D3 layouts

What is a layout?

Streamlined way to convert your data into a particular structure that d3 needs. (Similar to d3.nest, dateParse, etc.)

Kinds of chart that need a layout function

Pie charts

Hierarchies

Forces

Circle packing

Treemaps

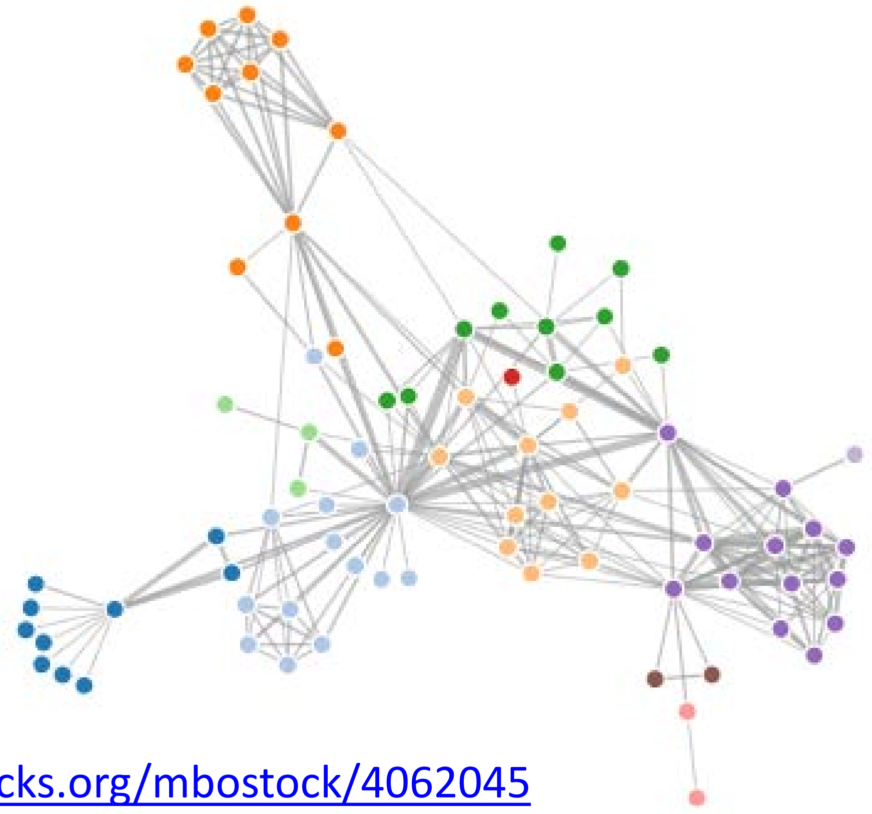
Dendrograms

Network diagrams

Making a force-based network diagram

Network diagrams are visualizations that show connections (links) between different entities (nodes).

Here, each dot represents a character in Les Miserables, and lines are drawn to show when two characters showed up in the same scene.

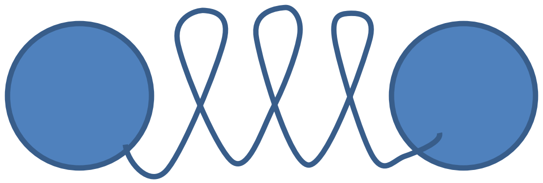


https://bl.ocks.org/mbostock/4062045

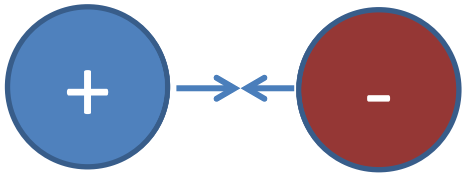
Forces

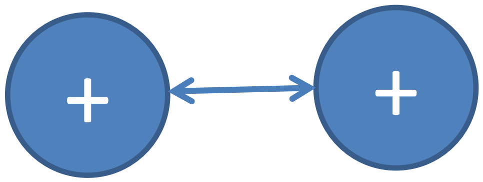
D3 uses the concept of forces (borrowed from physics) to optimize its network diagrams.

You can imagine forces being like invisible springs that connect different data points, and make them want to stay close to each other. Each link in the network represents an individual spring.



