Data visualization

COSC 480B

Reyan Ahmed

rahmed1@colgate.edu

Lecture 11

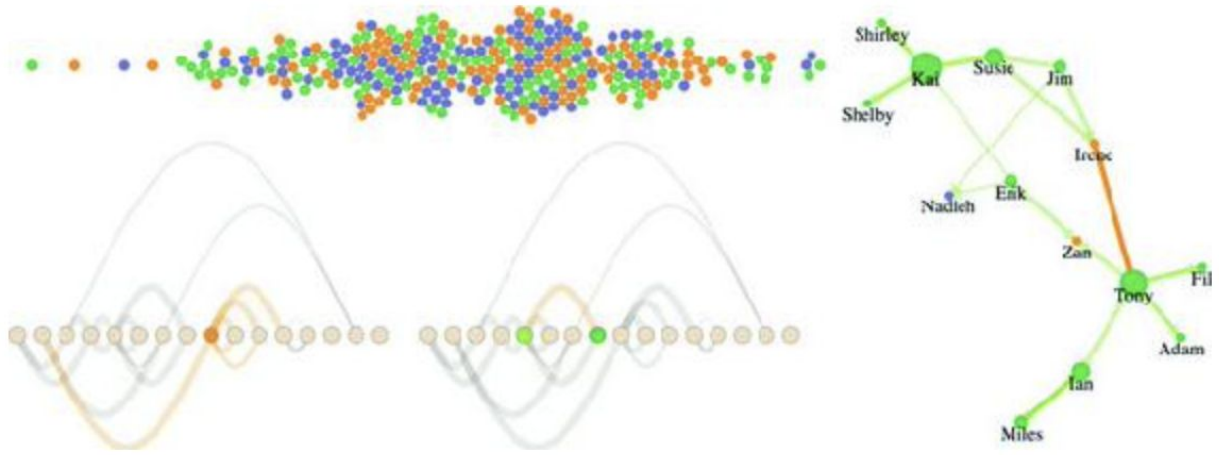
Network visualization

Overview

- Creating adjacency matrices and arc diagrams
- Using the force-directed layout
- Using constrained forces
- Representing directionality
- Adding and removing network nodes and edges

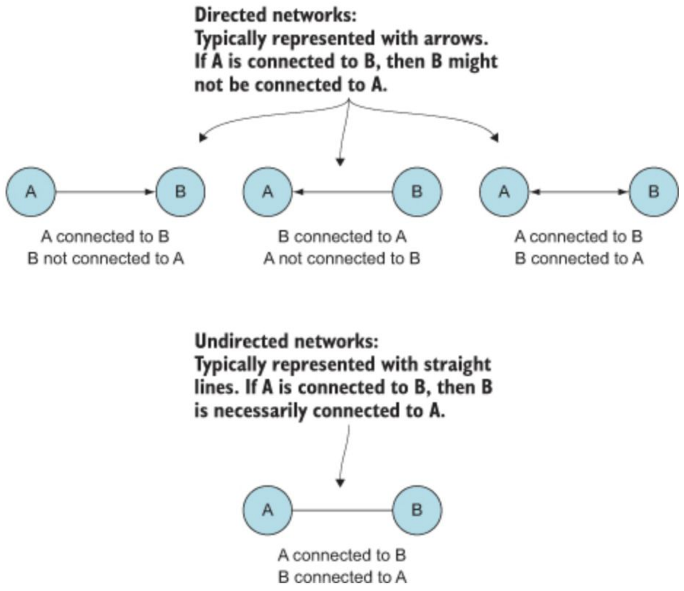
Overview

Along with explaining the basics of network analysis, this chapter includes laying out networks using xy positioning, force-directed algorithms, adjacency matrices, and arc diagrams



Network data

Some basic kinds of network connections (directed, reciprocated, and undirected) that show up in basic networks like simple directed and undirected networks


Directed networks:
Typically represented with arrows. If A is connected to B, then B might not be connected to A.
A
B
A connected to B
B not connected to A
A
B
B connected to A
A not connected to B
A
B
A connected to B
B connected to A
Undirected networks:
Typically represented with straight lines. If A is connected to B, then B is necessarily connected to A.
A
B
A connected to B
B connected to A

Network data

edgelist.csv

```
source,target,weight
Jim,Irene,5
Susie,Irene,5
Jim,Susie,5
Susie,Kai,5
Shirley,Kai,5
Shelby,Kai,5
Kai,Susie,5
Kai,Shirley,5
Kai,Shelby,5
Erik,Zan,5
Tony,Zan,5
```

```
Tony,Fil,5
Tony,Ian,5
Tony,Adam,5
Fil,Tony,4
Ian,Miles,1
Adam,Tony,3
Miles,Ian,2
Miles,Ian,3
Erik,Kai,2
Erik,Nadieh,2
Jim,Nadieh,2
```

