THE OHIO STATE UNIVERSITY

Introduction to D3.js, Part III

By Rui Li

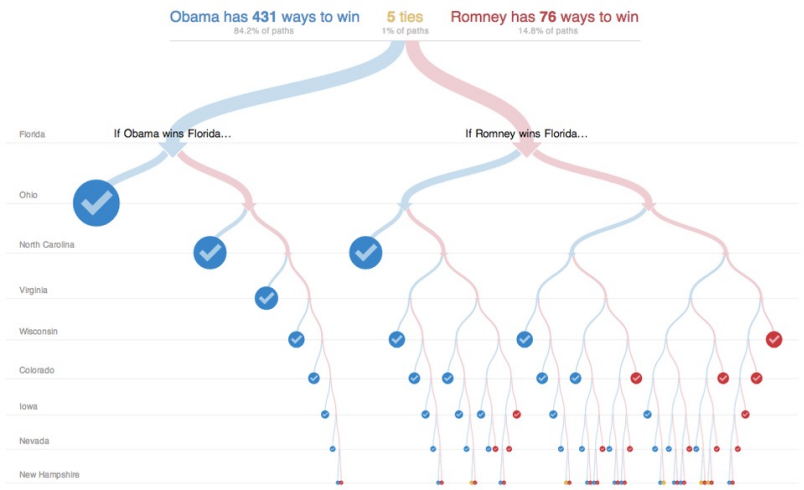
02/20/2023

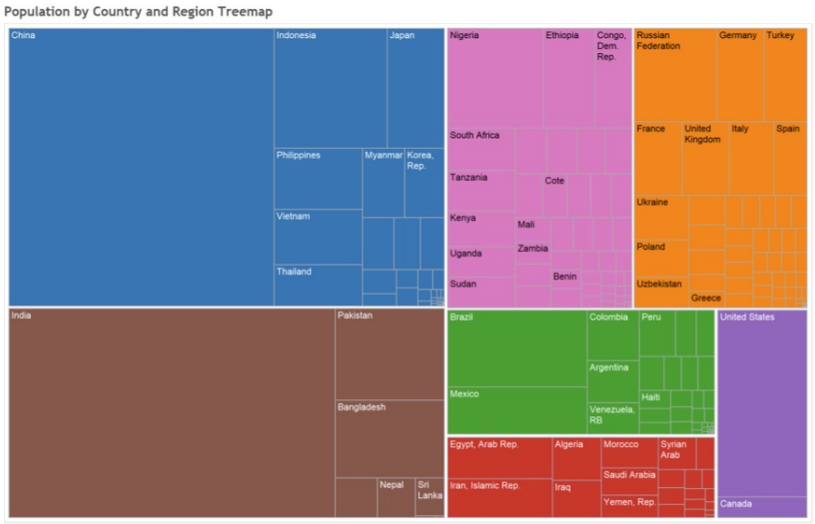
Slide Material Source Credits

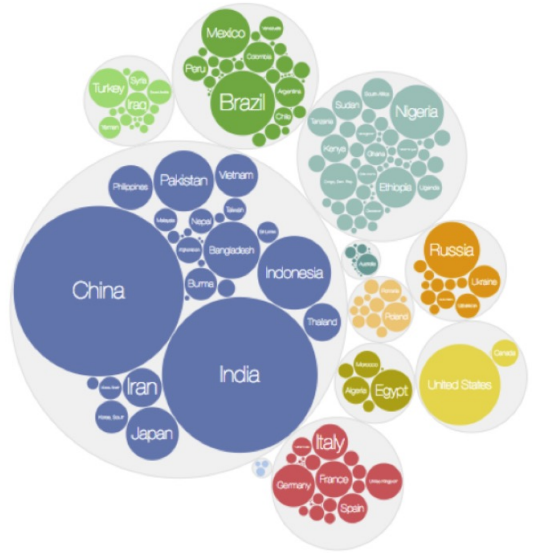
- https://www.d3indepth.com/
- https://d3js.org/
- https://www.d3-graph-gallery.com/
- HTML tutorial
- Prof. Han-Wei Shen, Jiayi Xu, and Wenbin He

Hierarchies

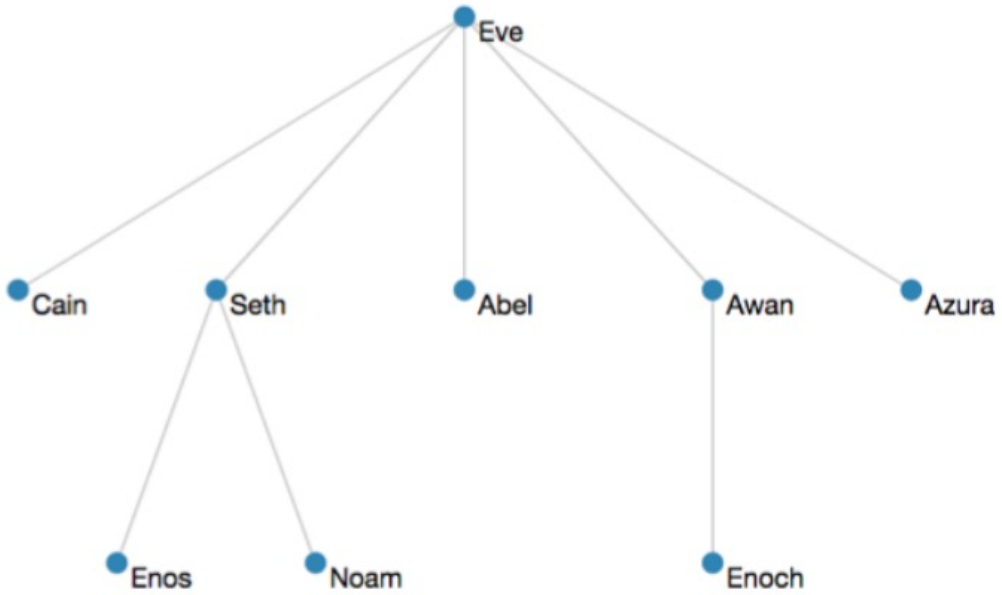
- Many datasets are intrinsically hierarchical (e.g., geographic entities)
- A common technique when analyzing or visualizing data is to organize your data into groups.







Hierarchical data




```
var data = {
  "name": "Eve",
  "children": [
    {
      "name": "Cain"
    },
    {
      "name": "Seth",
      "children": [
        {
          "name": "Enos"
        },
        {
          "name": "Noam"
        }
      ]
    },
    {
      "name": "Abel"
    },
    {
      "name": "Awan",
      "children": [
        {
          "name": "Enoch"
        }
      ]
    },
    {
      "name": "Azura"
    }
  ]
};
```

Name	Parent
Eve	
Cain	Eve
Seth	Eve
Enos	Seth
Noam	Seth
Abel	Eve
Awan	Eve
Enoch	Awan
Azura	Eve

d3.hierarchy

- d3.hierarchy is a nested hierarchical structure representing a tree
- Input: already in a hierarchical format (e.g., JSON), d3.hierarchy(data)

```
family = d3.hierarchy({
  name: "root",
  children: [
    {name: "child #1"},
    {
      name: "child #2",
      children: [
        {name: "grandchild #1"},
        {name: "grandchild #2"},
        {name: "grandchild #3"}
      ]
    }
  ]
})
```

```
family = ▼Zh {
  data: ▶Object {name: "root", children: Array(2)}
  height: 2
  depth: 0
  parent: null
  children: ▶Array(2) [Zh, Zh]
  x: 0
  y: 0
  <prototype>: ▶Zh {constructor: ƒ(t), count: ƒ(), e
}
```

d3.hierarchy

- Input: list representation (e.g., csv), d3.stratify(data)

```
chaos = d3.stratify()([
  {id: "Chaos"},
  {id: "Gaia", parentId: "Chaos"},
  {id: "Eros", parentId: "Chaos"},
  {id: "Erebus", parentId: "Chaos"},
  {id: "Tartarus", parentId: "Chaos"},
  {id: "Mountains", parentId: "Gaia"},
  {id: "Pontus", parentId: "Gaia"},
  {id: "Uranus", parentId: "Gaia"}
])
```

```
chaos = ▼ Zh {
  data: ▶ Object {id: "Chaos"}
  height: 2
  depth: 0
  parent: null
  id: "Chaos"
  children: ▶ Array(4) [Zh, Zh, Zh, Zh]
  x: 0
  y: 0
  <prototype>: ▶ Zh {constructor: ƒ(t), c
}
```

D3 Hierarchies – tree layout

The tree layout arranges the nodes of a hierarchy in a tree-like arrangement

- takes a hierarchy object
- configure the size using .size
- computes *x* and *y* attributes for each node