Each node in the network is given a “charge”, which makes them stay apart. Charges are similar to the poles of a magnet; like charges repel and opposite charges attract.



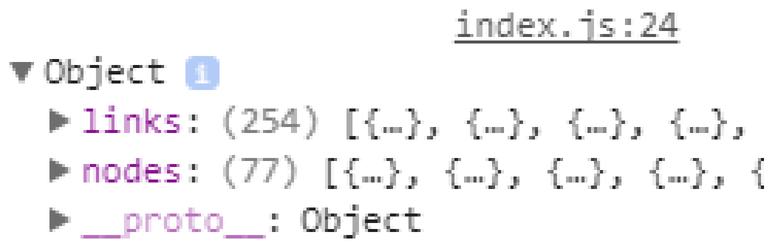


Setting up a forceSimulation( )

To program a network diagram, we first need to set up a forceSimulation, telling d3 what forces to use for the links (springs) and “charge” (+ or -) on the nodes, and where the center of gravity should be.

```
var simulation = d3 .forceSimulation ( )
            .force ( "link“ , d3 .forceLink ( ) .id ( function ( d )  { return d.id; }))
            .force( "charge“ , d3 .forceManyBody ( ) )
            .force ( "center“ ,  d3 .forceCenter ( width / 2, height / 2) );
```

The data is split up into two different objects: one that gives information about the nodes, and one that gives information about the links that connect them.



```
                               index.js:24
▼ Object i
  ▶ links: (254) [{…}, {…}, {…}, {…},
  ▶ nodes: (77) [{…}, {…}, {…}, {…}, {
  ▶ __proto__: Object
```

Links list source and target, and nodes have x and y coordinates.

Binding to the data

Once the data is loaded, we will make two d3 selections (one for links, one for nodes), and bind the data to each one.

```
var link = svg.append ( "g“ )
            .attr ( "class", "links“ )
            .selectAll ( "line“ )
            .data ( graph.links )
            .enter ( )
            .append ( "line“ )
            .attr ( 'stroke', 'gainsboro‘ )
            .attr ( "stroke-width", function ( d ) { return Math .sqrt ( d .value ); } );

var node = svg .append ( "g“ )
            .attr ( "class“ , "nodes“ )
            .selectAll ( "circle“ )
            .data ( graph .nodes )
            .enter ( ) .append( "circle“ )
            .attr ( "r", 5 )
            .attr ( "fill“ , function ( d ) { return color ( d .group ); } ) ;
```

Geometry minimization

Force simulations work by doing geometry minimizations, which means that they look at all of the nodes and links and the forces connecting them and make small adjustments many, many times, until the spacing between nodes is as good as it can possibly get.

This means that the function that determines the node and link distance must run over and over again. We've done something similar to this before, when we used the .setInterval function in Javascript.

Now, we want to write a function that will run every time an imaginary clock ticks.

```
function ticked ( ) {

}
```

Setting up the tick function

The ticked function is basically an update function for the drawing, that tells it what to do with all of the nodes and links when the simulation changes their position.

The d3 simulation edits the x and y coordinates of each node with each tick of the clock. Because the links “know” which nodes they are connected to, they can use this source and target information to look up the right coordinates for the beginning and end of the line.

```
function ticked() {

        link .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; } );

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

Telling the simulation what to do

Now, we need to tell the simulation what to use for links and nodes, and to run when the clock ticks.

Simulations have a special “on” function called “tick”, which we can use to call the ticked function that we just wrote.

```
simulation .nodes ( graph .nodes )
            .on ( "tick", ticked );

simulation.force ( “link“ )
            .links ( graph.links );
```

Adding titles and drag behavior

This node network looks interesting, but it isn’t terribly useful; you can’t tell what any of the nodes are, so you can’t tell what it actually means.

Let’s give the nodes a label that shows up when you hover over them:

```
node .append ( "title“ )
        .text ( function ( d )  { return d.id; } );
```

We can also allow define drag behaviors that allow the user to pick up a node and drag it around.

Defining drag behavior

Each drag behavior needs to have a function associated with it that tells d3 what happens when the user moves a node.