Network data

nodelist.csv

```
id,role,salary
Irene,manager,300000
Zan,manager,380000
Jim,employee,150000
Susie,employee,90000
Kai,employee,135000
Shirley,employee,60000
Erik,employee,90000
Shelby,employee,150000
```

```
Tony,employee,72000
Fil,employee,35000
Adam,employee,85000
Ian,employee,83000
Miles,employee,99000
Sarah,employee,160000
Nadieh,contractor,240000
Hajra,contractor,280000
```

Network data

networks.css

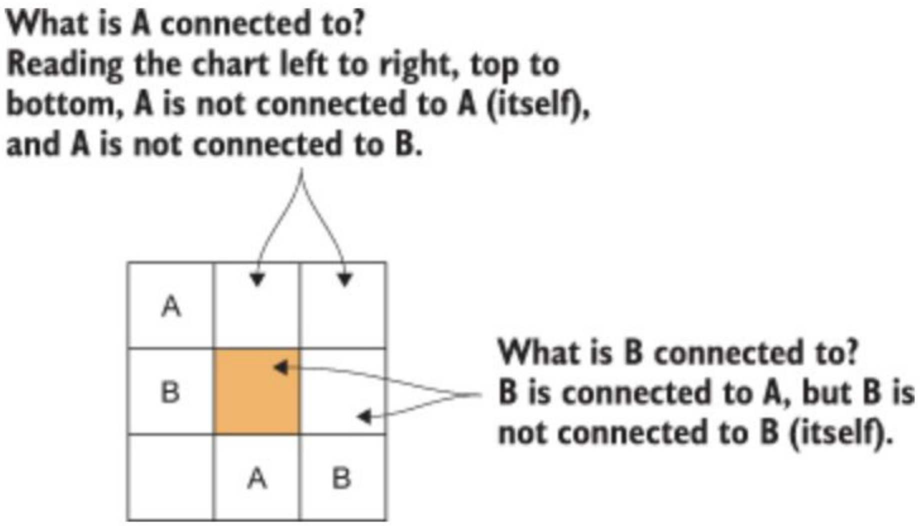
```
.grid {
  stroke: #9A8B7A;
  stroke-width: 1px;
  fill: #CF7D1C;
}
.arc {
  stroke: #9A8B7A;
  fill: none;
}
.node {                          1
  fill: #EBD8C1;
  stroke: #9A8B7A;
  stroke-width: 1px;
}
```

```
circle.active {
  fill: #FE9922;
}
path.active {
  stroke: #FE9922;
}
circle.source {
  fill: #93C464;
}
circle.target {
  fill: #41A368;
}
```

1 If you set the style of a <g> element, it will set that style for all its children, which can be useful if you have multipart elements that you want to have the same style

Network data

How edges are described graphically in an adjacency matrix. In this kind of diagram, the nodes are listed on the axes as columns, and a connection is indicated by a shaded cell where those columns intersect.


What is A connected to?
Reading the chart left to right, top to bottom, A is not connected to A (itself), and A is not connected to B.
A
B
A
B
What is B connected to?
B is connected to A, but B is not connected to B (itself).

Network data

The adjacency matrix function

```
function adjacency() {
  var PromiseWrapper = d => new Promise(resolve => d3.csv(d, p => resolve(p)))1
     Promise.all([PromiseWrapper("nodelist.csv"),
            PromiseWrapper("edgelist.csv")])
         .then(resolve => {
           createAdjacencyMatrix(resolve[0], resolve[1])                     2
         })
```

1 We need to wrap our calls in promises to use promise.all
2 Promise.all returns an array of results in the order of the promises that were sent

Network data

```
function createAdjacencyMatrix(nodes, edges) {
  var edgeHash = {};
  edges.forEach(edge => {
    var id = edge.source + "-" + edge.target;
    edgeHash[id] = edge;                                    3
  })
```

3 A hash allows us to test whether a source-target pair has a link

Network data

```
 var matrix = [];
 nodes.forEach((source, a) => {
   nodes.forEach((target, b) => {                                    4
   var grid =
     {id: source.id + "-" + target.id,
          x: b, y: a, weight: 0};                                5
    if (edgeHash[grid.id]) {
      grid.weight = edgeHash[grid.id].weight;                            6
    }
    matrix.push(grid);
  })
})
```

4 Creates all possible source-target connections
5 Sets the xy coordinates based on the source-target array positions
6 If there’s a corresponding edge in our edge list, give it that weight

Network data

```
d3.select("svg")
  .append("g")
  .attr("transform", "translate(50,50)")
  .attr("id", "adjacencyG")
  .selectAll("rect")
  .data(matrix)
  .enter()
  .append("rect")
  .attr("class", "grid")
  .attr("width", 25)
  .attr("height", 25)
  .attr("x", d => d.x * 25)
  .attr("y", d => d.y * 25)
  .style("fill-opacity", d => d.weight * .2)
```

