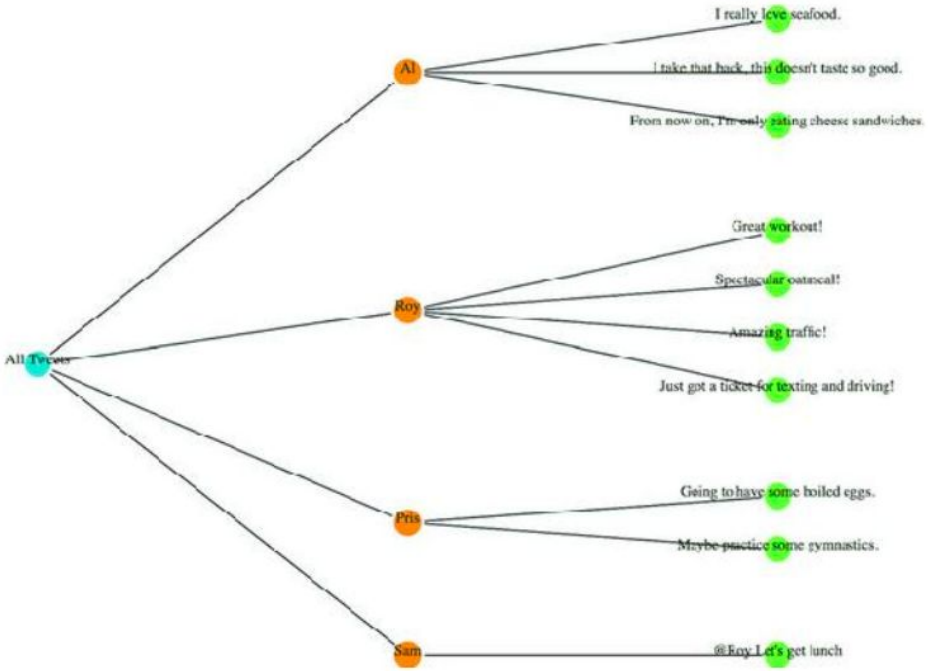
Trees

Let’s rotate the drawing by 90 degrees:

```
    .append("g")
...
        .append("g")
        .attr("class", "node")
        .attr("transform", d => `translate(${d.y},${d.x})`)
...
        .enter().insert("line","g")
        .attr("x1", d => d.parent.y)
        .attr("y1", d => d.parent.x)
        .attr("x2", d => d.y)
        .attr("y2", d => d.x)
```

Trees

The same dendrogram but laid out horizontally



Trees

Let’s add some interaction so that we can shift the drawing:

```
treeZoom = d3.zoom()                                          1
treeZoom.on("zoom", zoomed)                                     2
d3.select("svg").call(treeZoom)                                 3
function zoomed() {
d3.select("#treeG").attr("transform",
`translate(${d3.event.transform.x},${d3.event.transform.y})`)     4
}
```

1 Creates a new zoom component
2 Keys the “zoom” event to the zoomed() function
3 Calls our zoom component with the SVG canvas
4 Updating the <g> to set it to the same translate setting of the zoom component updates the position of the <g> and all its child elements

Trees

Radial tree diagrams

- Draws the tree in a circular fashion
- The size is the radius of the circle
- So we want to set the size equal to half of the display

```
treeChart.size([200,200])
```

Trees

we need to create a projection function to translate the xy coordinates into a radial coordinate system:

```
function project(x, y) {
   var angle = x / 90 * Math.PI
   var radius = y
   return [radius * Math.cos(angle), radius * Math.sin(angle)];
}
```

Trees

Then we use that function to calculate new coordinates for our nodes and lines:

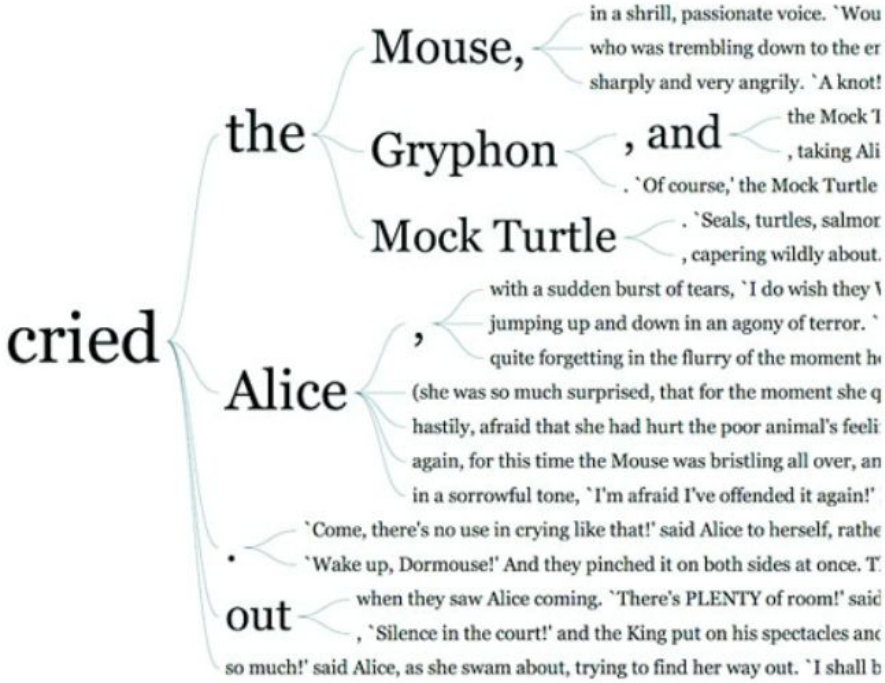
```
   .append("g")
   .attr("id", "treeG")
   .attr("transform", "translate(250,250)")
   .selectAll("g")
...
   .append("g")
   .attr("class", "node")
   .attr("transform", d => `translate(${project(d.x, d.y)})`)
...
   .enter().insert("line","g")
   .attr("x1", d => project(d.parent.x, d.parent.y)[0])
   .attr("y1", d => project(d.parent.x, d.parent.y)[1])
   .attr("x2", d => project(d.x, d.y)[0])
   .attr("y2", d => project(d.x, d.y)[1])
```

Trees

The same dendrogram laid out in a radial manner.

Trees

Example of using a dendrogram in a word tree by Jason Davies (www.jasondavies.com/wordtree/).



Trees

The same dataset rendered using d3.tree (left) and d3.cluster (right)

Trees

- It’s better to use tree instead of cluster
- Because cluster uses more ink than tree
- If all leaves needs to be shown in the same level, use it
- Overall trees are more suitable when the nodes are same type
- If nodes are different type, use circle pack

Partition

Drawing a simple partition layout

```
var root = d3.hierarchy(packableTweets, d =>
d.values)
   .sum(d => d.retweets ? d.retweets.length +
d.favorites.length + 1 :
   undefined)

var partitionLayout = d3.partition()
   .size([500,300])

partitionLayout(root)                    1
```

1 The code is all much the same as our earlier hierarchical layouts

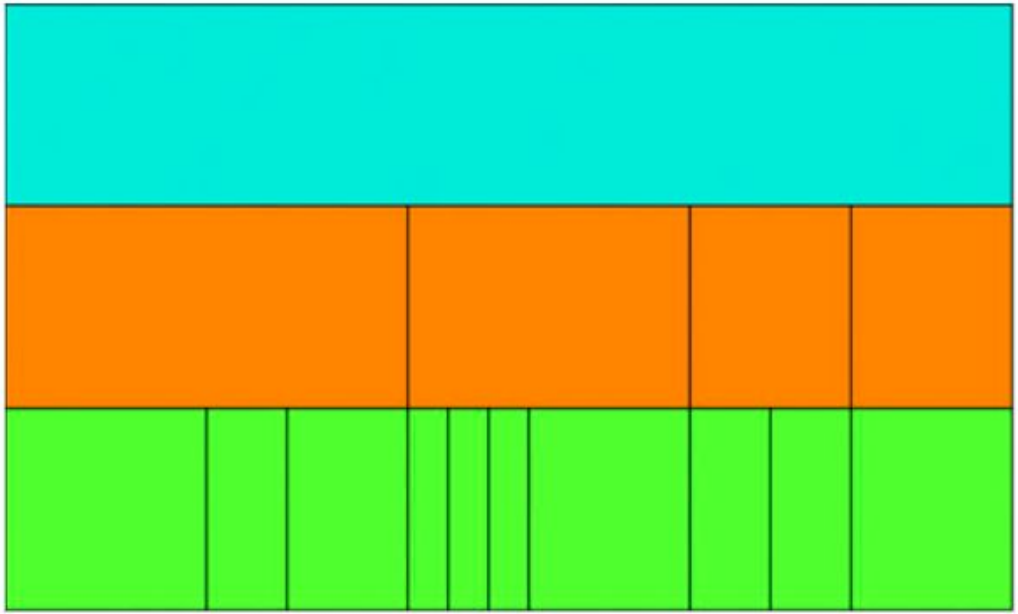
Partition

```
d3.select("svg")
  .selectAll("rect")
  .data(root.descendants())
  .enter()
  .append("rect")
  .attr("x", d => d.x0)                    2
  .attr("y", d => d.y0)                    2
  .attr("width", d => d.x1 - d.x0)           3
  .attr("height", d => d.y1 - d.y0)          3
  .style("fill", d => depthScale(d.depth))
  .style("stroke", "black")
```

