

# Lab 9: D3 Layouts

Wijaya, Mario edited this page 3 days ago · 6 revisions

This lab is scheduled for **Apr. 18th (Wednesday)**.

► Pages 18

## Learning Objectives

After completing this lab you will be able to:

- Understand the concept of how D3 layouts re-format data for advanced visualizations.
- Understand how D3 simulations work
- Be able to use D3 layouts for advanced visualizations (e.g., graphs, trees or stacked charts)

## Submission

Submit one or two screenshot(s) of your network layout (with hovering and dragging if you have implemented them). You can also include code in addition to the graph in your screenshot to show your work.

## Prerequisites

- Update your *starter code repository* using one of the following methods:
  - i. From the GitHub Desktop app click `sync` on the top right
  - ii. Open a command line prompt. Navigate to the repository directory, for example `cd ~\Development\CS4460-Spring2018\Labs` and run command `git pull`.
- You have read the *Getting Started* section of [Force Directed Graphs with d3 v4](#) from puzzlr

## Additional Reading

- [Chapter 11. Layouts](#) in *Interactive Data Visualization for the Web* by Scott Murray

### Example Blocks

- [D3 v4 - force layout](#) by shimizu
- [Treemap](#) by Mike Bostock
- [Simple Word Tree](#) by Elijah Meeks
- [Pie Chart](#) by Mike Bostock
- [Force Directed Radial Layout](#) by Jim Vallandingham
- [Force Layout with Multiple Foci](#) by Jim Vallandingham
- [Sunburst Tutorial](#) by David Richards
- [Sequences sunburst](#) by Kerry Rodden
- [Bubble Chart](#) by Mike Bostock
- [Circle-Packing](#) by Mike Bostock
- [Zoomable Circle Packing](#) by Mike Bostock
- [Stacked Area Chart](#) by Mike Bostock

### D3 API

- [d3-force layouts](#) - networks, hierarchies, bubble charts, beeswarm plots and physics engines
- [d3-hierarchy](#) - clusters, trees, treemaps, partitions, and packing algorithms
- [d3-chord](#) - circular chord layout

[Lab 0: HTML & CSS](#)

[Lab 1: Javascript 101](#)

[Lab 2: SVG](#)

[Lab 3: Intro to D3 \(Pre Lab\)](#)

[Lab 3: Intro to D3 \(Activities\)](#)

[Lab 4: D3 Chart Types & Scales \(Pre Lab\)](#)

[Lab 4: D3 Chart Types & Scales \(Activities\)](#)

[Lab 5: D3 Selections & Grouping \(Pre-Lab\)](#)

[Lab 5: D3 Selections & Grouping \(Activities\)](#)

[Lab 6: D3 Enter, Update & Exit \(Pre-Lab\)](#)

[Lab 6: D3 Enter, Update & Exit \(Activities\)](#)

[Lab 7: Interaction & Transition 1 \(Pre Lab\)](#)

[Lab 7: Interaction & Transition 1 \(Activities\)](#)

[Lab 8: Interaction and Transition 2](#)

[Lab 9: D3 Layouts](#)

[Lab 10: D3 Maps](#)

### Clone this wiki locally

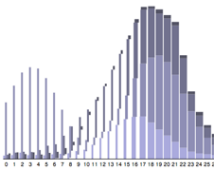
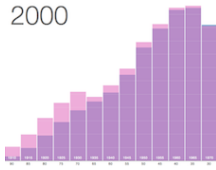
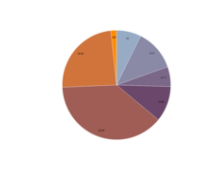

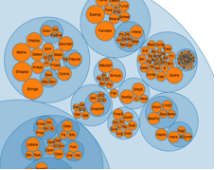
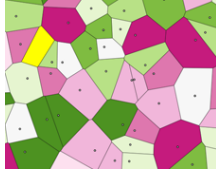




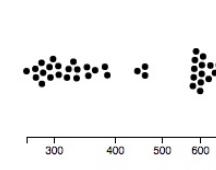

<https://github.gatech.edu>



In this lab you will learn how to use D3 layout methods to implement more complex visualizations (as compared to scatterplots, line charts and bar charts).

The D3 layout methods have no direct visual output. Rather, D3 layouts take data that you provide and re-map or otherwise transform it, thereby generating new data that is more convenient for a specific task. Half the battle when creating these visualizations is getting the dataset into the correct form.

D3 offers a number of different layouts, each with distinct characteristics.