Network data

```
d3.select("svg")                                              7
  .append("g")
  .attr("transform", "translate(50,45)")
  .selectAll("text")
  .data(nodes)
  .enter()
  .append("text")
  .attr("x", (d,i) => i * 25 + 12.5)
  .text(d => d.id)
  .style("text-anchor", "middle")
```

7 Creates horizontal labels from the nodes

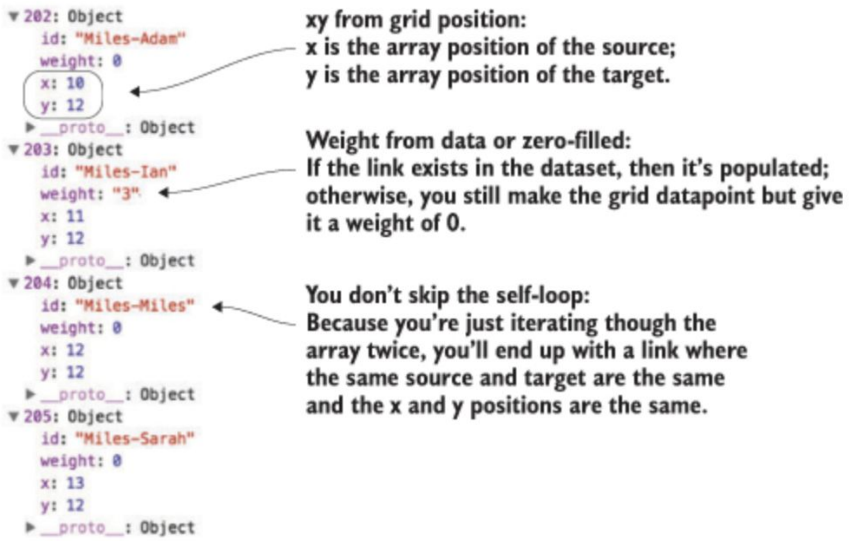
Network data

```
    d3.select("svg")                                                8
      .append("g")
      .attr("transform", "translate(45,50)")
      .selectAll("text")
      .data(nodes)
      .enter()
      .append("text")
      .attr("y", (d,i) => i * 25 + 12.5)
      .text(d => d.id)
      .style("text-anchor", "end")
  };
};
```

8 Vertical labels with text-anchor: end because that will line it up better

Network data

The array of connections we're building. Notice that every possible connection is stored in the array. Only those connections that exist in our dataset have a weight value other than 0. Also note that our CSV import creates the weight value as a string.



```
▼202: Object
   id: "Miles-Adam"
   weight: 0
   x: 10
   y: 12
  ▶__proto__: Object
▼203: Object
   id: "Miles-Ian"
   weight: "3"
   x: 11
   y: 12
  ▶__proto__: Object
▼204: Object
   id: "Miles-Miles"
   weight: 0
   x: 12
   y: 12
  ▶__proto__: Object
▼205: Object
   id: "Miles-Sarah"
   weight: 0
   x: 13
   y: 12
  ▶__proto__: Object
```

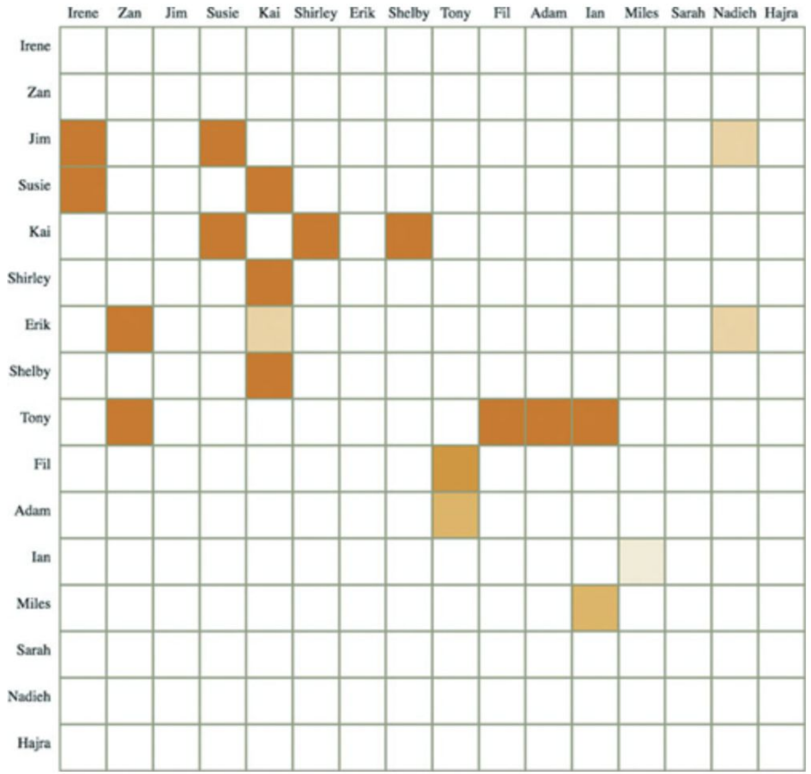
xy from grid position:
x is the array position of the source;
y is the array position of the target.