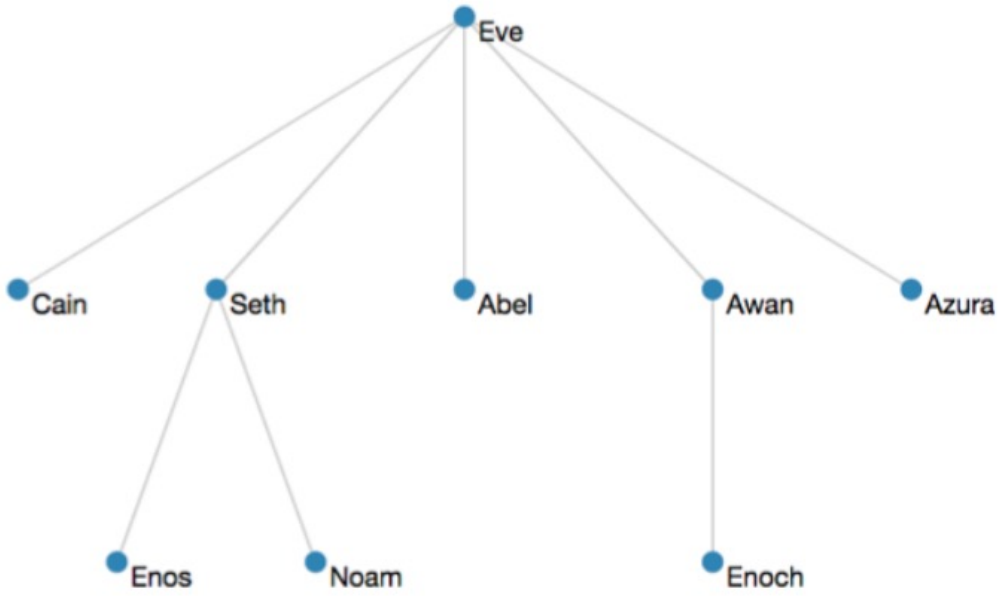
```
//convert a json data to a d3 hierarchy object
family = d3.hierarchy(json data...);

var treeLayout = d3.tree().size([400, 200]);
treeLayout(family);
```

```
var treeLayout = d3.tree().size([400, 200]);
treeLayout(family);
▼pd {data: {…}, height: 2, depth: 0, parent: null,
  ▼children: Array(2)
    ▼0: pd
      ▶data: {name: 'child #1'}
       depth: 1
       height: 0
      ▶parent: pd {data: {…}, height: 2, depth: 0,
       x: 100
       y: 100
      ▶[[Prototype]]: Object
    ▶1: pd {data: {…}, height: 1, depth: 1, parent:
     length: 2
    ▶[[Prototype]]: Array(0)
  ▶data: {name: 'root', children: Array(2)}
   depth: 0
   height: 2
   parent: null
   x: 150
   y: 0
  ▶[[Prototype]]: Object
```

D3 Hierarchies – tree layout

draw the tree

- draw nodes using circles
- draw links using lines

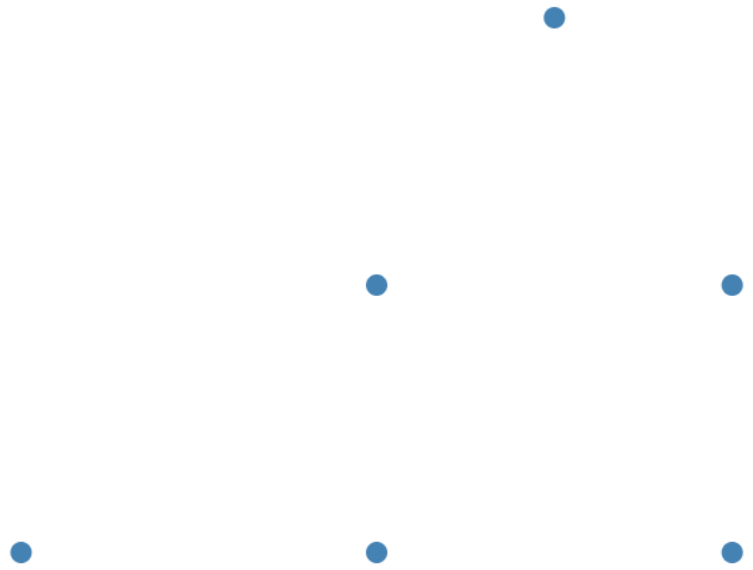


D3 Hierarchies – tree layout

draw the tree

- draw nodes using circles
  - use root.descendants() to get an array of all the nodes
  - join this array to circles (or any other type of SVG element)
  - use x and y to position the circles

```
// Nodes
d3.select('svg g.nodes')
  .selectAll('circle.node')
  .data(root.descendants())
  .join('circle')
  .classed('node', true)
  .attr('cx', function(d) {return d.x;})
  .attr('cy', function(d) {return d.y;})
  .attr('r', 4);
```

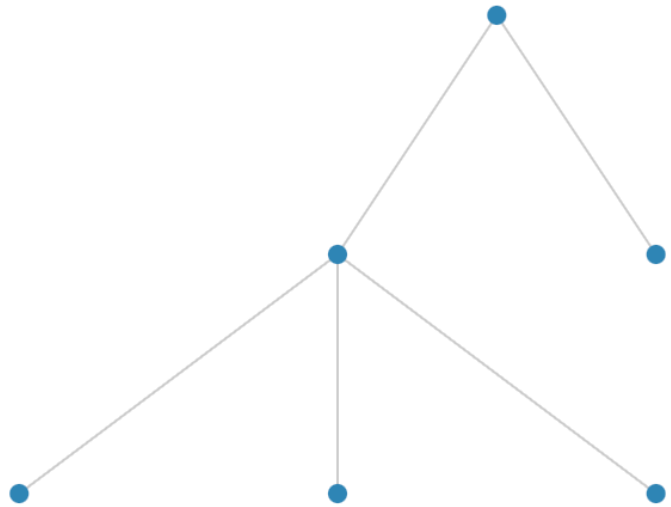


D3 Hierarchies – tree layout

draw the tree

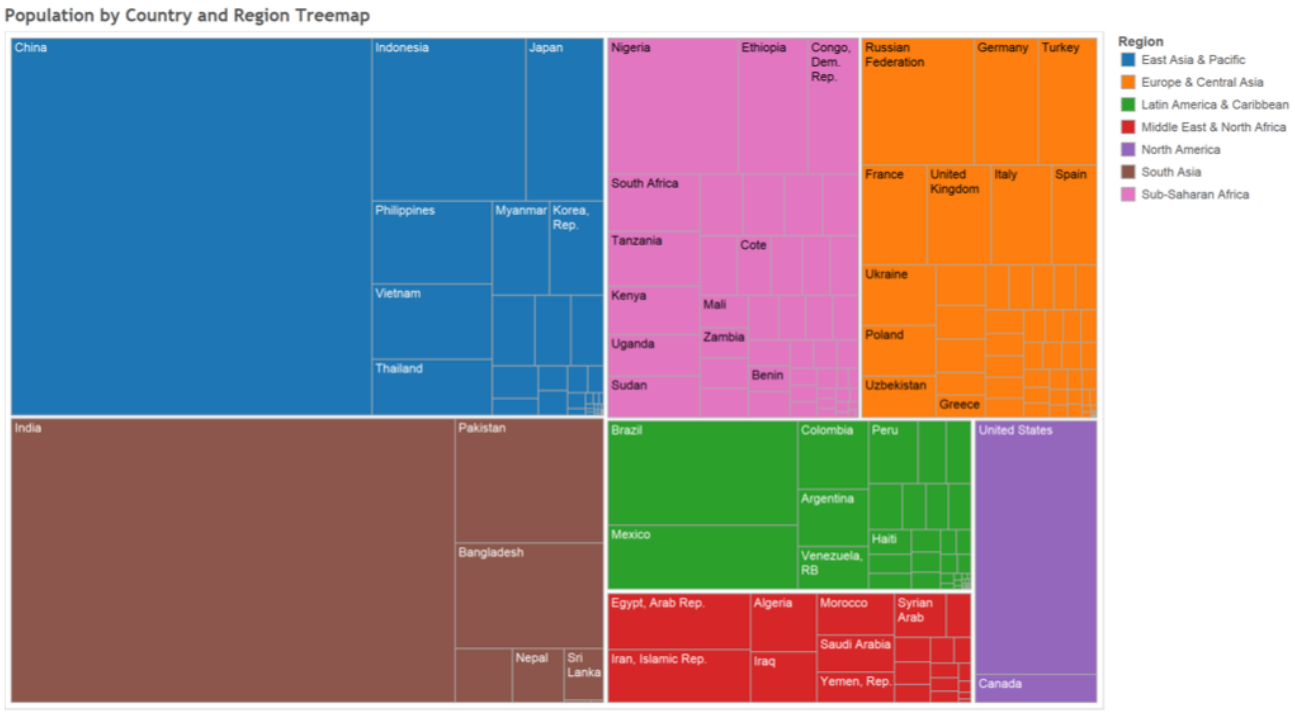
- draw links using lines
  - use root.links() to get an array of all the links
  - join the array to line (or path) elements
  - use x and y of the link's source and target properties to position the line

```
// Links
d3.select('svg g.links')
  .selectAll('line.link')
  .data(root.links())
  .join('line')
  .classed('link', true)
  .attr('x1', function(d) {return d.source.x;})
  .attr('y1', function(d) {return d.source.y;})
  .attr('x2', function(d) {return d.target.x;})
  .attr('y2', function(d) {return d.target.y;});
```



Treemap

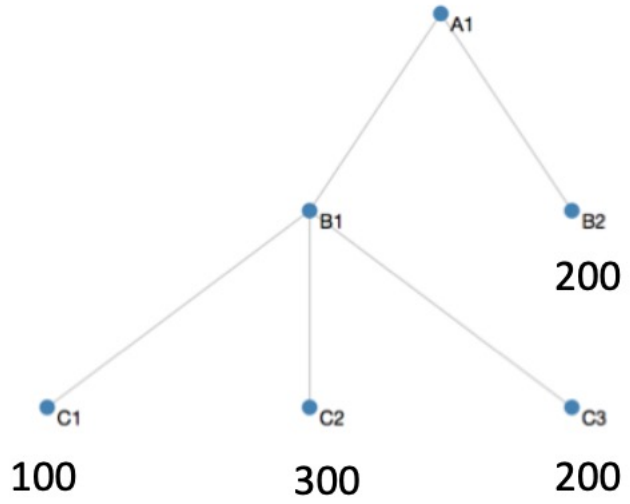
- Treemap can visually represent hierarchies where each item has an associated value
- For example, we can think of country population data as a hierarchy
  - The first level represents the region
  - The next level represents each country.