```
function dragstarted ( d ) {
        if (!d3.event.active)    simulation .alphaTarget ( 0.3 ) .restart ( );
        d .fx = d .x;
        d .fy = d .y;
}

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

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

Adding drag behavior

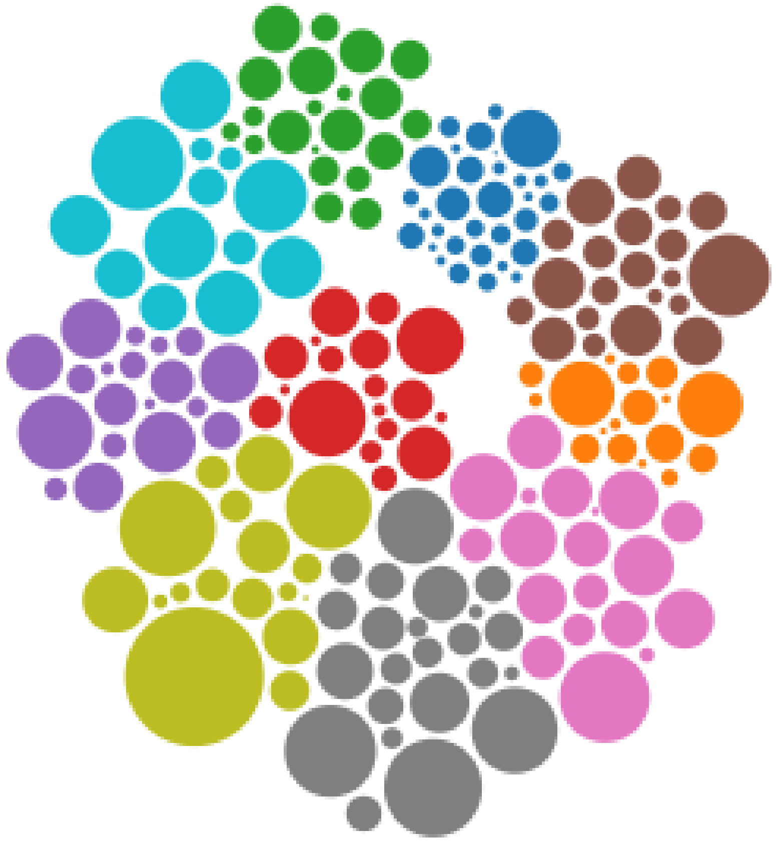
Add the drag behaviors to the individual nodes, using .on:

(add to var = node)

```
           .call ( d3.drag ( )
                     .on ( "start",  dragstarted )
                     .on ( "drag", dragged )
                     .on ( "end",  dragended ) );
```

Making a clustered bubble chart

Clustered bubbles are a very popular way to give an overview of a dataset. In this visualization, each group of circles is attracted to its own individual center, using a custom gravity function.



Getting the data

For our example, we will work with data for the 100 biggest banks in the world

Data is from here:
http://www.snl.com/web/client?auth=inherit#news/article?id=40223698&cdid=A-40223698-11568

I couldn't find a download site to get this information in .csv form, but I found a website that published it as an html table.

https://www.relbanks.com/worlds-top-banks/assets

Normally, you would be able to copy and paste this into Excel to use it. But that didn't work. Instead, I found an HTML table converter, and viewed the page source to get the table content. Saving the HTML table to a text file, uploading to this website, and converting to .csv allowed me to download the data contents as a .csv file.

https://conversiontools.io/convert_html_to_csv/

Finding a list of countries in the data

I want to make a bubble map with one cluster of bubbles for each country, and give each one of them a different color. To do that, I needed to find out how many countries were in my dataset. Counting them was tedious, so instead I searched Stack Overflow:

https://stackoverflow.com/questions/11246758/how-to-get-unique-values-in-an-array

In there, I found an array technique I'd never heard of, that returns a list of all the unique items in an array.

I started out by creating an array of all the country names using .map:

```
countryList  =  banks .map ( function ( d )  {  return d.Country  } );
```

Getting out unique country names

Then, I used the Stack Overflow example to get a list of the unique items in the array.

```
uniqueList  =   countryList .filter ( function ( d,  i,  a ) {
                return a .indexOf ( d ) = =  i ;
} );
```

This tells me that there are 25 unique countries in my dataset. I want one color for each, but d3's automatic color generation list only goes up to 25 items.