Weight from data or zero-filled:
If the link exists in the dataset, then it's populated; otherwise, you still make the grid datapoint but give it a weight of 0.

You don't skip the self-loop:
Because you're just iterating though the array twice, you'll end up with a link where the same source and target are the same and the x and y positions are the same.

Network data

A weighted, directed adjacency matrix where lighter orange indicates weaker connections and darker orange indicates stronger connections. The source is on the y-axis, and the target is on the x-axis. The matrix shows that Sarah, Nadieh, and Hajra didn't give anyone feedback, whereas Kai gave Susie feedback, and Susie gave Kai feedback (what we call a reciprocated tie in network analysis).



	Irene	Zan	Jim	Susie	Kai	Shirley	Erik	Shelby	Tony	Fil	Adam	Ian	Miles	Sarah	Nadieh	Hajra
Irene																
Zan																
Jim	🟧			🟧											🟨	
Susie	🟧				🟧											
Kai				🟧		🟧		🟧								
Shirley					🟧											
Erik		🟧			🟨										🟨	
Shelby					🟧											
Tony		🟧								🟧	🟧	🟧				
Fil									🟧							
Adam									🟨							
Ian													🟨			
Miles												🟨				
Sarah																
Nadieh																
Hajra																

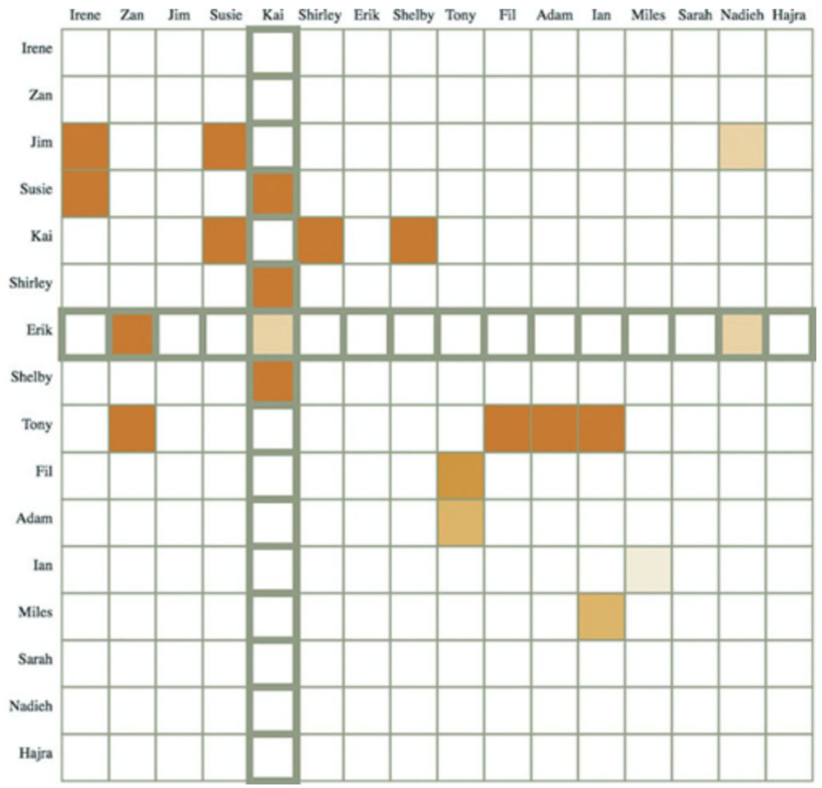
Network data

Grids can be hard to read without something to highlight the row and column of a square. It’s simple to add highlighting to our matrix.

```
d3.selectAll("rect.grid").on("mouseover", gridOver);
function gridOver(d) {
  d3.selectAll("rect").style("stroke-width", p =>
  p.x == d.x || p.y == d.y ? "4px" : "1px");
};
```

