CMPT 384 – Information Visualization

D3.js

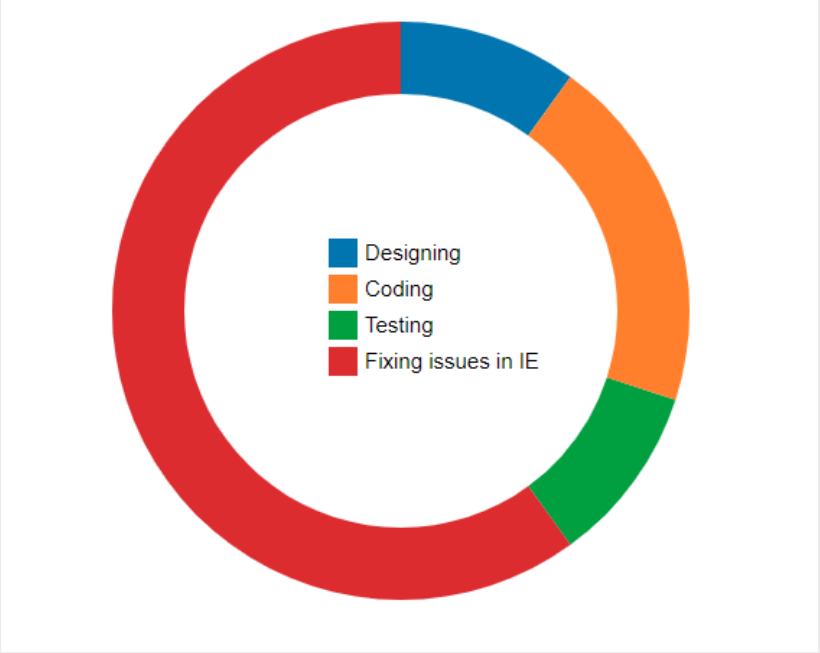
Lab 9

Course Instructor: Debajyoti Mondal

Lab Tutorial Instructor : Arman Heydari (arman.heydari@usask.ca)

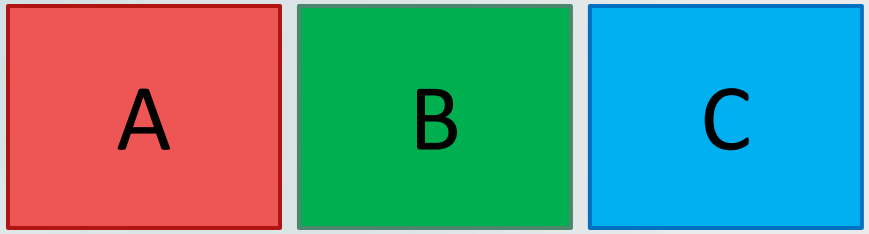
Agenda

- D3 Scale (Ordinal)
- D3 Generator (arc)
- D3 Color Scheme
- D3 Layout and Examples
- D3 Pie Chart
- Chart Legend
- D3 Force Layout
- D3 Drag function
- D3 Force Strength Panel



D3 Scales – Ordinal Scale

```
var color_bucket = d3.scaleOrdinal()
                     .domain(['A', 'B', 'C'])
                     .range(['Red', 'Green', 'Blue'])
```

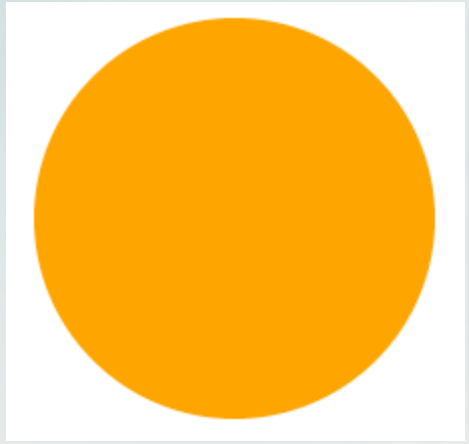



```
>> color_bucket('A')
>> 'Red'
>> color_bucket('C')
>> 'Blue'
```

D3 Generator (Arc)

```
d3.arc()
    .innerRadius(0)
    .outerRadius(100)
    .startAngle(0)
    .endAngle(Math.PI * 2)
```

Angle in Radian ; Math.PI = 180 degree

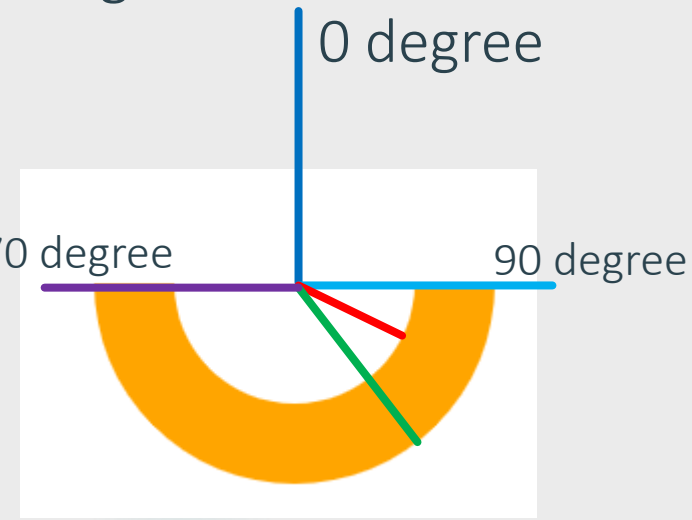


innerRadius = 60

outerRadius = 100

startAngle = Math.PI/2

endAngle = 3/2 * Math.PI



D3 Color Scheme

d3.**schemeCategory10** <>

An array of ten categorical colors represented as RGB hexadecimal strings.