2 Position is given back as a bounding box where the upper left corner in x0/y0
3 Size can be derived by subtracting the bottom right corner (x1/y1) from the upper left

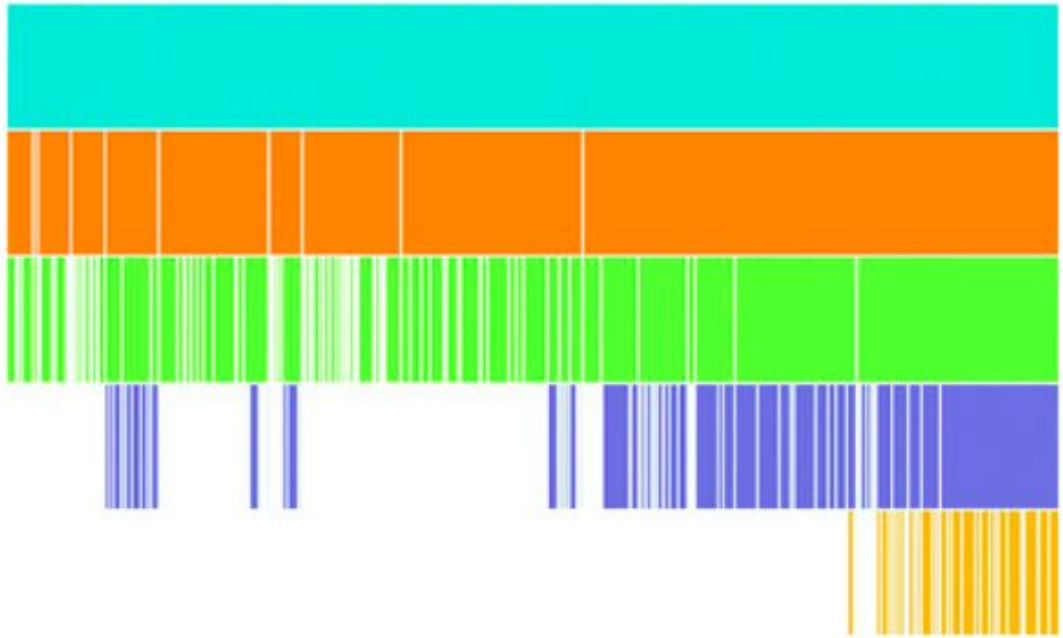
Partition

A partition layout of our data, showing tweets at the bottom in green, sized by “impact” with users in orange sized by the total impact of their tweets and the root node (in this case “All Tweets”) in blue.



Partition

Icicle charts look like melting icicles hanging from the gutter when you have hierarchical data of uneven depth, as you have in this example.