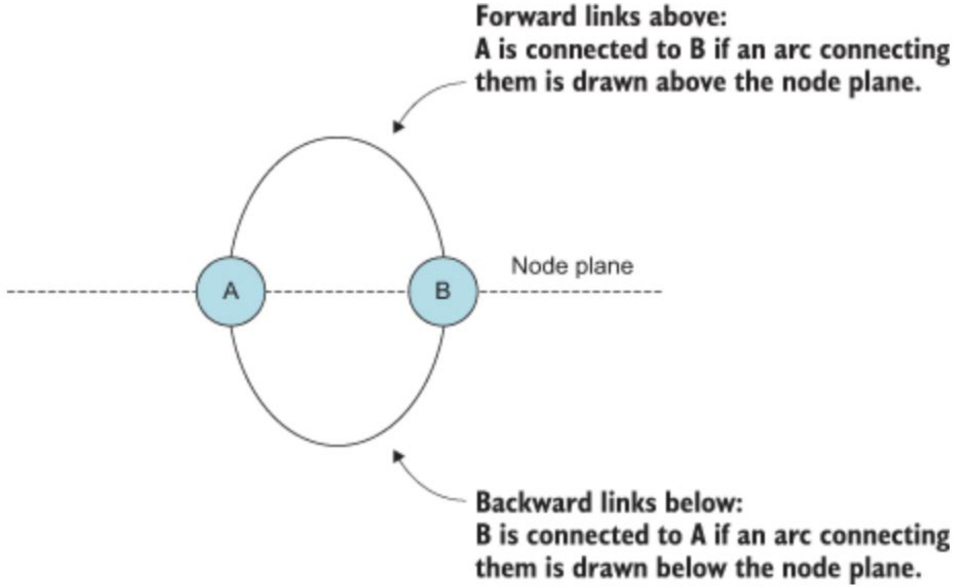
Network data

Adjacency highlighting of the column and row of the grid square. In this instance, the mouse is over the Erik-to-Kai edge, and as a result highlights the Erik row and the Kai column. You can see that Erik gave feedback to three people, whereas Kai received feedback from four people.



Network data

The components of an arc diagram are circles for nodes and arcs for connections, with nodes laid out along a baseline and the location of the arc relative to that baseline indicative of the direction of the connection.


Forward links above:
A is connected to B if an arc connecting them is drawn above the node plane.
A
B
Node plane
Backward links below:
B is connected to A if an arc connecting them is drawn below the node plane.

Network data

Arc diagram code

```
function createArcDiagram(nodes,edges) {                    1
 var nodeHash = {};
 nodes.forEach((node, x) => {
   nodeHash[node.id] = node;                           2
   node.x = parseInt(x) * 30;                         2
 })
 edges.forEach(edge => {
   edge.weight = parseInt(edge.weight);
   edge.source = nodeHash[edge.source];                    3
   edge.target = nodeHash[edge.target];                   3
 })
```

1 Takes the results of the same Promise.all as adjacencyMatrix
2 Creates a hash that associates each node JSON object with its ID value and sets each node with an x position based on its array position
3 Replaces the string ID of the node with a pointer to the JSON object

Network data

```
var arcG = d3.select("svg").append("g").attr("id", "arcG")
     .attr("transform", "translate(50,250)");

arcG.selectAll("path")
   .data(edges)
   .enter()
   .append("path")
   .attr("class", "arc")
   .style("stroke-width", d => d.weight * 2)
   .style("opacity", .25)
   .attr("d", arc)
arcG.selectAll("circle")
   .data(nodes)
   .enter()
   .append("circle")
   .attr("class", "node")
   .attr("r", 10)
   .attr("cx", d => d.x)
```

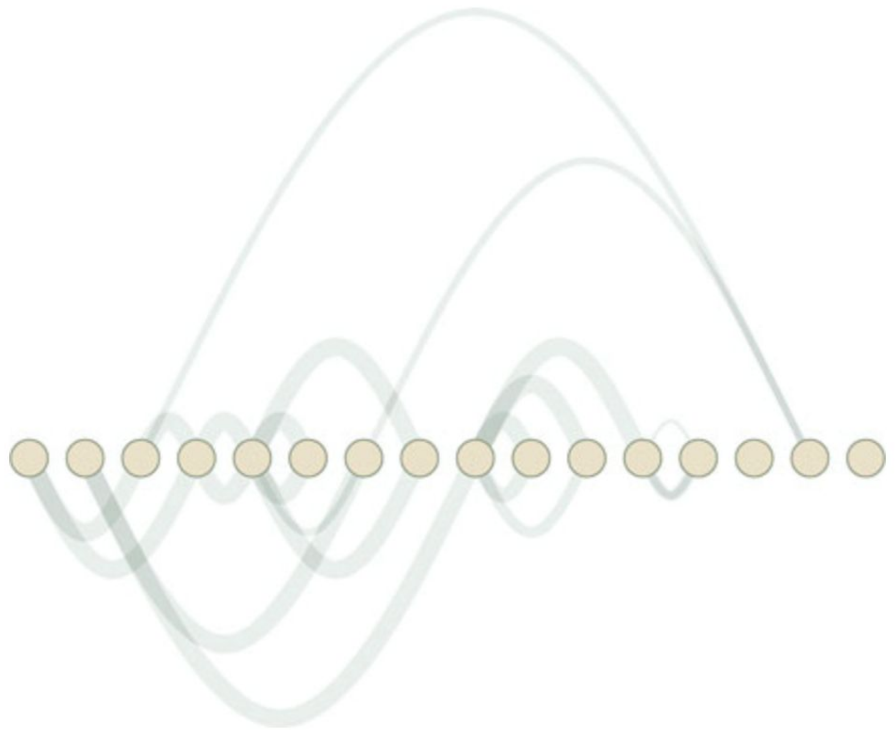
Network data

```
  function arc(d,i) {                                       4
    var draw = d3.line().curve(d3.curveBasis)
    var midX = (d.source.x + d.target.x) / 2
    var midY = (d.source.x - d.target.x)
    return draw([[d.source.x,0],[midX,midY],[d.target.x,0]])
  }
}
```