d3.**schemeAccent** <>

An array of eight categorical colors represented as RGB hexadecimal strings.

d3.**schemeDark2** <>

An array of eight categorical colors represented as RGB hexadecimal strings.

d3.**schemePaired** <>

https://github.com/d3/d3-scale-chromatic/blob/master/README.md

D3 Layouts

- Layouts in D3 are special built in functions that transform your data points for advanced visualizations like tree maps and network graphs.
- Layouts however do not directly create charts.
- They only augment your datasets with additional values (visual variables) such as position and size.

Example Layout for Pie-Chart

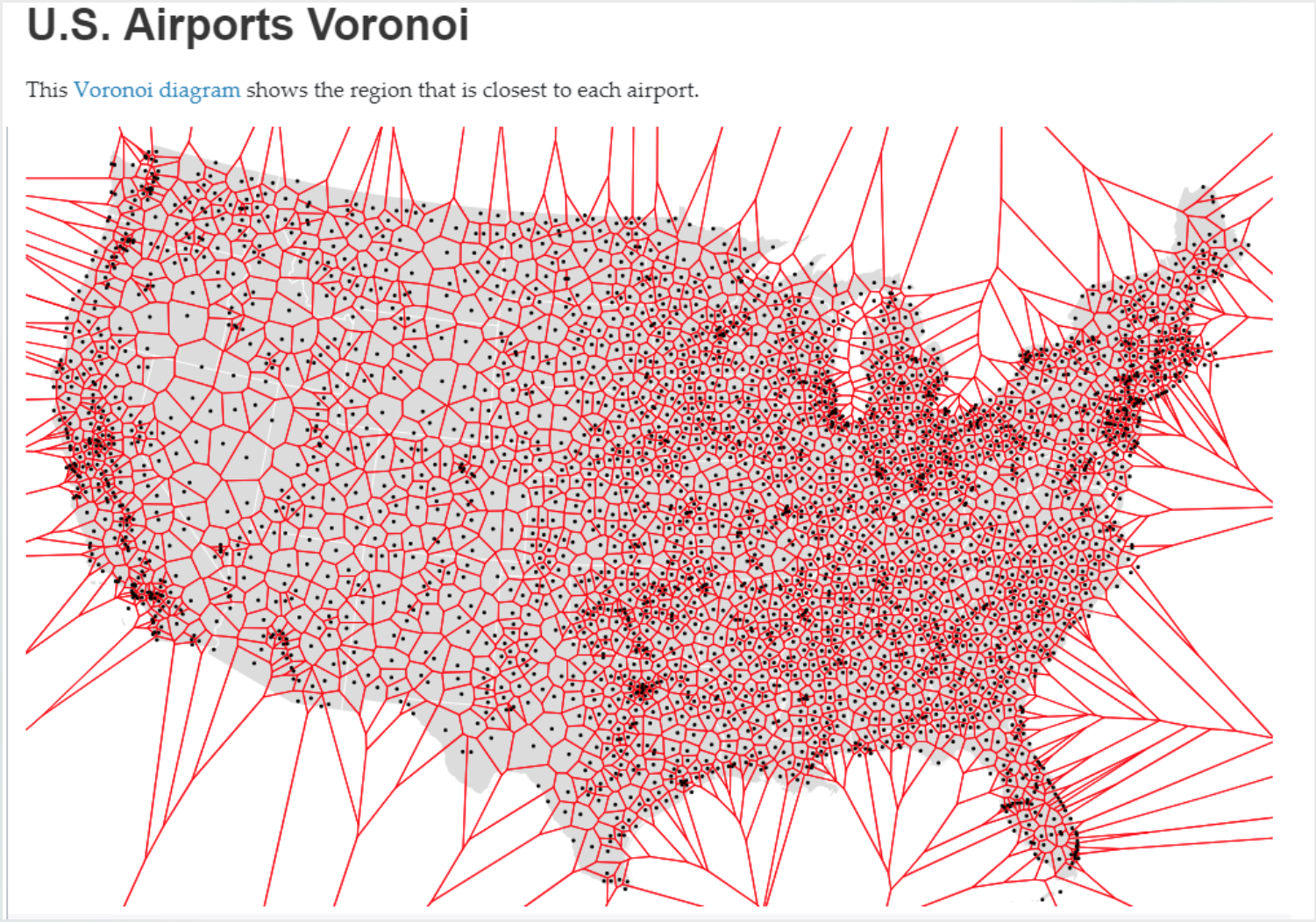

```
"[
 {
  "label": "Designing",
  "count": 20
 },
 {
  "label": "Coding",
  "count": 40
 },
 {
  "label": "Testing",
  "count": 20
 },
 {
  "label": "Fixing issues in IE",
  "count": 120
 }
]"
```