Stack Example	Histogram Example	Pie Example	Treemap Example
			
Pack Example	Partition Example	Tree Example	Networks Example
			
Hierarchies Example	Bubble Example	Beeswarm Example	Chord Example
			

All above example images come from the [D3 Gallery Page](#)

Each layout may have distinct features not shared by other layouts, so make sure to consult the D3 documentation. A lot of these layouts are very complex and with only the documentation in hand, it is tough to know how to implement them. Finding a good tutorial is key for creating these visualizations with your own data.

Note on D3 v4: In the older versions of D3 (i.e. before v4) the layouts were contained in one module `d3.layout` - now they are spread out across different namespaces to support modularity (e.g. `d3.histogram`, `d3.simulation`)

In today's lab we will cover the force-directed graph layout. The above sections contain a great starting point for examples and tutorials of D3 layouts to consider for your own visualizations.

**Force simulation**

The force layout or force-directed layout is typically used to create network graphs, or also called node-link-diagrams. It consists of nodes and edges (links connecting the nodes) and helps us to visualize networks and the relationships between objects (e.g., social networks, relationships between politicians, business relations).

The name force-directed layout comes from the fact that these layouts use simulations of physical forces to arrange elements on the screen. The goal is to reduce the number of crossing edges, so that is easy for the user to analyze the whole network.

## Activity - Creating an interactive node-link graph

Reminder: Start an http server for this lab's directory. From command line call `python -m SimpleHTTPServer 8080` (for Python 2) or `python -m http.server 8080` (for Python 3).

During today's activity you will use a force simulation to create connections between characters from the story "Les Miserables". If you are not familiar with the story that's fine, but I would recommend seeing the play if you have the chance (the movie with Hugh Jackman and Anne Hathaway is meh).

You will be working with the `les_miserables.json` dataset. *This dataset is not a CSV but a JSON file.* JSON datasets are not as uniform as a CSV, the makeup of the dataset is all up to its creator. In this case the JSON data is somewhat tame. The file consists of two lists of objects: `links` and `nodes`.

Here's a snippet of the `nodes` list:

```
{ "id": "Myriel", "group": 1 },
{ "id": "Cosette", "group": 5 },
{ "id": "Valjean", "group": 2 }
```

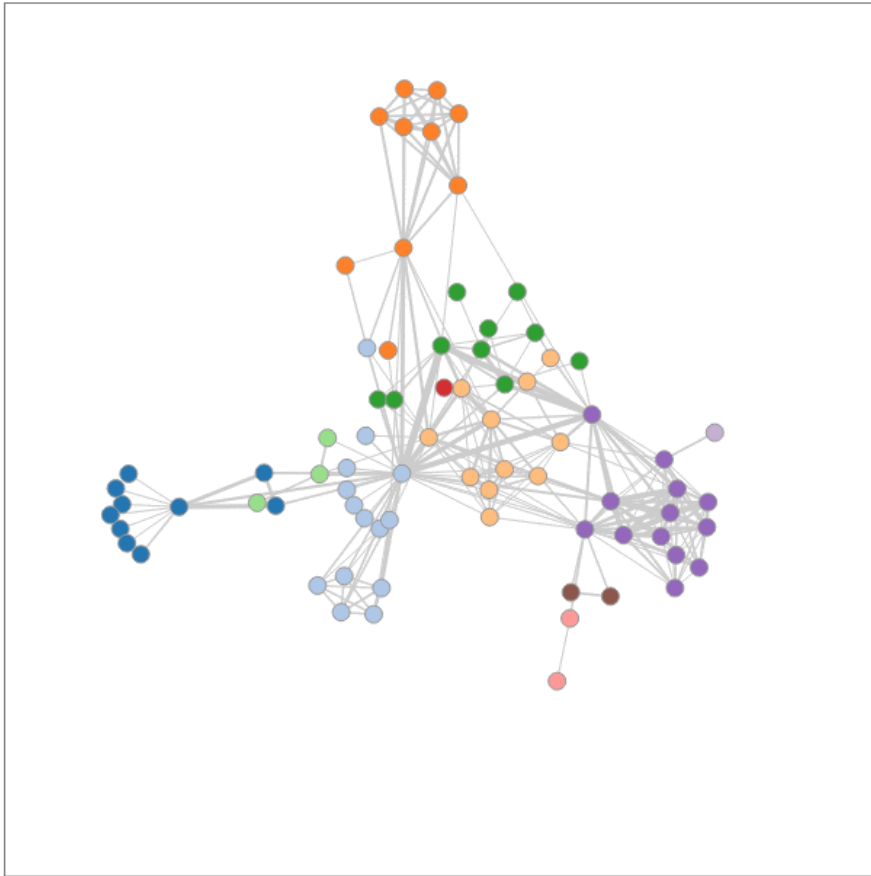
The `id` is the character's name and the `group` refers to the character's cohorts.