4 Draws a basis-interpolated line from the source node to a computed middle point above them to the target node

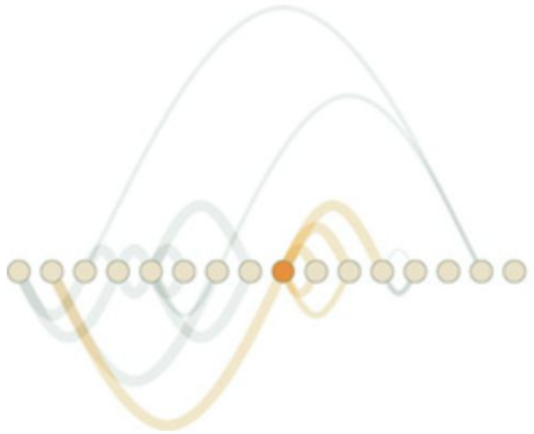
Network data

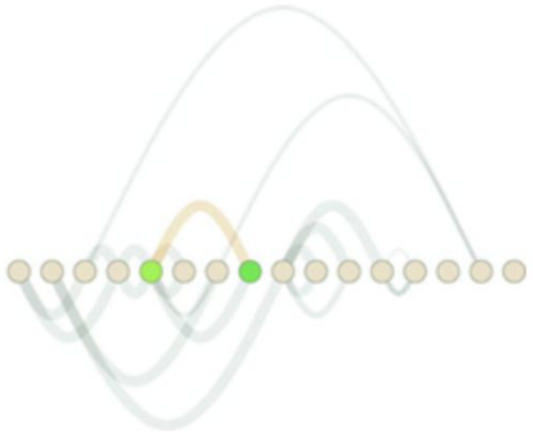
An arc diagram, with connections between nodes represented as arcs above and below the nodes. We can see the first (left) two nodes have no outgoing links, and the rightmost three nodes also have no outgoing links. The length of the arcs is meaningless and based on how we’ve laid the nodes out (nodes that are far away will have longer links), but the width of the arcs is based on the weight of the connection.



Network data

Mouseover behavior on edges (right), with the edge being moused over in orange, the source node in light green, and the target node in dark green. Mouseover behavior on nodes (left), with the node being moused over in orange and the connected edges in light orange.





Network data

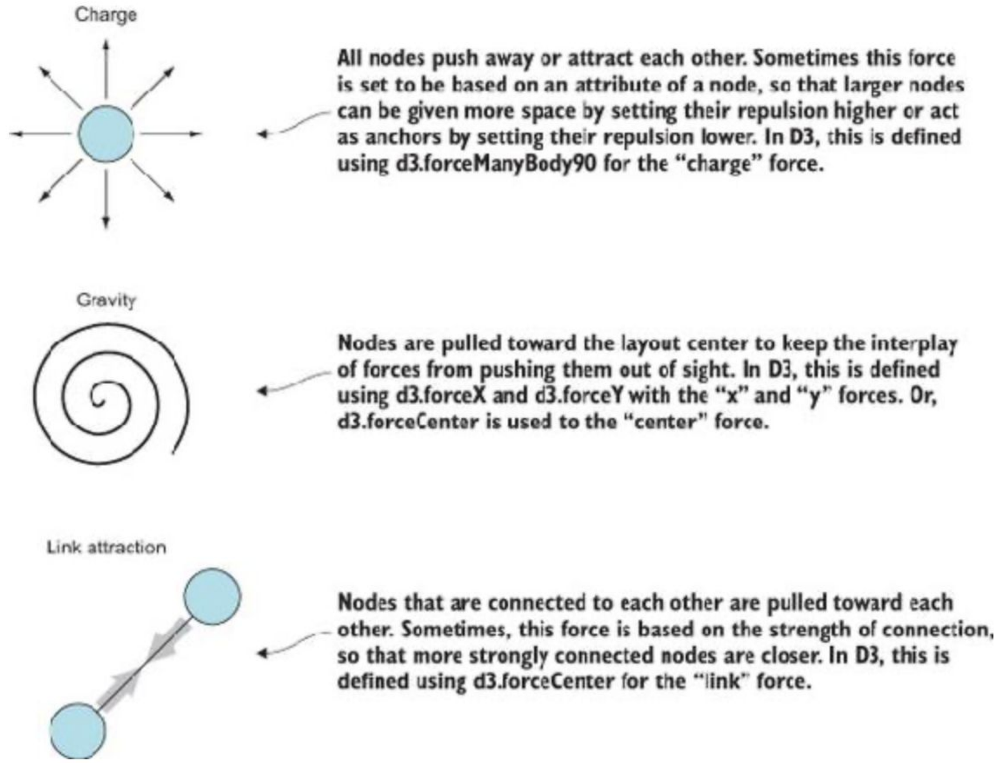
Arc diagram interactivity

```
d3.selectAll("circle").on("mouseover", nodeOver)
d3.selectAll("path").on("mouseover", edgeOver)
function nodeOver(d) {
  d3.selectAll("circle").classed("active", p => p === d)         1
  d3.selectAll("path").classed("active", p => p.source === d
       || p.target === d)                                        2
}
function edgeOver(d) {
  d3.selectAll("path").classed("active", p => p === d)
  d3.selectAll("circle")
    .classed("source", p => p === d.source)                      3
    .classed("target", p => p === d.target)                      3
}
```