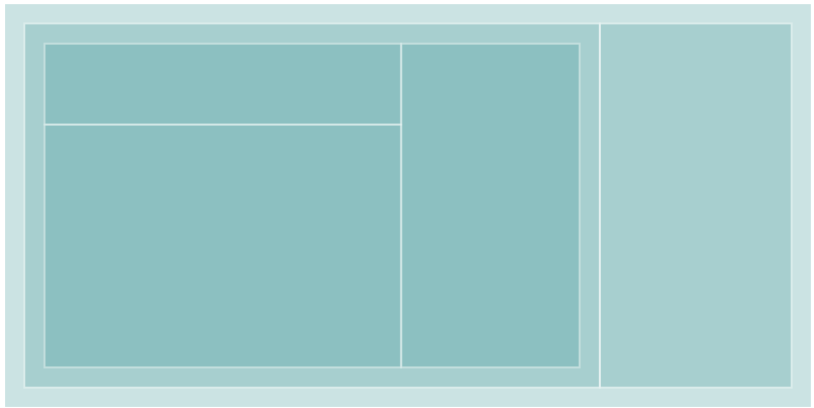
Treemap

Example

- Data
  - A fake hierarchical data
  - Each leaf node has a quantity value




```
var data = {
  "name": "A1",
  "children": [
    {
      "name": "B1",
      "children": [
        {
          "name": "C1",
          "value": 100
        },
        {
          "name": "C2",
          "value": 300
        },
        {
          "name": "C3",
          "value": 200
        }
      ]
    },
    {
      "name": "B2",
      "value": 200
    }
  ]
};
```



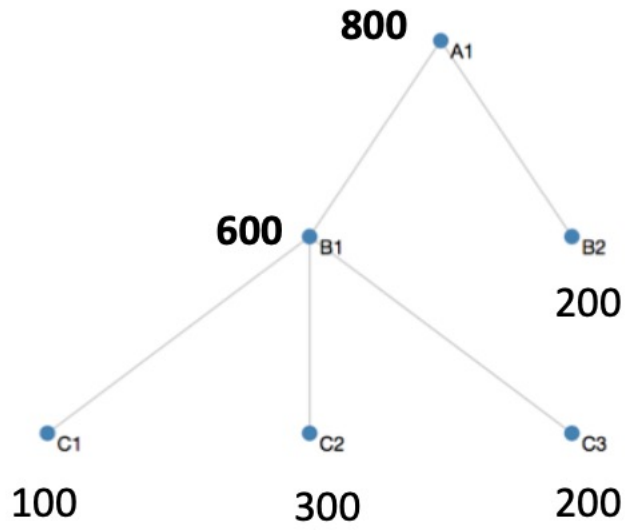
Treemap

- Draw a treemap
  - Construct the hierarchy structure

```
var root = d3.hierarchy(data);
```

  - Calculate values of parents
    - Equals to sum of children's values
    - *node*.sum() can calculates the sums automatically

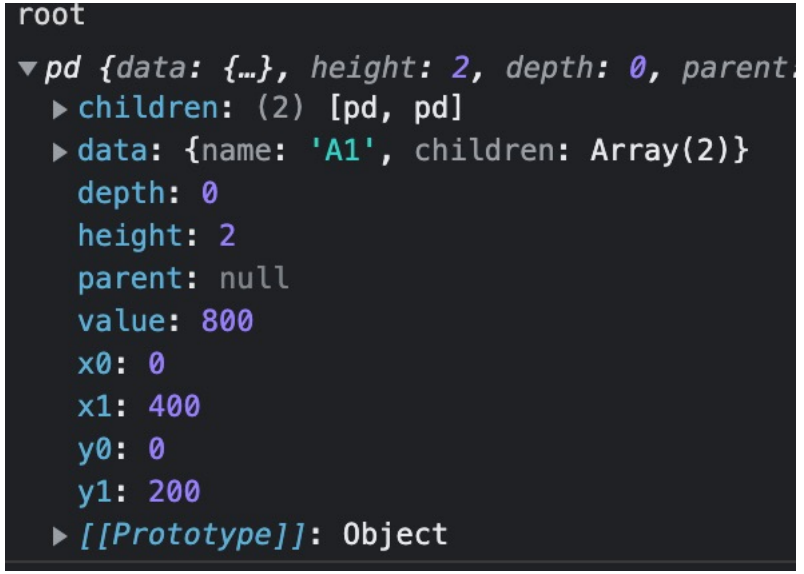
```
root.sum(function(d) {
  return d.value;
});
```



Treemap

- Draw a treemap
  - Treemap generator: d3.treemap()
    - Take the screen size and padding/gaps between rectangles
    - Compute the coordinates of rectangles

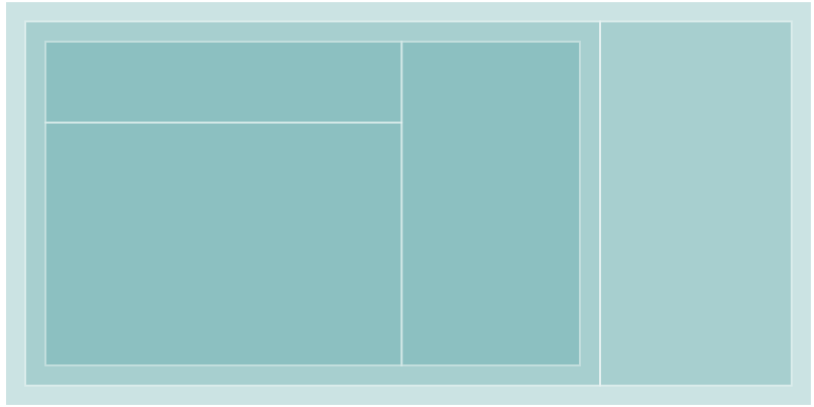
```
var treemapLayout = d3.treemap()
  .size([400, 200])
  .paddingOuter(10);

treemapLayout(root);
```



Treemap

- Draw a treemap
  - Draw rectangles by
    - top-left corner *(x0, y0)*
    - bottom-right corner *(x1, y1)*

```
d3.select('svg g')
  .selectAll('rect')
  .data(root.descendants())
  .join('rect')
  .attr('x', function(d) { return d.x0; })
  .attr('y', function(d) { return d.y0; })
  .attr('width', function(d) { return d.x1 - d.x0; })
  .attr('height', function(d) { return d.y1 - d.y0; })
```



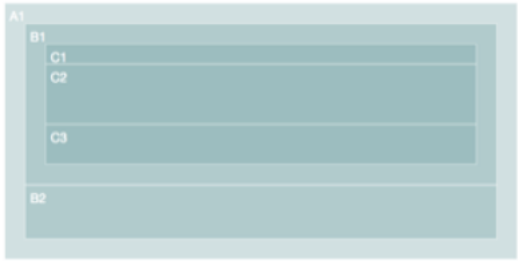
Treemap

- Tiling strategies
  - By default: squarified: generate rectangles with a golden aspect ratio

```
var treemapLayout = d3.treemap()
  .size([400, 200])
  .paddingOuter(10)
  .tile(d3.treemapDice);
```