Partition

Sunburst: radial icicle chart. The most popular blocks on October 20, 2016, which include not one or two but four different sunburst diagrams

Partition

Using the partition layout to create a sunburst

```
var partitionLayout = d3.partition()
   .size([2 * Math.PI,250])                    1

partitionLayout(root)

var arc = d3.arc()
   .innerRadius(d => d.y0)
   .outerRadius(d => d.y1)                      2
```

1 Set the size of the layout to be 2PI for with width and whatever radius we want of the total chart for the height
2 We'll use y0 and y1 to determine the upper and lower bounds of the arcs we draw so that they stack on top of each other radially

Partition

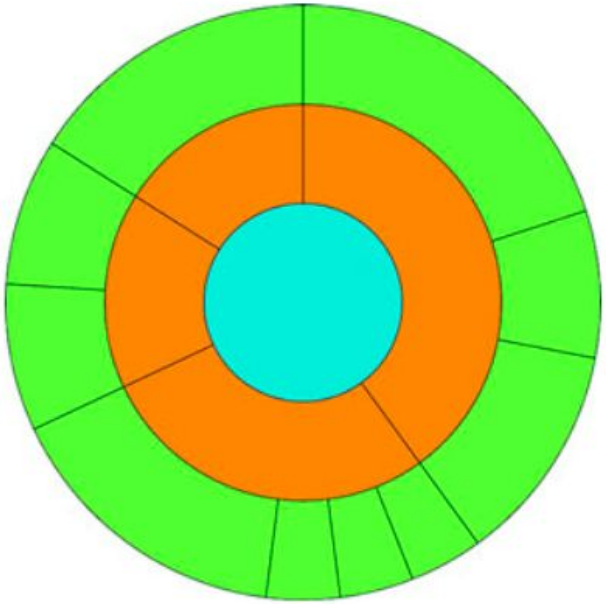
```
d3.select("svg")
  .append("g")
  .attr("transform", "translate(255,255)")            3
  .selectAll("path")
  .data(root.descendants())
  .enter()
  .append("path")
  .attr("d", ({ y0, y1, x0, x1 }) => arc({y0, y1,
       startAngle: x0, endAngle: x1}))                4
  .style("fill", d => depthScale(d.depth))
  .style("stroke", "black")
```

3 Remember we need to re-center the chart because the arc generator will draw out from 0,0

4 Create a properly formatted object of the kind that arc() expects using object destructuring and object literal shorthand

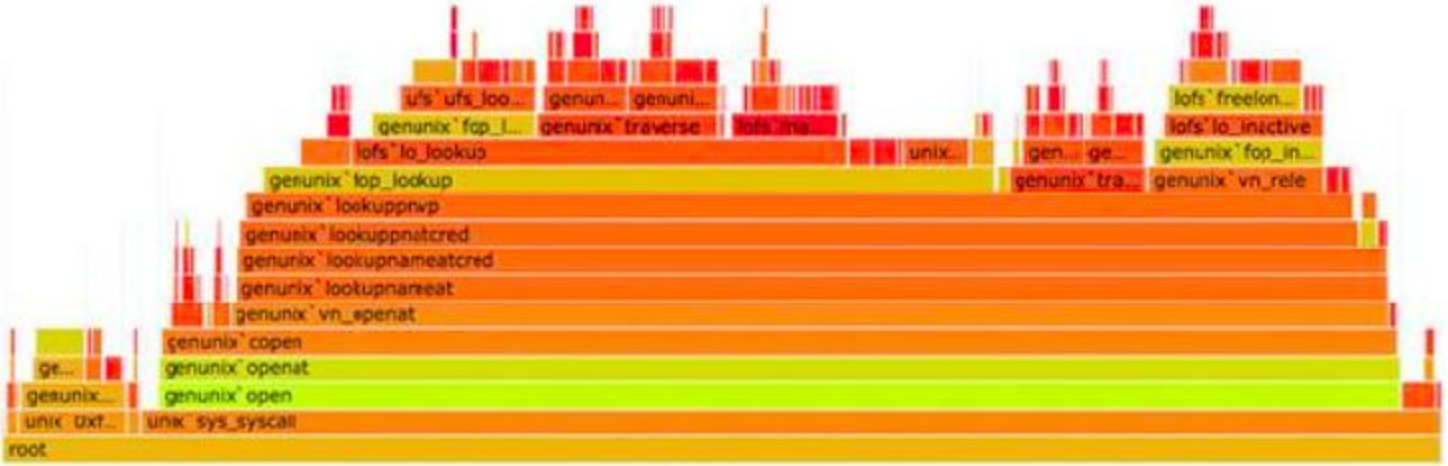
Partition

A sunburst version of our nested tweets.