1 Makes a selection of all nodes to set the class of the node being hovered over to “active”
2 Any edge where the selected node shows up as source or target renders as red
3 This nested if checks to see if a node is the source or target and sets its class accordingly

Force-directed layout

The forces in a force-directed algorithm: attraction/repulsion, gravity, and link attraction. Other factors, such as hierarchical packing and community detection, can also be factored into force-directed algorithms, but the aforementioned features are the most common. Forces are approximated for larger networks to improve performance.



All nodes push away or attract each other. Sometimes this force is set to be based on an attribute of a node, so that larger nodes can be given more space by setting their repulsion higher or act as anchors by setting their repulsion lower. In D3, this is defined using d3.forceManyBody90 for the "charge" force.

Nodes are pulled toward the layout center to keep the interplay of forces from pushing them out of sight. In D3, this is defined using d3.forceX and d3.forceY with the "x" and "y" forces. Or, d3.forceCenter is used to the "center" force.

Nodes that are connected to each other are pulled toward each other. Sometimes, this force is based on the strength of connection, so that more strongly connected nodes are closer. In D3, this is defined using d3.forceCenter for the "link" force.

Force-directed layout

An initial force simulation with no links or collision detection

```
var roleScale = d3.scaleOrdinal()
  .range(["#75739F", "#41A368", "#FE9922"])

var sampleData = d3.range(100).map((d,i) => ({r: 50 - i * .5}))
1

var manyBody = d3.forceManyBody().strength(10)
2
var center = d3.forceCenter().x(250).y(250)                    3

var.force("charge", manyBody)                                 4
   .force("center", center)                                 4
   .nodes(sampleData)                                        5
   .on("tick", updateNetwork)                                 6
```

1 Creating a hundred circles ranging in size from .5 radius to 49.5
2 Register a manyBody force with positive strength to make it attractive
3 Register a center strength to try to make the nodes center at 250,250
4 Attach the forces to our simulation
5 Send the nodes array to simulation so it knows what to calculate with
6 At each tick, have it run the updateNetwork function

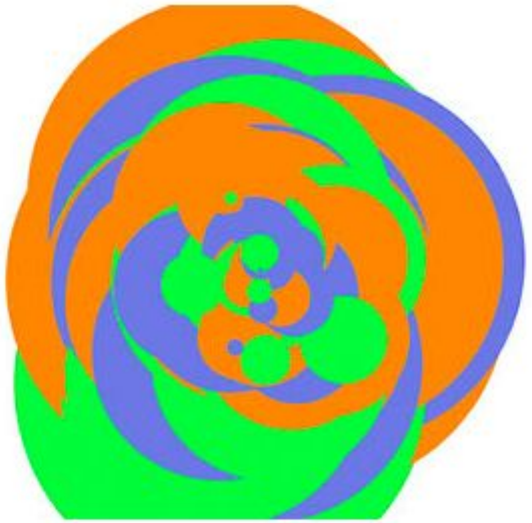
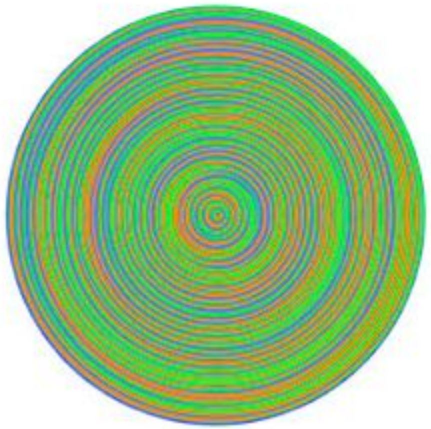
Force-directed layout

```
d3.select("svg")
   .selectAll("circle")
   .data(sampleData)                                                7
   .enter()
   .append("circle")
   .style("fill", (d, i) => roleScale(i))
   .attr("r", d => d.r)

function updateNetwork() {
   d3.selectAll("circle")
     .attr("cx", d => d.x)                                      8
     .attr("cy", d => d.y)
}
```