And the `links` list:

```
{ "source": "Valjean", "target": "Myriel", "value": 5 },
{ "source": "Cosette", "target": "Valjean", "value": 31 },
{ "source": "Javert", "target": "Valjean", "value": 17 }
```

This format of `source` and `target` is a very important part of creating a node-link graph. The `source` and `target` properties are reserved key-words that D3 uses to create a force-directed graph. Also the `value` is added to constrain the magnitude of that relationship between characters.

In this lab you will be creating the following interactive node-link graph:



For this activity you will use a lot of the concepts that you have learned so far to data-join SVG elements and update their position. We will build upon that by adding in the force-direct simulation to layout this network of characters. And finally we will add interaction to the graph with dragging capabilities.

### 1. Inspect the data

Take a second to log the loaded `dataset` and get a sense for the elements that make up the `links` and `nodes` lists.

### 2. Declare the force-simulation

Create the D3 force simulation. This simulation will be called on later to place the nodes in the canvas based on the force-directed layout. The layout is defined with the `.force('charge', d3.forceManyBody())` which defines how the linked nodes interact, and the property `.force('link', d3.forceLink().id(function(d) { return d.id;}))` defines how the links are connected within the graph. Declare the `simulation` object somewhere in your code so that you can use it within the `d3.json` callback:

```
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));
```

### 3. Append links

Notice that we have already created a `linkg` group on the SVG. This group element will hold all of the links for the chart. We append it first because we want the links to show up below the nodes.

Next we want to use D3's data-join pattern that we have been using to append `line` elements for each link in the dataset. We do this by just using the `Enter` selection:

```

var linkEnter = linkG.selectAll('.link')
  .data(network.links)
  .enter()
  .append('line')
  .attr('class', 'link')
  .attr('stroke-width', function(d) {
    return linkScale(d.value);
  });

```

Notice that we did not set the `x1`, `y1`, `x2`, `y2` properties of the lines. That is because we want to update their position with the output from the force-directed simulation.

#### 4. Append nodes

Again, notice that we have already created a `nodeG` group on the SVG. This group element will hold all of the nodes for the chart and appear on top of the links.

We append all of the `circle` node elements in a similar manner but this time with the `nodes` list as the data:

```

var nodeEnter = nodeG.selectAll('.node')
  .data(network.nodes)
  .enter()
  .append('circle')
  .attr('class', 'node')
  .attr('r', 6)
  .style('fill', function(d) {
    return colorScale(d.group);
  });

```

Once again, notice that we did not set the `cx` or `cy` properties of the nodes. This will be updated from the force-directed simulation.

#### 5. Update on simulation tick

Each iteration of the force-layout is called a "tick", similar to iterations in other physics simulations. With each tick, the force layout adjusts the x/y coordinates for each node and edge.

To make the simulation run properly we need to point the simulation to the data. We need to define the nodes and links that make up the network. And we need to define an event listener to be called every time the simulation "tick"s.