Initial Dataset


```
"[
 {
  "data": {
   "label": "Designing",
   "count": 20
  },
  "index": 0,
  "value": 20,
  "startAngle": 0,
  "endAngle": 0.6283185307179586,
  "padAngle": 0
 },
 {
  "data": {
   "label": "Coding",
   "count": 40
  },
  "index": 1,
  "value": 40,
  "startAngle": 0.6283185307179586,
  "endAngle": 1.8849555921538759,
  "padAngle": 0
 },
 {
  "data": {
   "label": "Testing",
   "count": 20
  },
  "index": 2,
  "value": 20,
  "startAngle": 1.8849555921538759,
  "endAngle": 2.5132741228718345,
  "padAngle": 0
 },
 {
  "data": {
   "label": "Fixing issues in IE",
   "count": 120
  },
  "index": 3,
  "value": 120,
  "startAngle": 2.5132741228718345,
  "endAngle": 6.283185307179586,
  "padAngle": 0
 }
]"
```

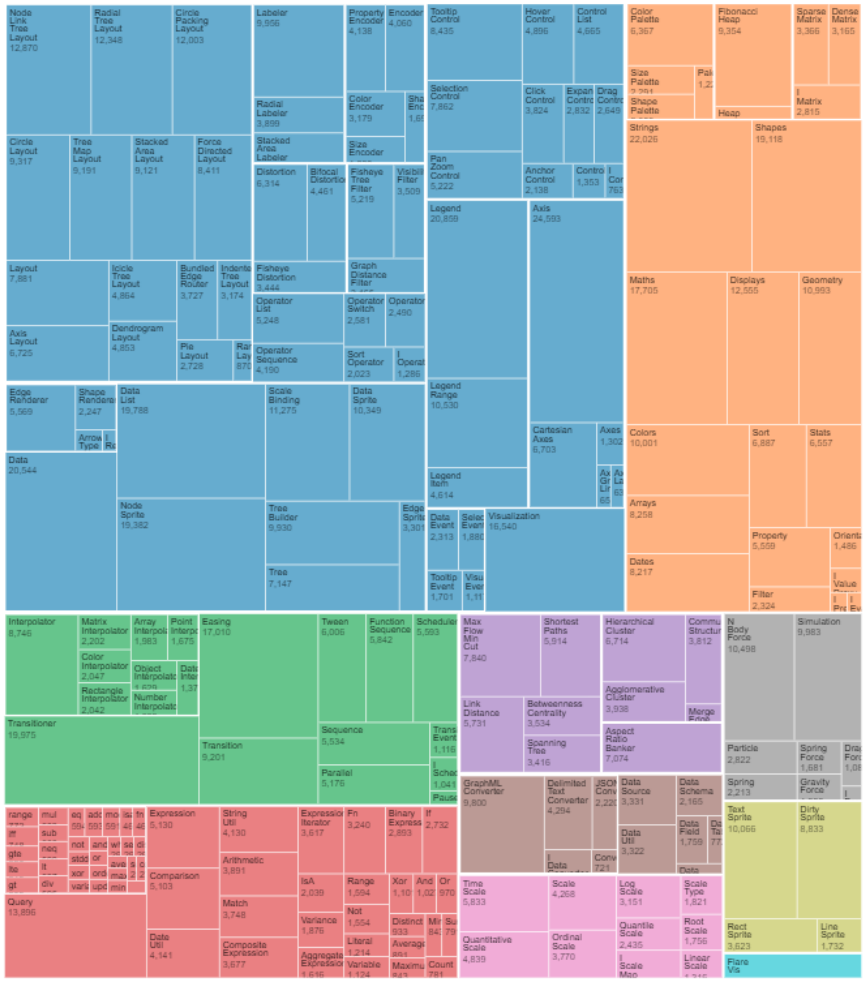

Dataset after applying layout

Layout output converted into graphics with generator

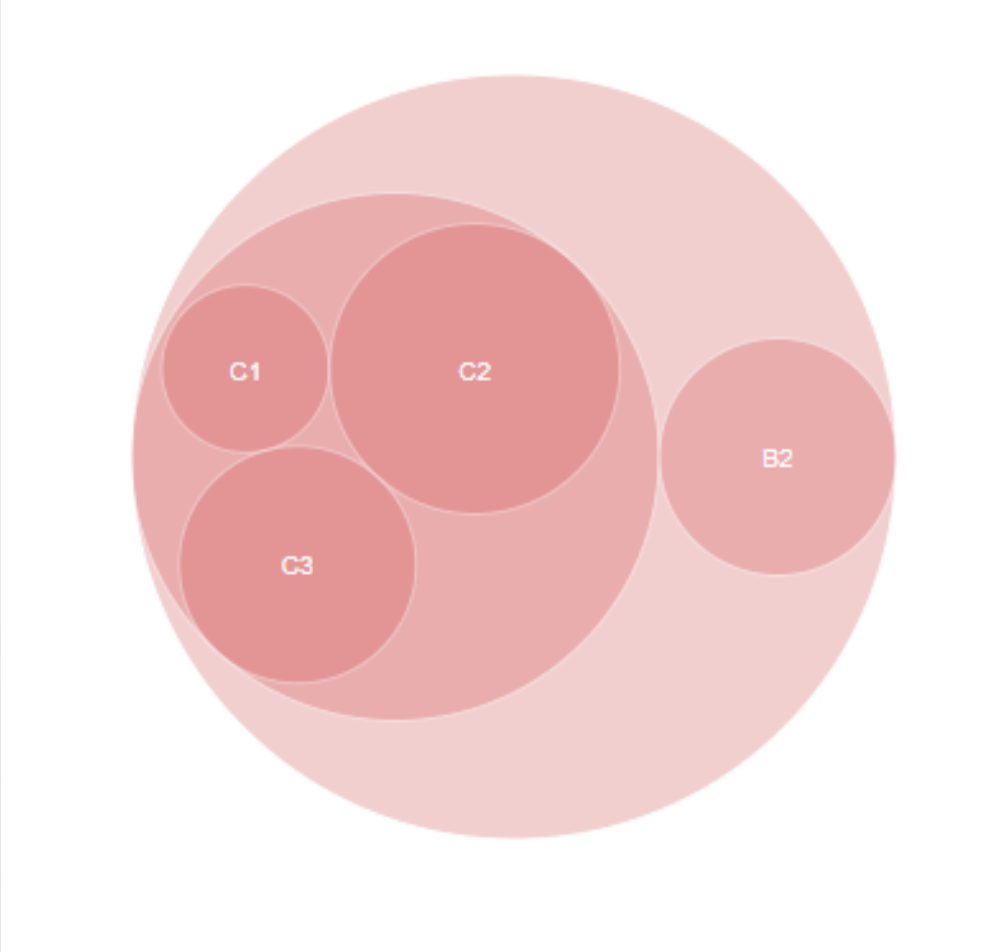


https://github.com/d3/d3-voronoi

The Flare visualization toolkit package hierarchy.



https://github.com/d3/d3-3.x-api-reference/blob/master/Treemap-Layout.md



https://github.com/d3/d3-3.x-api-reference/blob/master/Pack-Layout.md
Image reference - https://d3indepth.com/layouts/