Golden ratio

d3.treemapDice

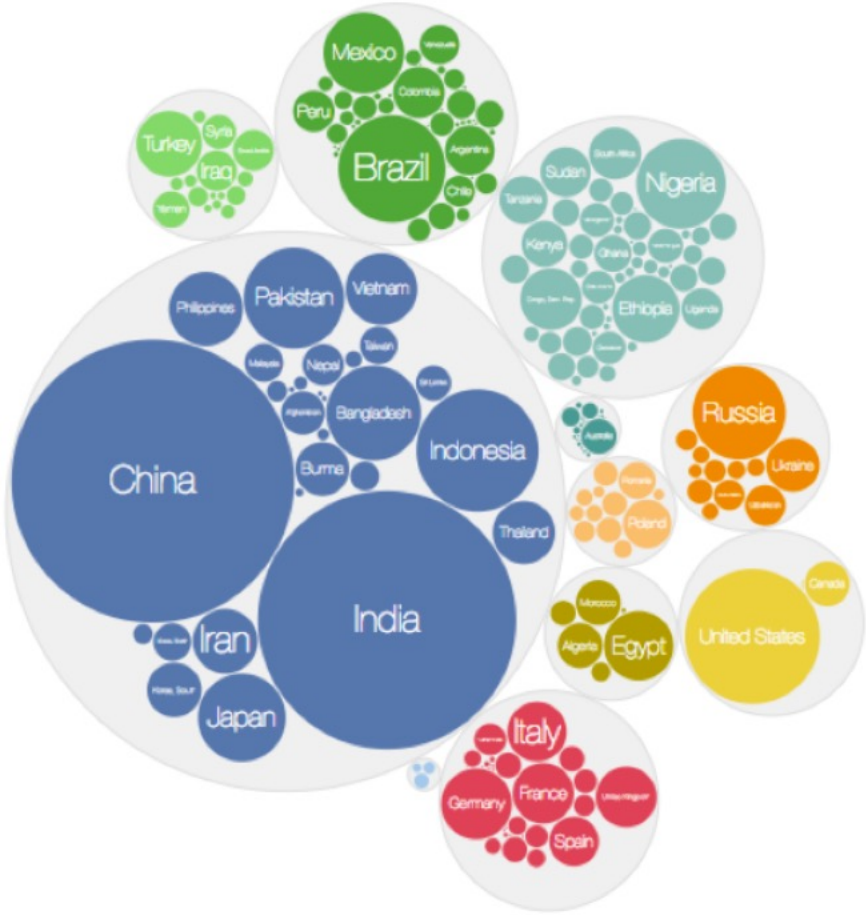


d3.treemapSlice

Pack Layout

The pack layout is similar to the treemap layout

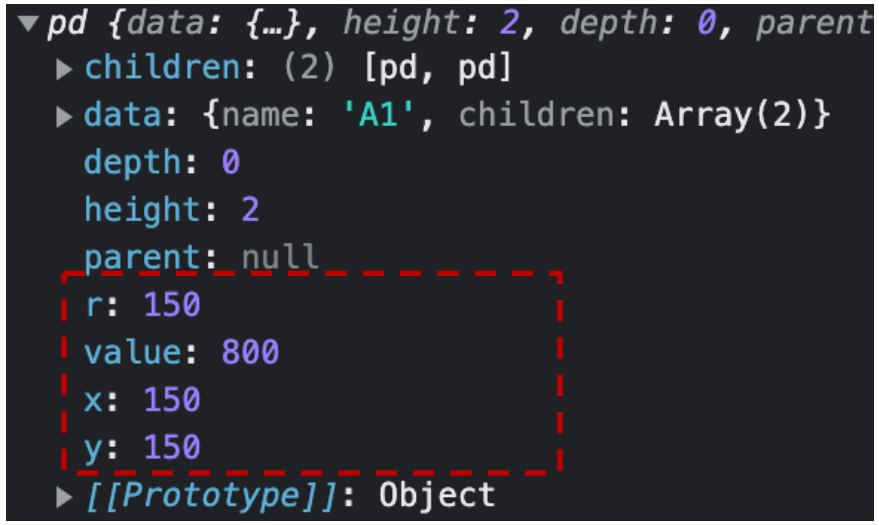
- **But circles instead of rectangles** are used to represent nodes.
- For example, we could visualize region and country data using pack layout



Pack Layout

- Draw a circular pack chart

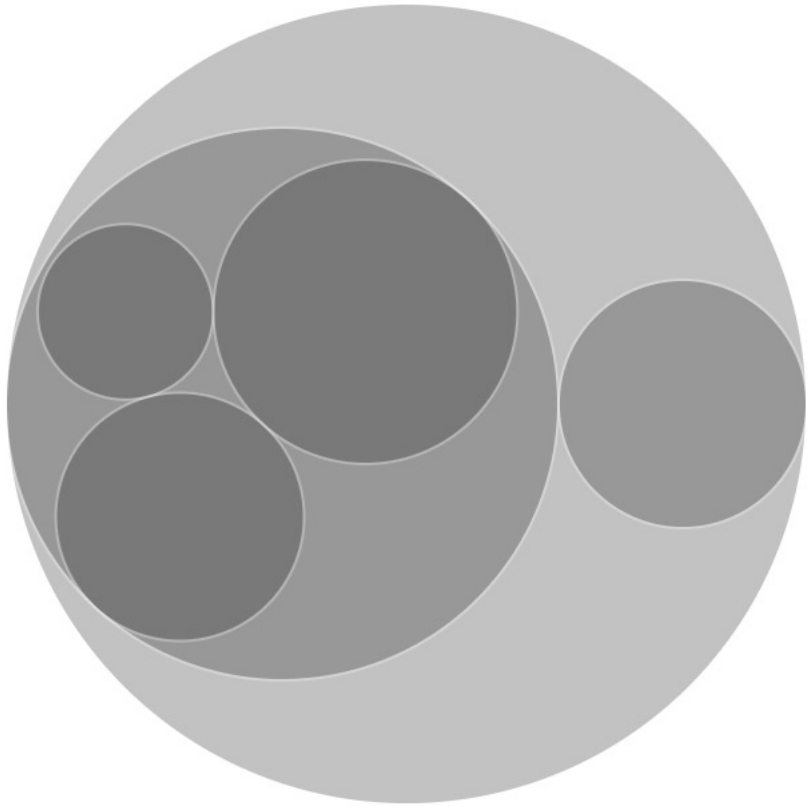
```
var rootNode = d3.hierarchy(data)

//create the pack layout
var packLayout = d3.pack()
  .size([300, 300]);

//calculate values of parents
rootNode.sum(function(d) {
  return d.value;
});

//calculate the coordinates of circles
packLayout(rootNode);
```



Pack Layout

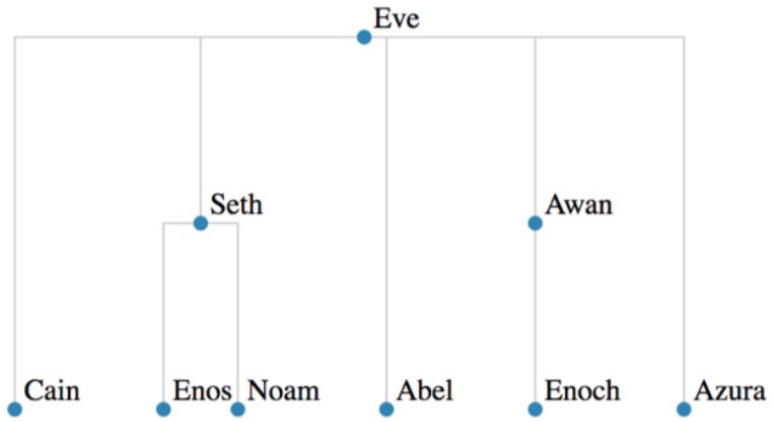
- Draw a circular pack chart
  - Draw circles by
    - center point (cx, cy)
    - radius (r)

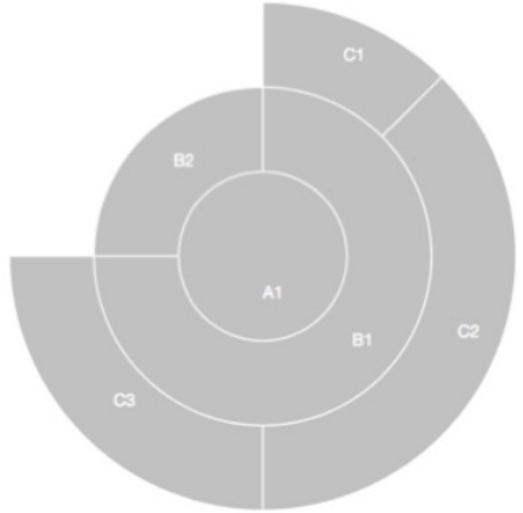
```
d3.select('svg g')
  .selectAll('circle')
  .data(rootNode.descendants())
  .join('circle')
  .attr('cx', function(d) { return d.x; })
  .attr('cy', function(d) { return d.y; })
  .attr('r', function(d) { return d.r; })
```