I searched again, and found another Stack Overflow post in which someone mentioned a tool that he'd developed for generating categorical colors with maximal perceptual separation (makes them easy to tell apart):

http://jnnnnn.github.io/category-colors-constrained.html

Use that tool, let it run to 25, and copy and paste the hex codes for use in d3.

Setting up a color scale

Plug the hex codes into a color scale:

```
var colorScale = d3 .scaleOrdinal ( )
                    .range ( [    *put the list of hex codes here*   ]);
```

And, once the data has loaded, set the domain:

```
colorScale .domain ( uniqueList );
```

Now, we have a color connected to each country name in our dataset.

Figure out the data structure

Pause to look at example code. Work through to figure out what‘s going on with the data structure. Copy and paste everything into a temporary folder (_translateBostock) to make sure that there are no bugs.

This code generates a random dataset, similar to what we did for our first circle drawings at the beginning of the semester. We need to figure out which pieces of it are important, so that we can make similar structures in our code.

Comment out everything but the cluster array, and look at it.

Then, uncomment the nodes piece and look at that, too.

d3.range (n) (where n = 200) creates a list of integers from 1 - 200.

For each thing in that list:

- make a random value i that assigns it to a cluster.
- calculate a random radius, smaller than a maximum value.
- make a parameter d that contains information about both the cluster and the radius of the circle.

Figure out the data structure

Look inside the nodes list, and verify that each one has a cluster number and a radius for the node.

This is what we need in our data. Currently, we have one item for each value in our spreadsheet, which will become a node in our diagram. To get it into the right format, we need two things:

- A cluster id for each node, based on the country name
- A radius for the circle that we want to draw, based on the bank size

The second part is easy; just set up a new scale for the radius

```
var radiusScale  =  d3 .scaleLinear ( ) .range ( [ 0 , 30 ] );
```

and connect it to the bank assets

```
radiusScale .domain ( [ 0, d3 .max (
            banks .map ( function ( d )  { return d.assets } ) ) ] );
```

Assigning clusters

Now, we need to give each node a cluster id to belong to, so that we can assign colors to each cluster. We want one cluster for each country in the dataset, and we want to be able to look up which cluster each node belongs to.

Again, we've done this before.

```
var clusterLookup  =  d3 .map ( );

uniqueList .forEach ( function ( d , i ) {
        clusterLookup .set ( d ,  i );
} );
```

Check that it works:

```
console .log ( clusterLookup .get ( "China“ ) );
```

Saving cluster assignment

Now that we can look up a cluster number for each node, save that information in the nodes dataset, so that it becomes a part of the data bound to that node.

```
banks .forEach ( function ( d )  {
       d .cluster = clusterLookup .get ( d.country );
} )
```

Setting up the force layout

Bubble packing doesn’t have links, so instead we need to use a collision function to keep the circles from overlapping.

```
var forceCollide  =  d3 .forceCollide ( )
        .radius (  function ( d )  {  return radiusScale ( d .assets )  + 1.5; } )
        .iterations ( 1 );

var force = d3 .forceSimulation ( )
        .nodes ( banks )
        .force ( "center", d3 .forceCenter ( ) )
        .force ( "collide", forceCollide )
        .force ( "gravity", d3 .forceManyBody ( 30 ) )
        .force ( "x", d3 .forceX ( ) .strength ( .7 ) )
        .force ( "y", d3 .forceY ( ) .strength ( .7 ) );
```

Actually draw the circles

```
var circle = svg .selectAll ( "circle“ )
            .data ( banks )
            .enter ( )
            .append ( "circle“ )
            .attr ( "r", function ( d ) { return radiusScale ( d.assets ) } )
            .style ( "fill", function ( d ) { return colorScale ( d.cluster ) ; } )
```

Notice the errors in the console; showing up in the radius function. Console.log to find out why.

Reformat numbers in Excel.

Add a title so we can see what the bubbles represent

```
circle .append ( 'title‘ ) .text ( function ( d )  { return d .bank   +  '; ‘  + d .country } );
```

Set up the force layout

Create a tick function to adjust the position of the nodes:

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

Add it to the forceSimulation using .on():

```
.on("tick", tick);
```

Getting the circles to cluster

Switch to clusteredBubbleLayout

To make the bubbles group into clusters, we need a way of keeping track of other bubbles in the same cluster, and saving the position of the cluster itself.