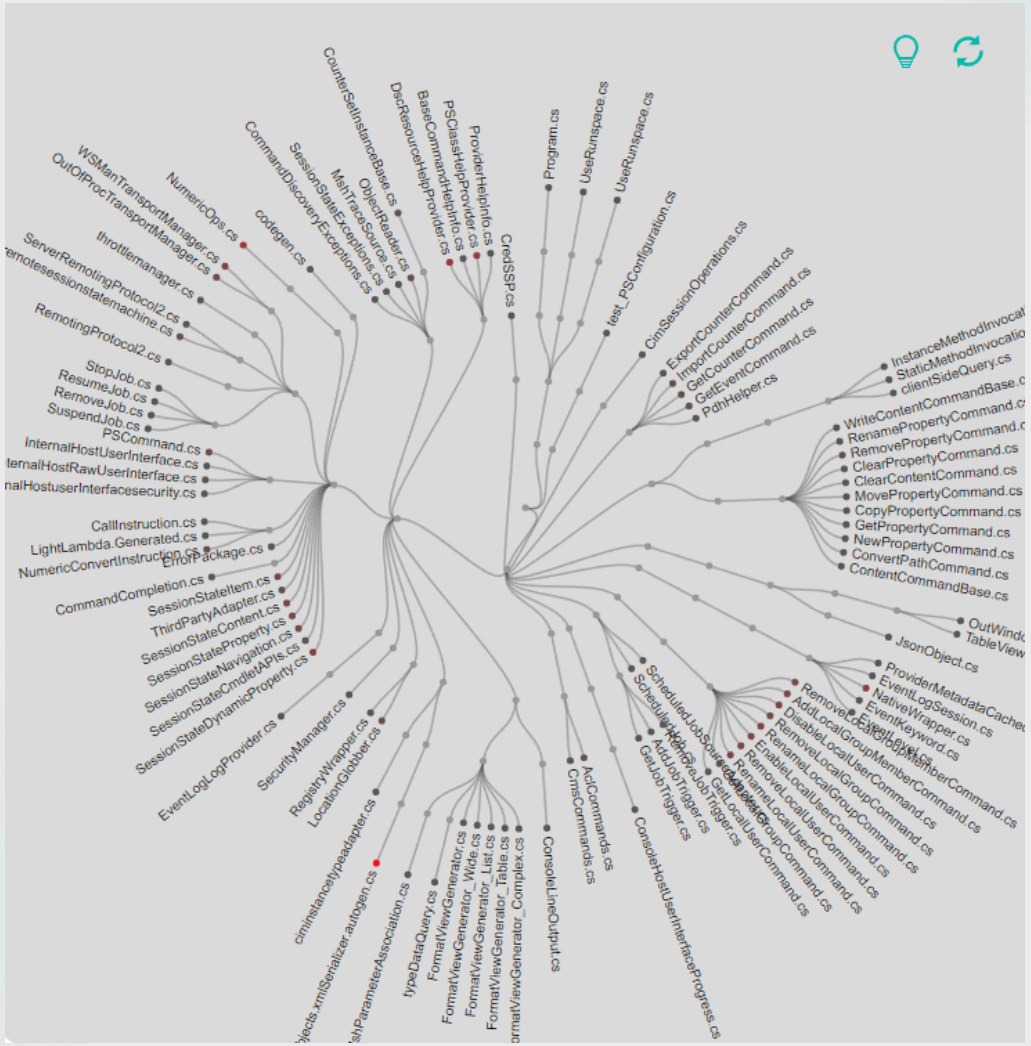
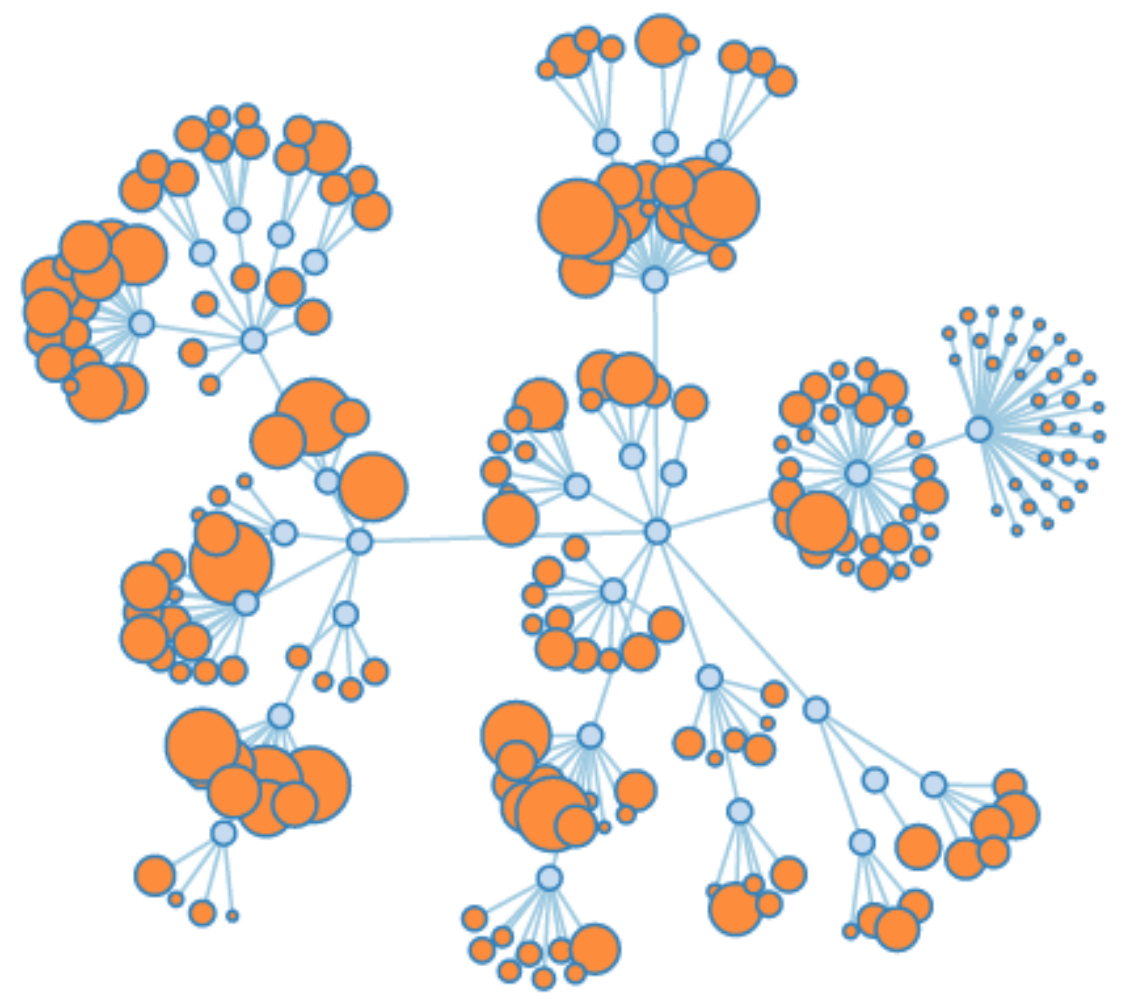