D3 Hierarchies – More Layouts

- cluster
- partition

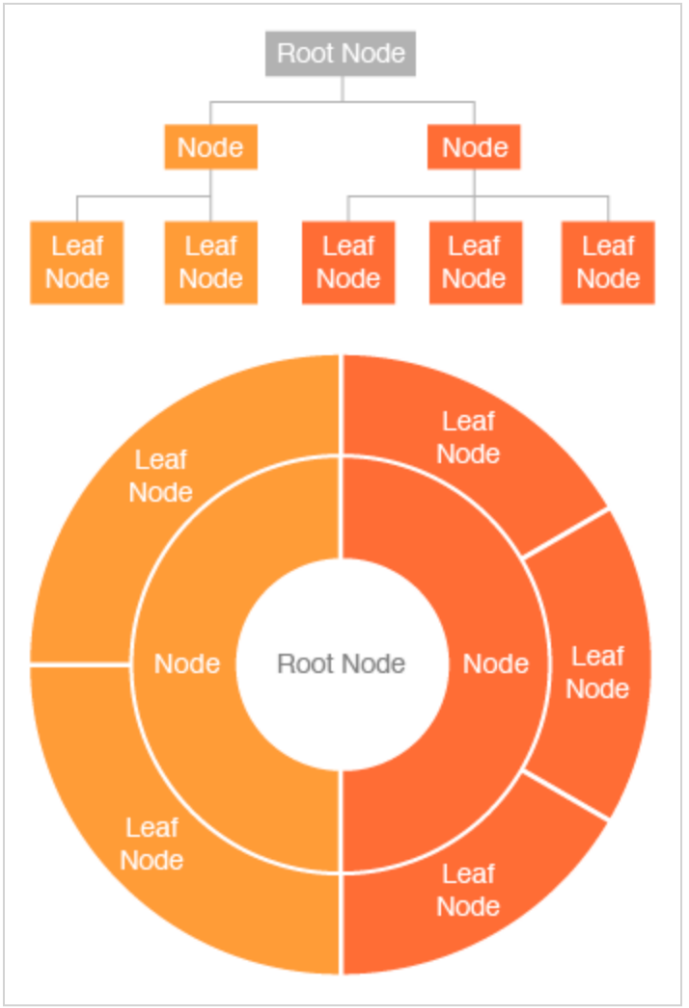




Sunburst Diagram

- Shows hierarchy through a series of rings that are sliced for each category node
- Each ring corresponds to a level in the hierarchy
- The central circle representing the root node

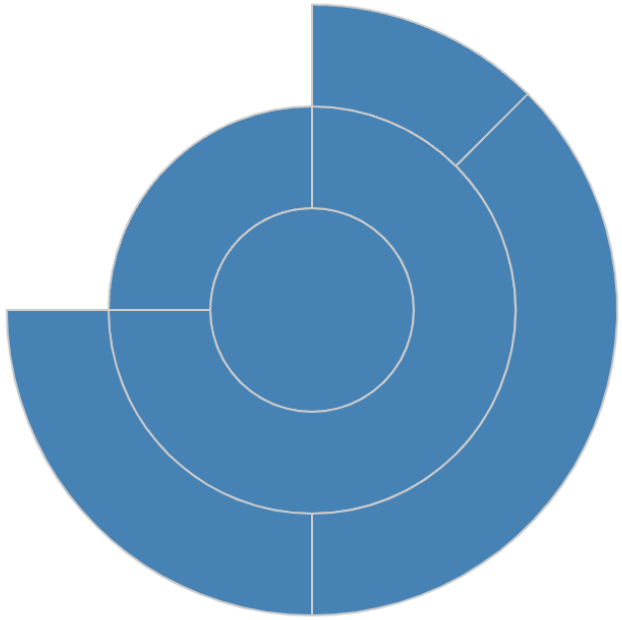
https://observablehq.com/@d3/zoomable-sunburst



https://datavizcatalogue.com/methods/sunburst_diagram.html

Sunburst Diagram

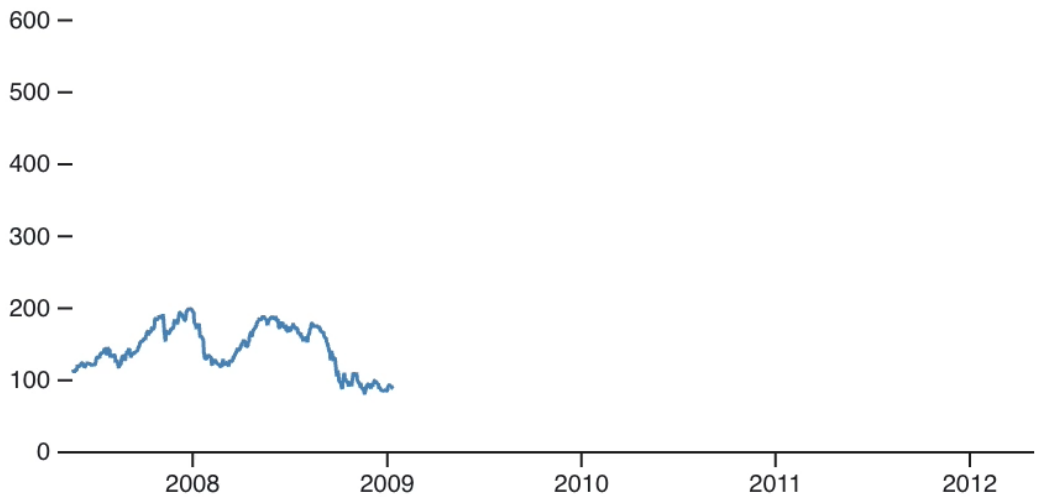
```
const radius = 150;

const arcGenerator = d3.arc()
    .startAngle(function (d) { return d.x0; })
    .endAngle(function (d) { return d.x1; })
    .innerRadius(function (d) { return d.y0; })
    .outerRadius(function (d) { return d.y1; });

const rootNode = d3.hierarchy(data);

rootNode.sum(function (d) {
    return d.value;
});

const partitionLayout = d3.partition()
    .size([2 * Math.PI, radius]);

partitionLayout(rootNode);
```

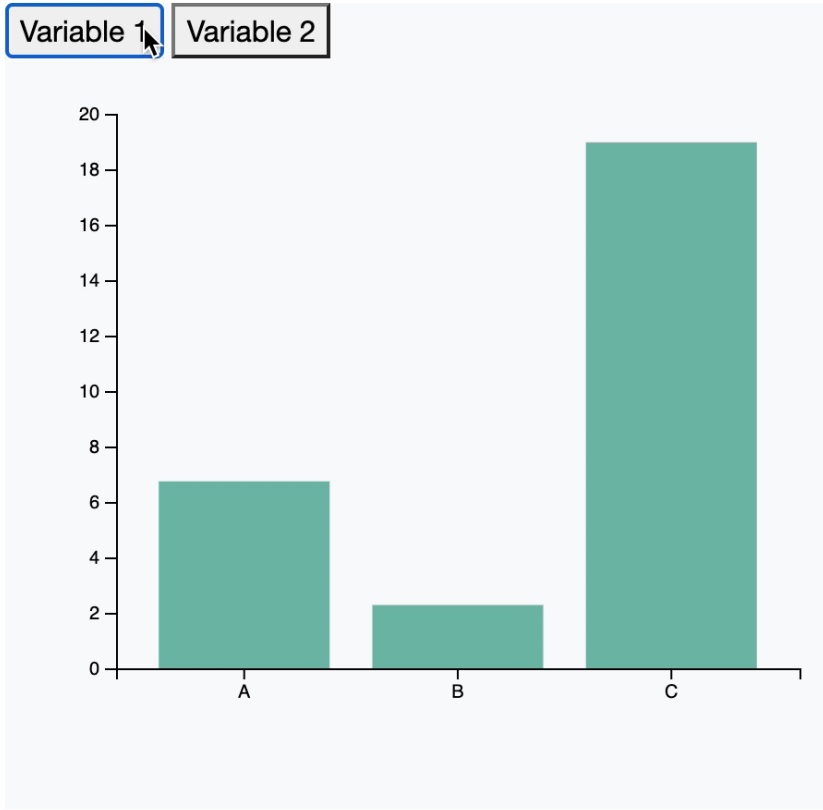


D3 Transitions

D3 transitions let you smoothly **animate** between different **chart states**.

- Example:
  - timeseries datasets
  - update dataset





D3 Transitions

Create a d3 transition

- transition()
- add it before calling .attr() or .style methods.

```
function update() {
  d3.select('svg')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cy', 50)
    .attr('r', 40)
    .transition()
    .attr('cx', function(d) {
      return d;
    });
}
```

cx attribute of the circles transitioning into new positions

D3 Transitions

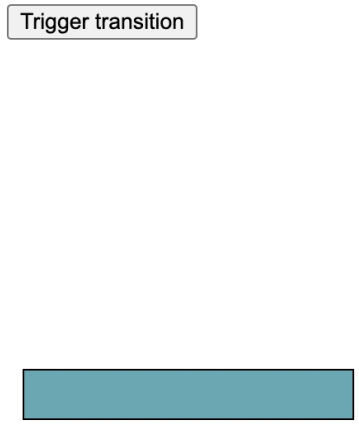
Transition methods:

- duration(time)
- delay(time)
- easing effect

```
d3.select('svg')
  .selectAll('circle')
  .data(data)
  .join('circle')
  .attr('cy', 50)
  .attr('r', 40)
  .transition()
  .delay(function(d, i) {
    return i * 75;
  })
  .attr('cx', function(d) {
    return d;
  });
```

Chained transitions

- Transitions can be chained together by adding multiple calls to .transition. Each transition takes its turn to proceed. (When the first transition ends, the second one will start, and so on.)




```
d3.select("#my_rect")
  // First, make the bar wider
  .transition()
  .duration(2000)
  .attr("width", "400")
  // Second, higher
  .transition()
  .attr("height", "100")
  // Change its color
  .transition()
  .style("fill", "#69b3a2")
  // And now very small
  .transition()
  .duration(200)
  .attr("height", "10")
  .attr("width", "10")
```

Customizing transitions

- D3 provided default transition behavior. However, there are times when the default behavior is not suitable.
- We can customize the transition behavior by:
  - Writing your own tween function


```
d3.select('svg g')
  .selectAll('circle')
  .data(data)
  .join('circle')
  .transition()
  .tween('my-tween', function(d) {
    ...

    // Return a tween function
    return function(t) {
      // Do stuff here such as updating
    }
  });
```