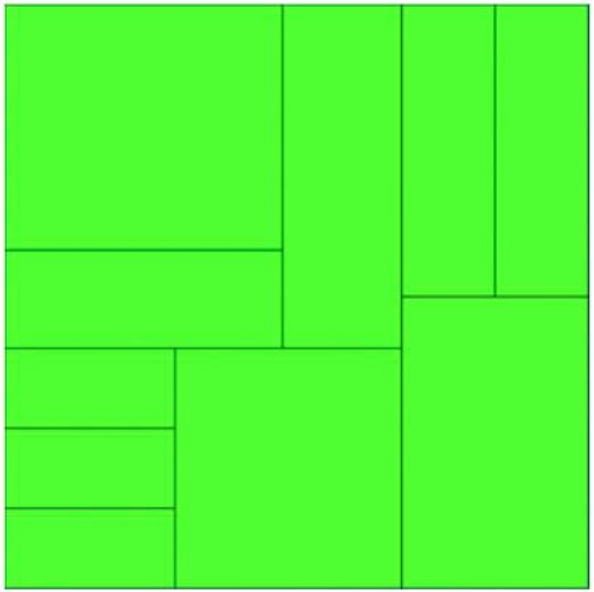
Partition

An example of d3-flame-graph, which implements the flame graph first developed by Brandon Gregg. Note that the value of the children (in this case the higher bars) often adds up to less than the value of the parents.



Treemaps

- Another way to show hierarchical data
- A mix of circle packing and partition
- Uses rectangles to present nodes
- Encloses rectangles with parent rectangles



Treemaps

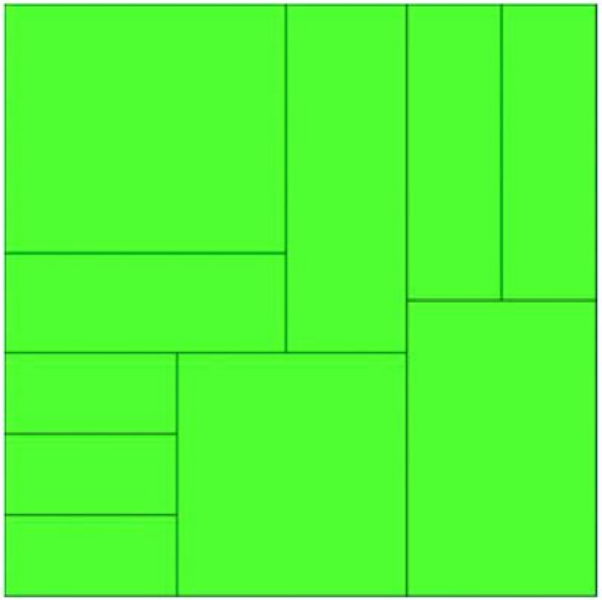
Drawing a treemap

```
var treemapLayout = d3.treemap()
   .size([500,500])

treemapLayout(root)                                          1

d3.select("svg")
  .selectAll("rect")
  .data(root.descendants(),
     d => d.data.content || d.data.user || d.data.key)       2
  .enter()
   append("rect")
  .attr("x", d => d.x0)
  .attr("y", d => d.y0)
  .attr("width", d => d.x1 - d.x0)
  .attr("height", d => d.y1 - d.y0)                          3
  .style("fill", d => depthScale(d.depth))
  .style("stroke", "black")
```

1 Because the layout mutates root, you can run it without assigning it to a variable
2 Set a key so we can filter-zoom later
3 All the hierarchical layouts that are typically represented with rectangles expose x1,x0 and y1,y0 because it's easy to derive height/width from that and still use it for other projections

Treemaps

A treemap without padding will only show the leaf nodes.