We add the following code directly after our `linkEnter` and `nodeEnter` declarations:

```

simulation
  .nodes(network.nodes)
  .on('tick', tickSimulation);

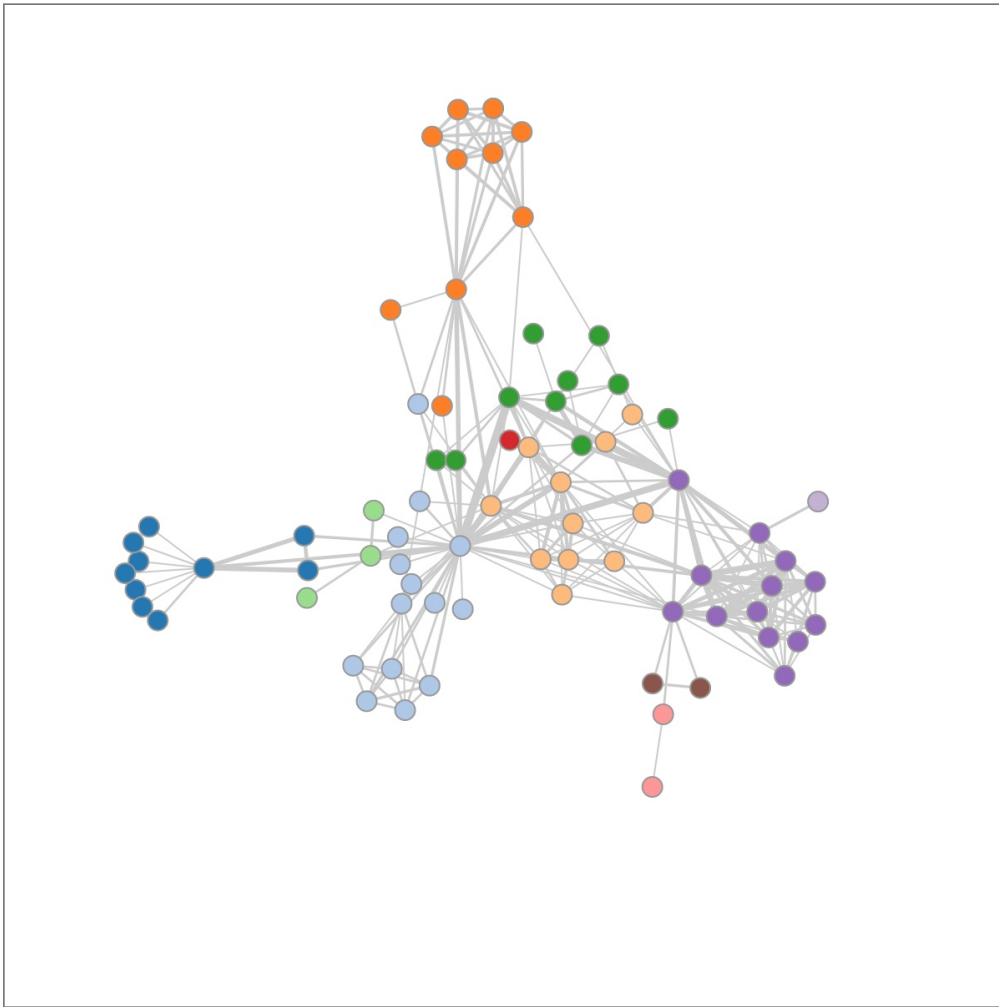
simulation
  .force('link')
  .links(network.links);

function tickSimulation() {
  linkEnter
    .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;});

  nodeEnter
    .attr('cx', function(d) { return d.x;})
    .attr('cy', function(d) { return d.y;});
}

```

Now when you refresh your browser you should see the following node-link diagram:



## 6. Implement draggable nodes

Next to add a little bit more interaction to our chart we can very easily add dragging. D3 provides a handful of interaction modules, we have already seen the d3-brush module last week. This week we will cover the d3-drag module.

The d3-drag supports events for clicking and dragging elements. D3 handles all of the nuts and bolts for taking mousedown, mousemove, and mouseup events and translating them into start, drag, and end events.

To declare a drag interaction module we point to event listeners for each of the 3 drag events:

```
var drag = d3.drag()
  .on('start', dragstarted)
  .on('drag', dragged)
  .on('end', dragended);
```

We also need to call the drag event on our circle node objects:

```
nodeEnter.call(drag);
```

Finally we need to add these 3 event listeners to our code. You can add them anywhere you want, but its standard practice to add event listeners to the end of the JS file:

```
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;
}
```

You might be wondering what exactly do these functions do. The if-statements at the beginning of the `dragstarted` and `dragended` functions are relaxing or re-engaging the simulation to allow the user's drag event to take precedence in defining the position of the node. The `restart()` method can be used to “reheat” the simulation during interaction, such as when dragging a node.

Then by setting the `fx` and `fy` positions of the dragged node in `dragged` based on the `d3.event` position, we are setting the simulation's position for that node. Then when `tickSimulation` is called (and remember it is called frequently) the charts nodes and links will update for that dragged node.

#### Challenge 1. Tooltip on hover

We have added the `d3-tooltip` library to the directory. Can you add a tooltip on hover to show the name for each character?

#### Challenge 2. Pinning nodes

Can you figure out how to drag and pin a node? Pinning follows this behavior: once a node has been dragged then it will be positioned at that point until it is dragged again. The simulation should not change the `fx` and `fy` of that node.

Congratulations, you have now finished Lab 9 and created your first D3 network layout. Our final lab after the break will cover geographic maps with D3 and GeoJSON datasets.

#### This lab was based on the following material:

- Hanspeter Pfister's CS171 Lab Material (Harvard)
- [D3 - Interactive Data Visualization for the Web](#) by Scott Murray

[Home](#)