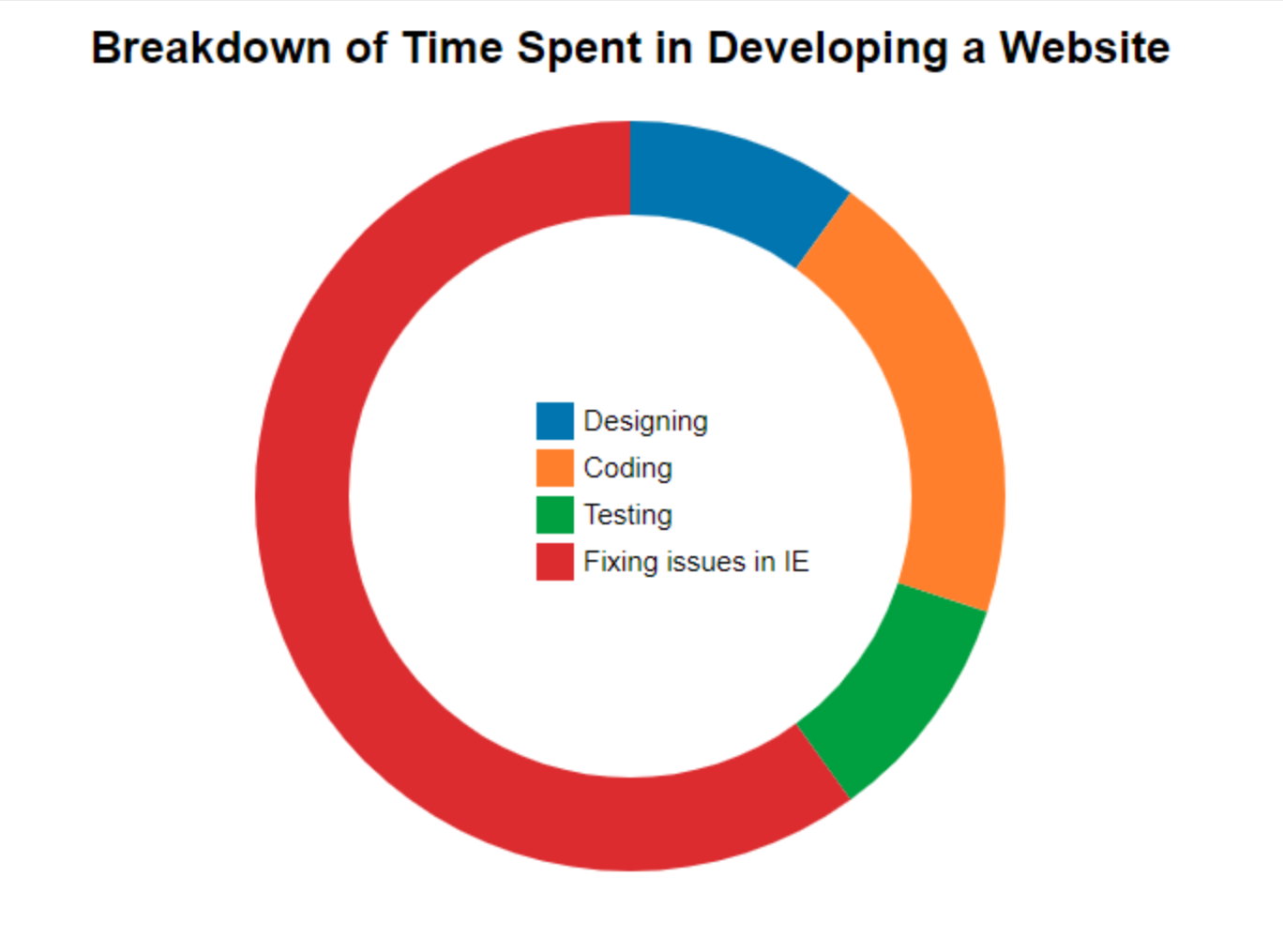
Image reference - https://clone-swarm.usask.ca/

Info - https://d3-wiki.readthedocs.io/zh_CN/master/Tree-Layout/



https://github.com/d3/d3-force

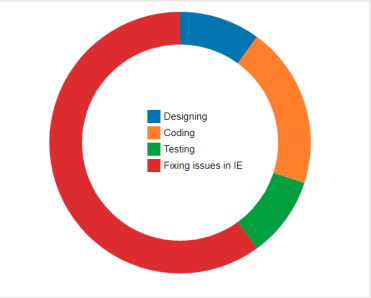
Example – Pie Chart



Example – Creating the Container

```
//Create SVG element
var graphicContainer = d3.select("body")
    .append("svg")
    .attr("width", w)
    .attr("height", h)
    // we create an inner graphic container and center it to the middle of the svg
    // because the pie is created from the center , directly creating the pie on the svg
    // would should only a quarter of the pie chart
    .append('g')
    .attr('transform', 'translate(' + w / 2 + ',' + h / 2 + ')');
```

Create the actual graphic inside an inner container and then center the inner container in the middle of the SVG.
This way we transform the center of the graphic to the center of the SVG instead of top left corner.