Customizing transitions

- D3 provided default transition behavior. However default behavior is not suitable.
- We can customize the transition behavior by:
  - Writing your own tween function
  - Specifying details when entering and exiting elements

```
d3.select('svg')
  .selectAll('circle')
  .data(data)
  .join(
    function(enter) {
      return enter
        .append('circle')
        .attr('cy', 50)
        .attr('cx', function(d) { return d; })
        .attr('r', 40)
        .style('opacity', 0);
    },
    function(update) {
      return update;
    },
    function(exit) {
      return exit
        .transition()
        .attr('cy', 500)
        .remove();
    }
  )
  .transition()
  .attr('cx', function(d) { return d; })
  .style('opacity', 0.75);
```

D3 Interactions

It is often required to interact with our visualizations, e.g., hovering, zooming, clicking etc., to change the appearance of the visual elements or drill drown information.

examples:

- zoom, panning, and selection
  - http://web.cse.ohio-state.edu/~li.8950/project/DVLab4/dvlab4.html
- lasso selection
  - https://observablehq.com/@bumbeishvili/d3-lasso-selection
- brush
  - https://observablehq.com/@d3/focus-context

D3 Interactions

- Mouse Events
- Tooltip
- Zoom and pan
- Drag
- Brush and Lasso section

D3 Mouse Events

- Mouse events like *click*, *mousedown*, *mouseover* etc. are vey common in UI interaction
- *selection*.on(*EventType*, *listener*)
  - Register an event listener to a selection
  - *EventType* is the name (string) of an event type, e.g., *click*, *mouseover*, etc.
    Any DOM event type supported by your browser may be used (not only mouse events)
  - Event list: https://developer.mozilla.org/en-US/docs/Web/Events#Standard_events
  - When a specified event is triggered, the *listener* function will be invoked

D3 Mouse Events

- **mousedown**, depresses the mouse button on the element
- **mouseup**, user releases the mouse button on the element
- **click**, one mousedown and one mouseup detected on the element
- **mouseout**: Fires when the mouse leaves the canvas or any of its children.
- **mouseover**: Fires when the mouse enters the canvas or any of its children

```
svg.append('g')
        .selectAll('dot')
        .data(data)
        .join('circle')
        .attr('cx', function(d, i) {
            return x(d.Sepal_Length);
        })
        .attr('cy', function(d, i) {
            return y(d.Petal_Length);
        })
        .attr('r', 4)
        .style('fill', function(d, i) {
            return color(d.Species);
        })
        .on("mouseover", function() {
            d3.select(this).attr('r', 10);
        })
        .on("mouseout", function() {
            d3.select(this).attr('r', 4);
        })
```

D3 Mouse Events

- Event manager
  - You can get the triggered event and the data item through the following syntax

```
selection.on("mousemove", function(event, d) {
  … do something with event and d …
})
```

  - You can get the event target by calling:
    - event.currentTarget gives access to the element to which the listener is bound and can replace this in arrow functions
    - this

```
MouseEvent {isTrusted: tru
  i
  isTrusted: true
  altKey: false
  bubbles: true
  button: 0
  buttons: 0
  cancelBubble: false
  cancelable: true
  clientX: 394
  clientY: 384
  composed: true
  ctrlKey: false
  currentTarget: null
  defaultPrevented: false
```

```
//es6 arrow function
selection.on("click", (event, d) => {
    console.log(event.currentTarget);
})

//conventional function
selection.on("click", function(event, d) {
    console.log(this);
})
```

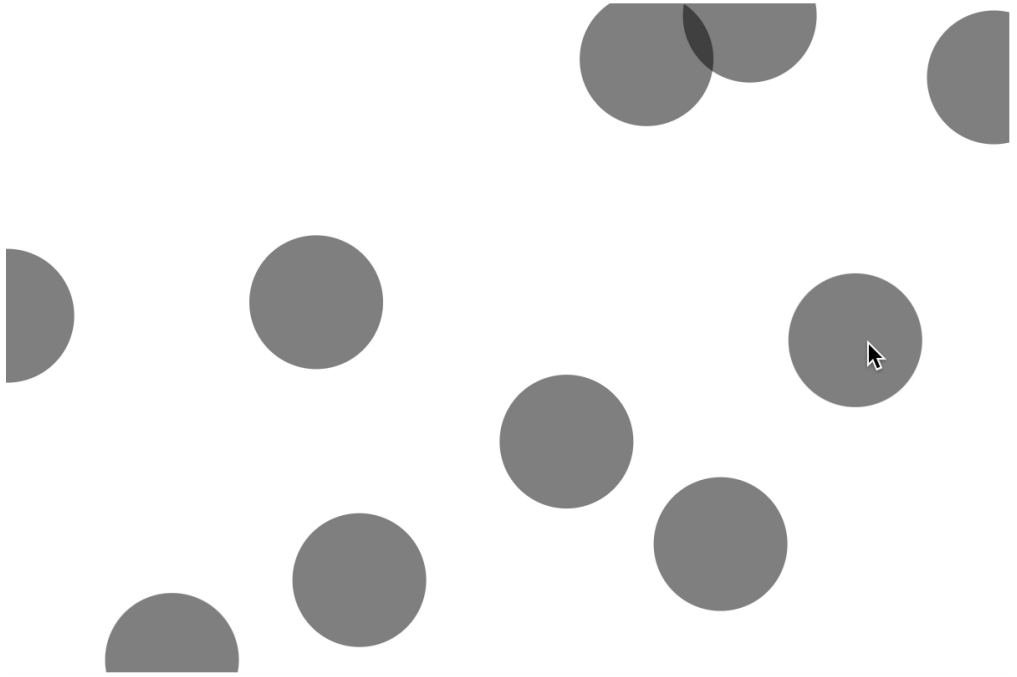
D3 tooltip

- Create a blank tooltip div
- Display the tooltip when mouseover or mousemove
  - determine the position of the tooltip based on the mouse event
  - update the tooltip content based on the data item (d)
- Disable the tooltip when mouseleave
- Example

```
//es6 arrow function
selection.on("click", (event, d) => {
    console.log(event.currentTarget);
})

//conventional function
selection.on("click", function(event, d) {
    console.log(this);
})
```

D3 Drag

- Drag-and-drop is a popular and easy-to-learn pointing gesture
  - move the pointer to an object
  - press and hold to grab it
  - “drag” the object to a new location • release to “drop”



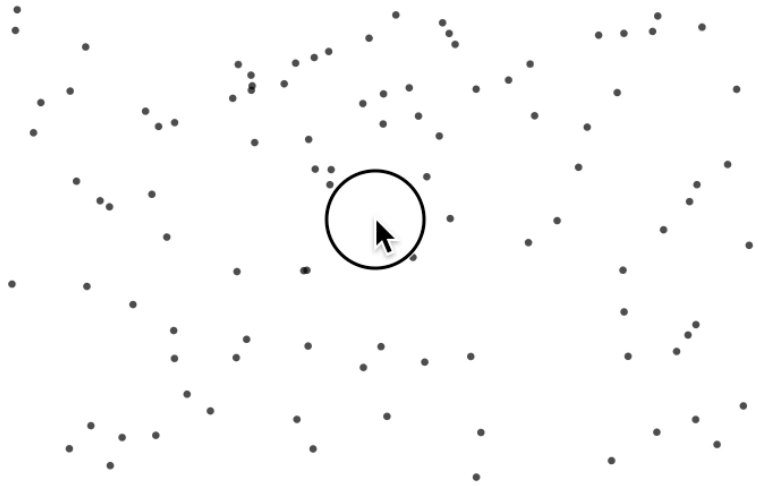
D3 Drag

- create a **drag behavior** function
- add an event handler that's called when a drag events occurs
- attach the drag behavior to the elements

```
//step 1
var drag = d3.drag()
    .on("start", dragStarted)
    .on('drag', dragged)
    .on("end", dragEnded);

//step 2
function dragStarted() {
    d3.select(this).attr("stroke", 'black');
}

function dragged(event) {
    d3.select(this).attr("cx", event.x).attr('cy', event.y);
}

function dragEnded() {
    d3.select(this).attr("stroke", null);
}

//step 3
svg.append('g')
    .selectAll('dot')
    ...
    .call(drag)
```

D3 Zoom and Pan

- D3 provides a module that adds **zoom** and **pan** behavior to an HTML or SVG element.
- Let's use d3.zoom() to enable our scatterplot to support zooming and panning