To keep things simple, use the biggest circle in the cluster as the “center” that everyone else follows around .

We also need a list of clusters, so that we can perform a geometry minimization on them using the forceSimulation.

Use uniqueList to populate this list:

```
var clusterList = [ ];

uniqueList .forEach ( function ( d , i ) {
	clusterLookup .set ( d, i );
	clusterList .push ( { cluster : i } );
} );
```

Finding the biggest node

If we want to center the cluster on the biggest node, we need to first figure out which one is the biggest.

For each cluster in the list, go through the banks array and find all of the nodes that belong in that cluster.

Look for their max value, and save a new property called maxAsset inside each cluster that records the value of the biggest node.

```
clusterList .forEach ( function ( d ) {
        d .maxAsset   =  d3 .max ( banks .map ( function ( e ) {
                if ( e .cluster  = =  d .cluster ) {
                                return + e .assets
                 }
                 else { return 0; }
            }

        ) );
} );
```

Writing a new force function

Now, we need to write a custom force function to teach the forceSimulation how to create clusters.

Look at the _translateBostock code again to see what their force function does.

We need a forceCluster function that takes an alpha value and updates the position of all of our nodes and clusters.

```
function forceCluster (alpha) {

  //look at all the nodes in the system
  //compare them to the biggest node in their cluster
  //move them closer to the biggest node

}
```

This function will run over and over and over again, until the value of alpha gets very small.

Writing the force function, cont’d

Each time the force function runs, we want to go through the entire banks array and move each node closer to other nodes in its cluster. Force simulations create x,y, vx and vy properties for each node automatically, and use them to keep track of the positions of the nodes when the simulation runs.

Start by writing a for loop that goes through the whole array.

Save each node in a temporary variable, and use it to look up its cluster in the clusterList (remember that the cluster number stored in each node is the array index of the cluster in the cluster list).

(++i returns the value of i after the increment has been applied.)

```
for ( var i = 0 , n = banks .length ;  i < n ;  ++ i ) {

        var node = banks [ i ];
        var cluster = clusterList [ node .cluster ];
        var k = alpha * 2 ;
}
```

Writing the force function, cont'd

Because all of the nodes moved the last time that the force function ran, we need to update the position of the clusters using the biggest node:

Check to see if the current node is the biggest one in the cluster, using the x, y, vx and vy coordinates. If it is the biggest, use it to re-set the cluster position.

```
if (node.assets == cluster.maxAsset){
        cluster.x = node.x;
        cluster.vx = node.vx;
        cluster.y = node.y;
        cluster.vy = node.vy;
}
```

Then, see how far the current node is from the cluster center, and subtract this difference to move the node closer to the center of the cluster.

```
node.vx -= (node.x - cluster.x) * k;
node.vy -= (node.y - cluster.y) * k;
```

Apply the force function to the simulation

Now that we have a custom force function, we can apply it to the layout as an additional force that gets added into the system behavior:

```
var force = d3 .forceSimulation ( )
        .nodes ( banks )
        .force ( "center", d3 .forceCenter ( ) )
        .force ( "collide", forceCollide )
        .force ( "cluster", forceCluster )
        .force ( "gravity", d3 .forceManyBody ( 30 ) )
        .force ( "x", d3 .forceX ( ) .strength ( .7 ) )
        .force ( "y", d3 .forceY ( ) .strength ( .7 ) );
```

Adjust values of k to get different cluster separation; smaller values have less perturbation and less separation of clusters, with larger values, the clusters may start to wander off. (try .2 and 5)

Tweaking the simulation parameters

Getting a force simulation to behave how you want it to takes a lot of trial and error.

Adjust values of k to get different cluster separation; smaller values have less perturbation and less separation of clusters, with larger values, the clusters may start to wander off. (try .2 and 5)

Alpha and velocity decay can also affect the speed of resolution for the simulation:
.alphaDecay (.07)
.velocityDecay (.9)

Velocity decay ranges between 0 and 1; low values take longer to resolve, large values settle out more quickly.

Play around with this simulator to get a better idea of how changing parameters modifies the effect:
https://bl.ocks.org/steveharoz/8c3e2524079a8c440df60c1ab72b5d03