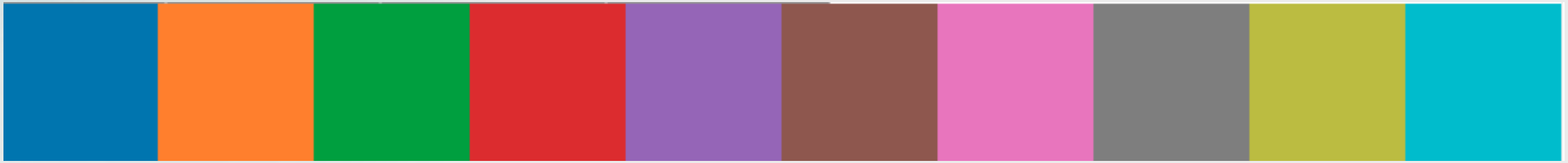


Create a color scale for the data

```
var data = [{
    'label': 'Designing',
    'count': 20
}, {
    'label': 'Coding',
    'count': 40
}, {
    'label': 'Testing',
    'count': 20
}, {
    'label': 'Fixing issues in IE',
    'count': 120
}];
```

```
// d3.schemeCategory10 gives an array of 10 colours
// we use an ordinal scale and map the labels to the colours
var colorScale = d3.scaleOrdinal()
    .domain(data.map(function(d) {
        return d.label;
    }))
    .range(d3.schemeCategory10.slice(0, data.length));
```

10 Basic Colors in D3 scheme Category 10



Configure Pie Layout and Arc Layout

```
// This function lets us configure the layout of our pie chart
var pieMapper = d3.pie()
    .sort(null)
    .value(function(d) {
        return d.count;
    });

var pieData = pieMapper(data);
```

Since we don't want any sorting the sort function is set to null .

The angles of the pie are calculated based on the count value and so an accessor function for the same must be passed in.

The pie chart needs to be visible on the screen without any clipping so we set the radius based on the minimum value.

Since we are creating a doughnut chart the inner radius is set to ¾ of the radius.

If we set the inner radius to 0 the chart is completely filled in. GIVE IT A TRY !

```
// The pie should be visible completely on the screen so we take
// the minimum value and take half of that.
var radius = Math.min(w, h) / 2;

// Function to decide how our arc must look
var arc = d3.arc()
    .innerRadius(radius * 0.75)
    .outerRadius(radius);
```

Generate Path Elements

```
// Convert the layout data into actual graphic elements
graphicContainer.selectAll('path')
    .data(pieData)
    .enter()
    .append('path')
    .attr('d', arc)
    .attr('fill', function(d) {
        return colorScale(d.data.label);
    });
```

Chart Legends

```
// Code to create Legends for our pie chart
// Each legend sits inside its own g element
// because each legend is a combination of a coloured rectangle and a text

var legendContainer = graphicContainer.selectAll('.legend')
    .data(data)
    .enter()
    .append('g')
    .attr('transform', function(d, i) {
        var offset = -radius / 4;
        var horizontalPostion = -radius / 4;
        var verticalPosition = i * 25 + offset;
        return "translate(" + horizontalPostion + "," + verticalPosition + ")";
    });
```


```
legendContainer.append('rect')
    .attr('width', 20)
    .attr('height', 20)
    .style('fill', function(d) {
        return colorScale(d.label);
    });
```

```
legendContainer.append('text')
    .attr('x', 25)
    .attr('y', 15)
    .text(function(d) {
        return d.label;
    });
```

D3-Force

- **D3-Force** is an inbuilt module for simulating physical forces on particles.
- The **D3- Force** module provides a layout for positioning visual elements based on a combination of different forces.

- For Example if we had a couple of circles , the types of forces could be :
- **Attraction Force** – All circles are clumped together
- **Collision Detection Force** – Circles do not overlap each other
- **Gravitational Forces** – All circles are forced towards a center of gravity
- **Linked Forces** – Circles are equidistant or fixed distance apart like in a network.

Steps in Setting up a Force Simulation

- Modify the data into the required format – an array of objects (nodes)
- Pass in the data to the force simulation layout , by calling the function **forceSimulation.**
- Define the various types of forces acting on the nodes using the different kinds of default force functions present in d3 like **forceManyBody** , **forceCenter** and **forceCollide**
- Update the attributes of the visual elements in the **tick function** as it is called each time the simulation iterates.