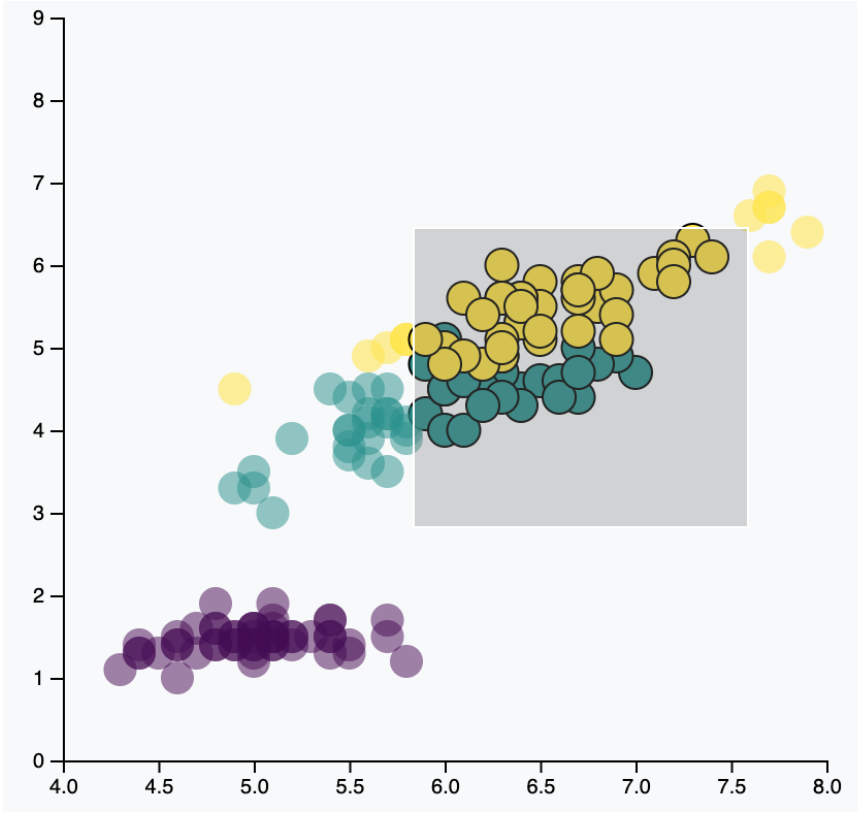
D3 Zoom and Pan

- create a **zoom behavior** function
- add an event handler that gets called when a zoom or pan event occurs.
- attach the zoom behavior to an element

```
//step 1
var zoom = zoom = d3.zoom()
    .scaleExtent([0.5, 32])
    .on("zoom", zoomed);

//step 2
function zoomed(event) {
    //create new scale objects
    var new_xScale = event.transform.rescaleX(x);
    var new_yScale = event.transform.rescaleY(y);

    points.attr('transform', event.transform);
    points.attr("display", function(d) {
        if (new_xScale(d.Sepal_Length) < 60 ||
            new_xScale(d.Sepal_Length) > width + 60 ||
            new_yScale(d.Petal_Length) < 0 ||
            new_yScale(d.Petal_Length) > height) {
            return 'none'
        }
    });

    xAxis.call(d3.axisBottom(x).scale(new_xScale));
    yAxis.call(d3.axisLeft(y).scale(new_yScale));
};

//step 3
svg.call(zoom);
```

D3 Brush

- Brushing lets you user **specify an area** (by pressing the mouse button, moving the mouse, then releasing) in order to, for example, select a group of elements.



D3 Brush

- create a **brush behavior** function
- add an event handler that's called when a brush event occurs.
- attach the brush behavior to elements

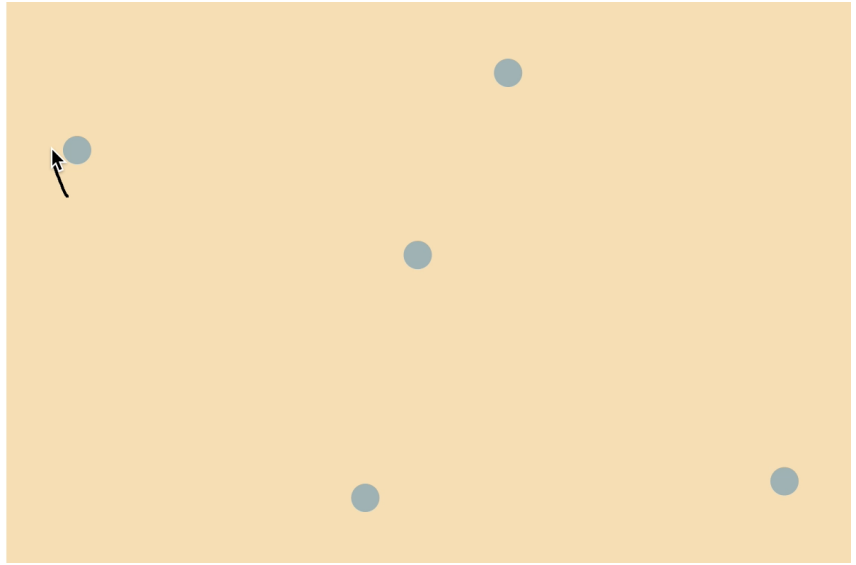
```
let brush = d3.brush()
    .on('brush', handleBrush);

function handleBrush(event) {
    // get the brush boundingbox
    var extent = event.selection;
    points.style("fill", function(d) {
        let ifIncluded =
            x(d.Sepal_Length) >= extent[0][0] &&
            x(d.Sepal_Length) <= extent[1][0] &&
            y(d.Petal_Length) >= extent[0][1] &&
            y(d.Petal_Length) <= extent[1][1];
        if (ifIncluded) {
            return 'purple';
        } else {
            return color(d.Species);
        }
    })
}

svg.call(brush);
```

D3 lasso selection

```
function dragStart() {
    coords = [];
    circles.attr("fill", "steelblue");
    d3.select("#lasso").remove();
    d3.select("#chart")
        .append("path")
        .attr("id", "lasso");
}

function dragMove(event) {
    let mouseX = event.sourceEvent.offsetX;
    let mouseY = event.sourceEvent.offsetY;
    coords.push([mouseX, mouseY]);
    drawPath();
}

function dragEnd() {
    let selectedDots = [];
    circles.each((d, i) => {
        let point = [
            xScale(d.x),
            yScale(d.y),
        ];
        if (pointInPolygon(point, coords)) {
            d3.select("#dot-" + d.id).attr("fill", "red");
            selectedDots.push(d.id);
        }
    });
    console.log(`select: ${selectedDots}`);
}
```

Remove previous lasso selections and append the new lasso path to the SVG

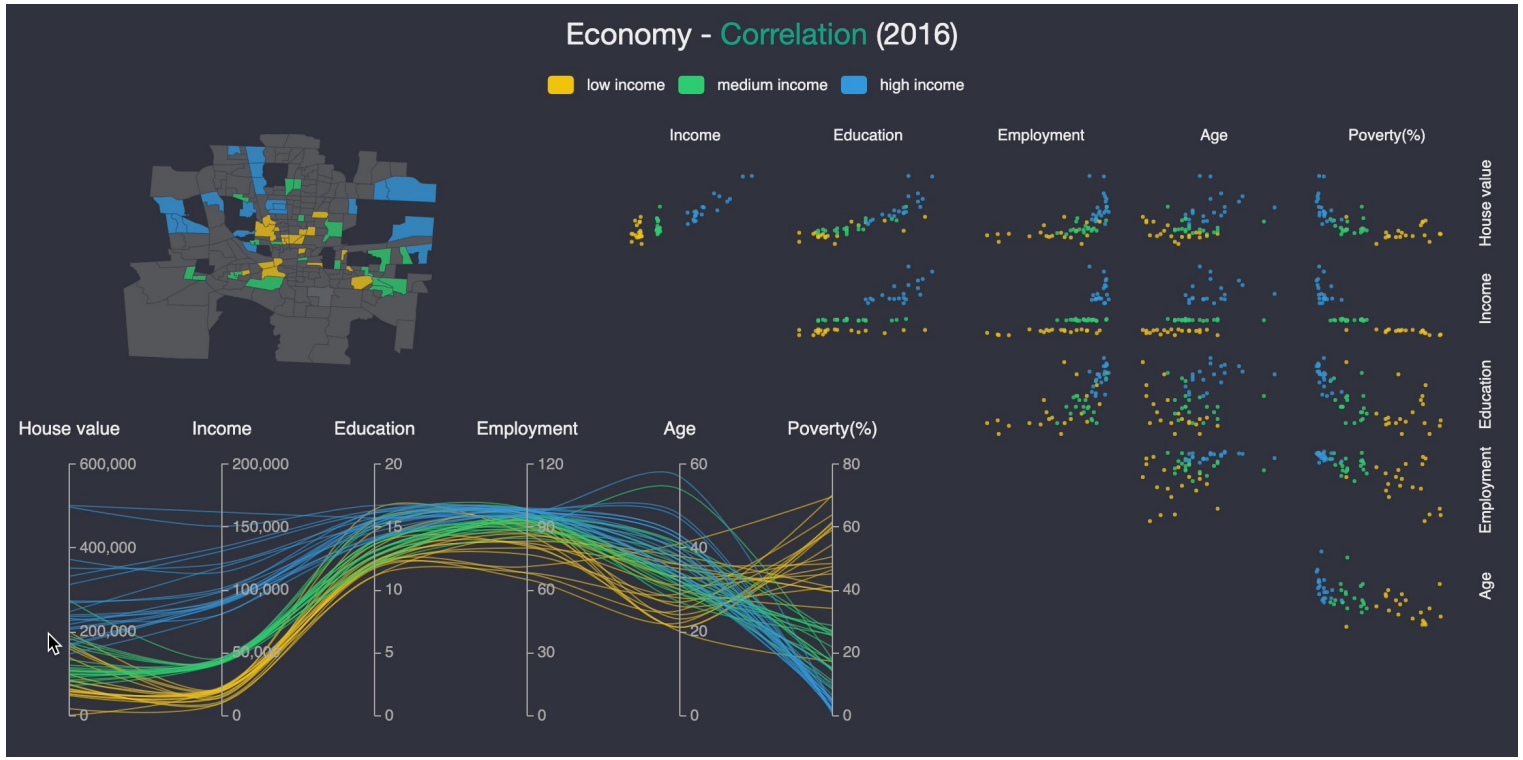
- Put the current mouse position (mouseX and mouseY) into an array (coords).
- Render a path based on the coords (using lineGenerator)