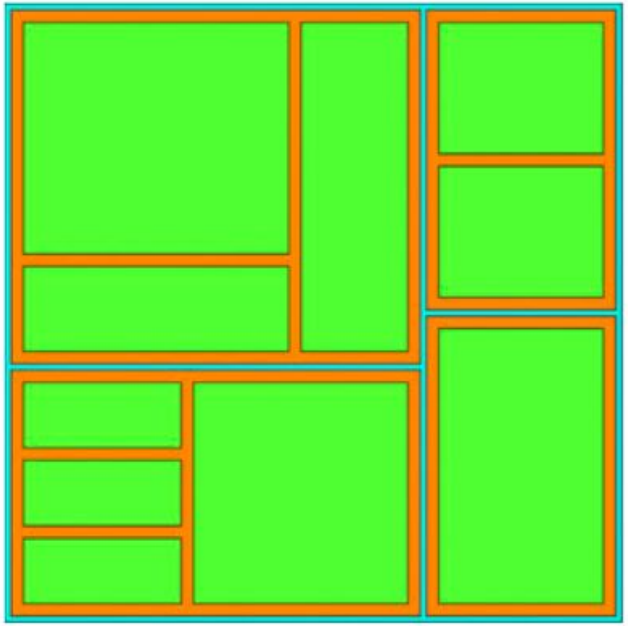
Treemaps

Let’s add padding:

```
var treemapLayout = d3.treemap()
  .size([500,500])
  .padding(d => d.depth * 5 + 5]
```

Treemaps

A treemap with the padding method set. Notice that padding determines the space between children and not siblings.



Treemaps

Filter zoom example

```
...
      .style("fill", d => depthScale(d.depth))
      .style("stroke", "black")
      .on("click", filterTreemap)

function filterTreemap(d) {
    var newRoot = d3.hierarchy(d.data, p => p.values)                    1
    .sum(p => p.retweets ? p.retweets.length + p.favorites.length + 1 :
undefined)

     treemapLayout(newRoot)
```

1 Build a new hierarchy using the currently clicked node as the root node

Treemaps

```
  d3.select("svg")
    .selectAll("rect")
    .data(newRoot.descendants(), p => p.data.content ||
p.data.user || p.data.key)
    .enter()
    .append("rect")                                          2
    .style("fill", p => depthScale(p.depth))
    .style("stroke", "black")

  d3.select("svg")
    .selectAll("rect")
    .data(newRoot.descendants(), p => p.data.content ||
p.data.user || p.data.key)
    .exit()
    .remove()                                               3
```

2 Add any new nodes (when we're zooming out)
3 Remove any trimmed nodes (when we're zooming in)

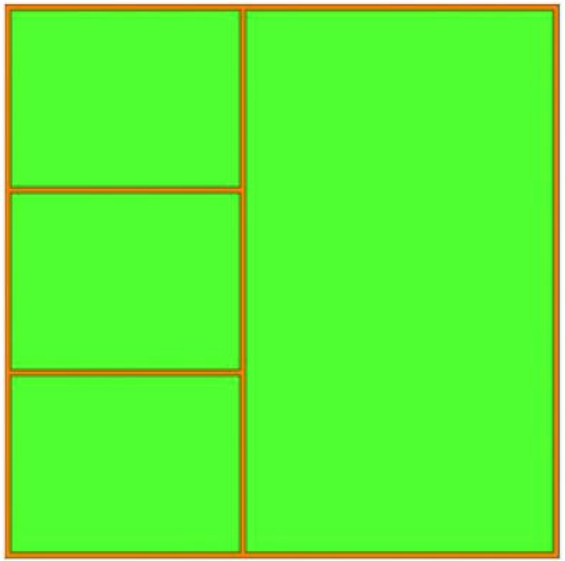
Treemaps

```
  d3.select("svg")
    .selectAll("rect")
    .on("click", d === root ?
    p => filterTreemap(p) : () => filterTreemap(root))                4
    .transition()
    .duration(1000)
    .attr("x", p => p.x0)                                        5
    .attr("y", p => p.y0)                                        5
    .attr("width", p => p.x1 - p.x0)                                 5
    .attr("height", p => p.y1 - p.y0)                                 5
}
```

4 Update the filter function so it zooms out if we're zoomed in and zooms in if we're zoomed out
5 Redraw any remaining nodes to the new scale

Treemaps

The “zoomed in” view of our treemap, showing only the leaf nodes in one of the intermediary node views. Note that the recalculated treemap has adjusted the padding because the orange node is now the root node.



Treemaps

A radial treemap accomplished by taking the drawing instructions from d3.treemap and using them to draw paths using d3.arc instead of svg:rect elements.