Built in Force Functions

- `forceCenter` (for setting the center of gravity of the system)
- `forceManyBody` (for making elements attract or repel one another)
- `forceCollide` (for preventing elements overlapping)
- `forceX` and `forceY` (for attracting elements to a given point)
- `forceLink` (for creating a fixed distance between connected elements)

Force functions are added to the simulation using `.force()` where the first argument is a user defined id and the second argument the force function:

```
simulation.force('charge', d3.forceManyBody())
```

https://d3indepth.com/force-layout/

Input Data


```
"[
 {
  "name": "Adam"
 },
 {
  "name": "Bob"
 },
 {
  "name": "Carrie"
 },
 {
  "name": "Donovan"
 },
 {
  "name": "Edward"
 },
 {
  "name": "Felicity"
 },
 {
  "name": "George"
 },
 {
  "name": "Hannah"
 },
 {
  "name": "Iris"
 },
 {
  "name": "Jerry"
 }
]"
```


Nodes


```
"[
 {
  "source": 0,
  "target": 1
 },
 {
  "source": 0,
  "target": 2
 },
 {
  "source": 0,
  "target": 3
 },
 {
  "source": 0,
  "target": 4
 },
 {
  "source": 1,
  "target": 5
 },
 {
  "source": 2,
  "target": 5
 },
 {
  "source": 2,
  "target": 5
 },
 {
  "source": 3,
  "target": 4
 },
 {
  "source": 5,
  "target": 8
 },
 {
  "source": 5,
  "target": 9
 },
 {
  "source": 6,
  "target": 7
 },
 {
  "source": 7,
  "target": 8
 },
 {
  "source": 8,
  "target": 9
 }
]"
```


Edges

Setup Force Simulation

```
//Initialize a simple force layout, using the nodes and edges in dataset
var force = d3.forceSimulation(dataset.nodes)
    .force("charge", d3.forceManyBody())
    .force("link", d3.forceLink(dataset.edges))
    .force("center", d3.forceCenter().x(w / 2).y(h / 2));
```

forceManybody has a default strength of -30 , negative indicating that the force is repulsive , so this force ensures that nodes are not overlapping one another.

The **forceLink** force holds the node togethers and ensures that nodes move relative to one another

The **forceCenter** force centers the nodes towards the center of the SVG.

Create Visual Elements

```
var colors = d3.scaleOrdinal(d3.schemeCategory10);

//Create SVG element
var svg = d3.select("body")
    .append("svg")
    .attr("width", w)
    .attr("height", h);
```

Note that the visual elements don't have any positions set as of yet.
They are only created and added to the webpage.

```
//Create edges as lines
var edges = svg.selectAll("line")
    .data(dataset.edges)
    .enter()
    .append("line");

//Create nodes as circles
var nodes = svg.selectAll("circle")
    .data(dataset.nodes)
    .enter()
    .append("circle")
    .attr("r", 10)
    .style("fill", function(d, i) {
        return colors(i);
    });
```

As the Simulation Iterates – Set the positions

```
//Every time the simulation "ticks", this will be called
force.on("tick", function() {

    edges.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;
    });

    nodes.attr("cx", function(d) {
        return d.x;
    })
    .attr("cy", function(d) {
        return d.y;
    });

});
```

Nodes are circles so their cx and cy are set thus positioning the circles based on the force.

Edges are lines drawn between two points (x1,y1) and (x2,y2)

As the simulation ticks these values are continuously updated creating the smooth effect of animation of nodes moving and colliding with one another.

Dragging Effects

```
nodes.call(d3.drag() //Define
    .on("start", dragStarted)
    .on("drag", dragging)
    .on("end", dragEnded));
```

```
//Define drag event functions
function dragStarted(d) {
    if (!d3.event.active) force.alphaTarget(0.3).restart();
    d.fx = d.x;
    d.fy = d.y;
}

function dragging(d) {
    d.fx = d3.event.x;
    d.fy = d3.event.y;
}

function dragEnded(d) {
    if (!d3.event.active) force.alphaTarget(0);
    d.fx = null;
    d.fy = null;
}
```

Other kinds of Layouts

Changing the initial set of forces that define the simulation changes the way in which the nodes are laid out and move.

For example in the dragging layout html file , modify the force simulation to the code snippet below -

```
//Initialize a simple force layout, using the nodes and edges in dataset
var force = d3.forceSimulation(dataset.nodes)
    .force("charge", d3.forceManyBody().strength(80))
    .force("collide", d3.forceCollide(10).strength(0.5))
    .force("center", d3.forceCenter().x(w / 2).y(h / 2));
```

The strength in the **forceManyBody** is positive so the nodes are pulled towards each other
While the **forceCollide** prevents the nodes from overlapping by giving a repelling force.

Other kinds of Layouts

We can also separate the nodes into two groups.

For this we can play around with the centering force where we force the **centerX** of all even numbered nodes to one X position 200 and all odd nodes to another position 700.

```
var force = d3.forceSimulation(dataset.nodes)
    .force("charge", d3.forceManyBody().strength(80))
    .force("collide", d3.forceCollide(10).strength(0.5))
    .force("center", d3.forceCenter().x(w / 2).y(h / 2))
    .force("x", d3.forceX().x(function(d, i) {
        return i % 2 == 0 ? 200 : 700;
    }))
```