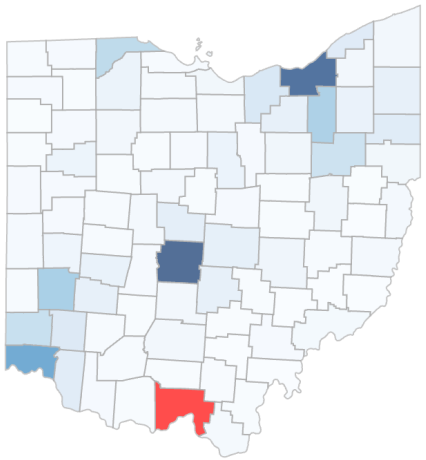
For each element on the SVG, check whether it's inside or outside the lasso boundary (determine if a point is inside a polygon)

Multi-view visualization

- Triggered by interactions such as clicking and mouseover, etc.
- Attribute **ID** is very useful in selecting elements from a different view!

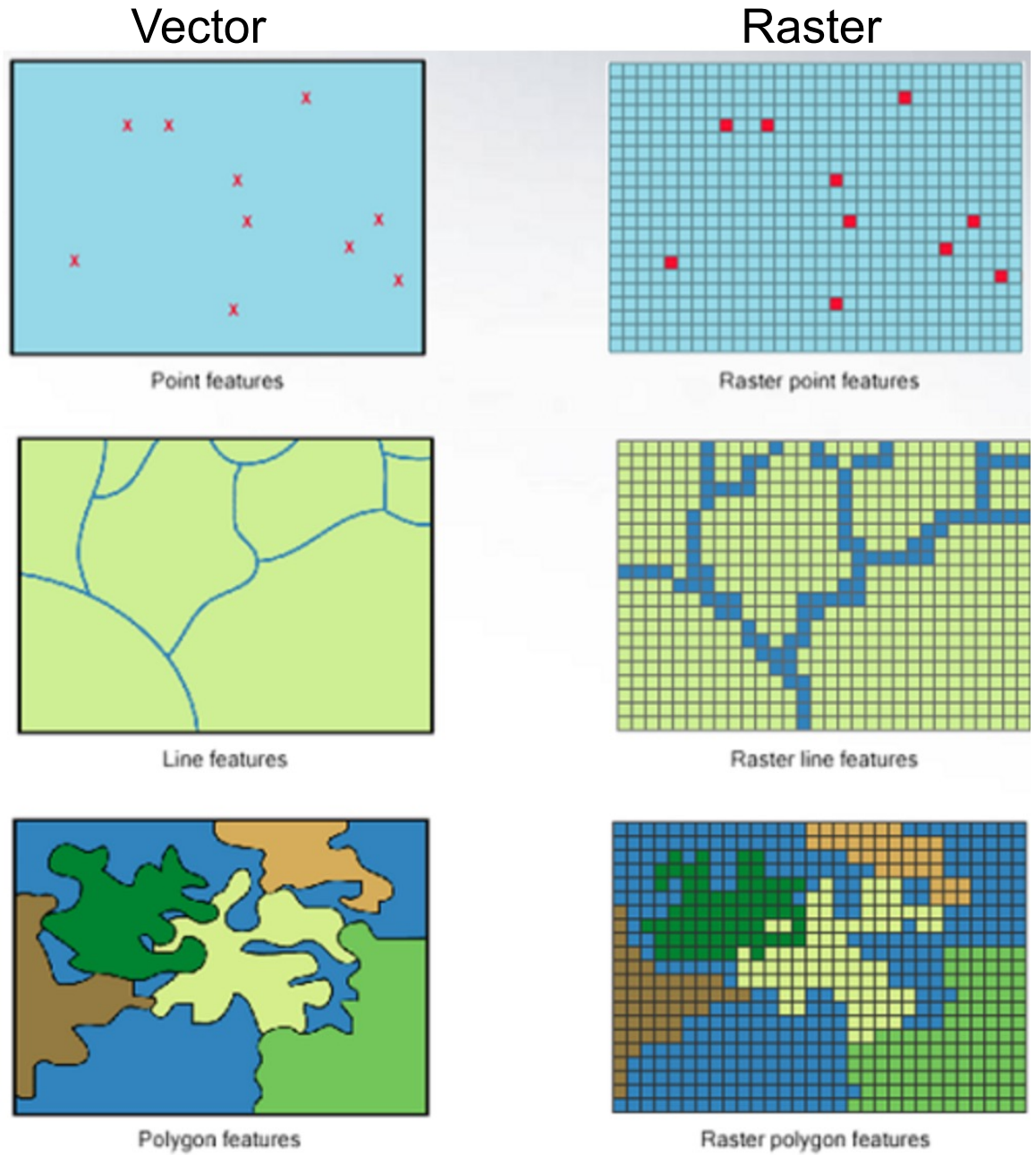


d3.select("#C-Scioto").style("fill","red")



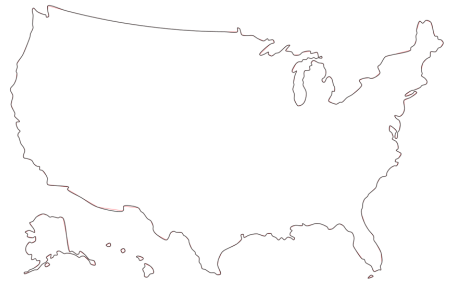
Geospatial Visualization

- Raster map
  - based on raster tiles
- Vector map
  - smaller
  - more aesthetically-pleasing
  - many vector map formats: Shp, GeoJson, TopoJson, PBF, etc.
  - D3 uses GeoJson



credit: https://desktop.arcgis.com/en/arcmap/10.3/manage-data/raster-and-images/how-features-are-represented-in-a-raster.htm

GeoJson Format



- Vector map geometries
  - points/lines/polygons
- GeoJSON format
  - specification (RFC 7946)
  - typically contains a collection of features
  - each feature contains:
    - properties
    - geometry information
  - Converting tools
    - shp2json
    - QGIS
    - GDAL

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": {
        "name": "Africa"
      },
      "geometry": {
        "type": "Polygon",
        "coordinates": [[[-6, 36], [33, 30], ... , [-6, 36]]]
      }
    },
    {
      "type": "Feature",
      "properties": {
        "name": "Australia"
      },
      "geometry": {
        "type": "Polygon",
        "coordinates": [[[143, -11], [153, -28], ... , [143, -11]]]
      }
    },
```

GeoJSON Format

- Load GeoJSON data
  - d3.json()
- Find GeoJSON data
  - Natural Earth
  - Census Bureau
  - QGIS
  - great tutorial

Map projections

- **Input**: [longitude, latitude]
- **Output**: pixel coordinate [x, y] (depends on the projection type)

```
let projection = d3.<projectType>();

projection( [-3.0026, 16.7666] );
```

scale	Scale factor of the projection
center	Projection center [longitude, latitude]
translate	Pixel [x,y] location of the projection center
rotate	Rotation of the projection [lambda, phi, gamma] (or [yaw, pitch, roll])

- Example

```
let projection = d3.geoAzimuthalEqualArea()
  .scale(300)
  .center([-3.0026, 16.7666])
  .translate([480, 250]);
```

Geographic path generators

- A geographic path generator is a function that transforms **GeoJSON object** into an **SVG path string** (or into canvas element calls)
- geoPath()

```
let projection = d3.geoEquirectangular();

let geoGenerator = d3.geoPath()
  .projection(projection);

geoGenerator(geoJson); //e.g.returns a SVG path string "M464.01,154.09L491.15
```

Map Visualization Pipeline in D3

1. Prepare and load the GeoJSON data
2. Create the geoPath() and specify the projection configurations
3. Join a GeoJSON features array to SVG path elements
4. Update each path element's d attribute using the geographic path generator
5. Customize each feature (e.g., colors and interactions)

D3 Applications

- Commonly used setup
  - node.js + database (sql&nosql) + d3.js
  - flask (python based) + database (sql&nosql) + d3.js
  - observable
- For spatial / 3D data, three.js / vtk is recommended.

Flask

- Flask is a micro web framework written in Python. It is classified as a microframework because it does not require particular tools or libraries.
- Install Flask
  - pip install Flask
  - conda install -c anaconda flask

Takeaways

- be careful with D3 version iterations
- learn d3.js through online examples.
  - https://www.d3-graph-gallery.com/index.html (choose the v6 version)
  - https://observablehq.com/@d3/gallery
  - stackoverflow