7 Draw a circle for each datapoint
8 Each time the simulation ticks, update their position based on the newly calculated position by the simulation

Force-directed layout

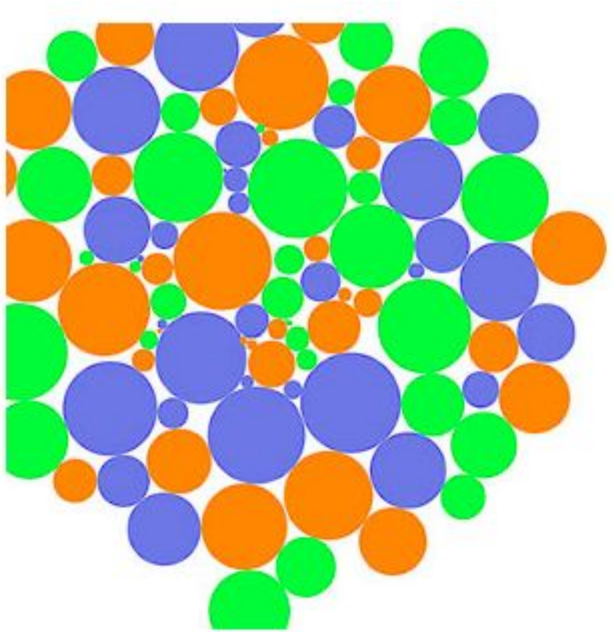
The results of a force simulation where the only force acting on the nodes is attraction

Force-directed layout

let's register a "collide" force:

```
.force("collision", d3.forceCollide(d => d.r))
```



Force-directed layout

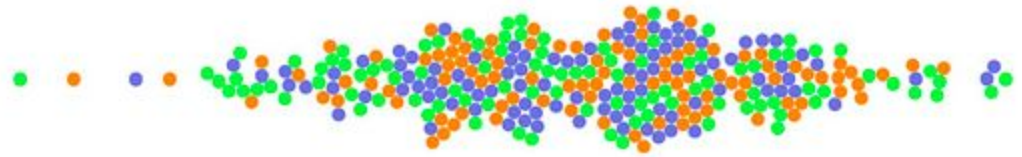
Code modifications for a beeswarm plot

```
var sampleData = d3.range(300).map(() =>
({r: 2, value: 200 + d3.randomNormal()() * 50}))              1
...
  var force = d3.forceSimulation()
    .force("collision", d3.forceCollide(d => d.r))
    .force("x", d3.forceX(100))                            2
    .force("y", d3.forceY(d => d.value).strength(3))          3
    .nodes(sampleData)
    .on("tick", updateNetwork)
```

1 A normally distributed bunch of points that we've offset so they can easily appear onscreen
2 Exert a force on each node that tries to make its x position as close to 100 as possible
3 Exert a stronger force on each node that tries to make its y position reflect its value

Force-directed layout

A beeswarm plot created with our code (rotated to better fit on the page)



Force-directed layout

Force layout function:

```
function createForceLayout(nodes,edges) {
var roleScale = d3.scaleOrdinal()
  .domain(["contractor", "employee", "manager"])
  .range(["#75739F", "#41A368", "#FE9922"])

     var nodeHash = nodes.reduce((hash, node) =>
{hash[node.id] = node;
return hash;
}, {})

     edges.forEach(edge => {
        edge.weight = parseInt(edge.weight)
        edge.source = nodeHash[edge.source]
        edge.target = nodeHash[edge.target]
      })

    var linkForce = d3.forceLink()
```

Force-directed layout

```
 var simulation = d3.forceSimulation()
  .force("charge", d3.forceManyBody().strength(-40))        1
  .force("center", d3.forceCenter().x(300).y(300))
  .force("link", linkForce)
  .nodes(nodes)
  .on("tick", forceTick)

simulation.force("link").links(edges)

d3.select("svg").selectAll("line.link")
   .data(edges, d => `${d.source.id}-${d.target.id}`)       2
   .enter()
   .append("line")
   .attr("class", "link")
   .style("opacity", .5)
   .style("stroke-width", d => d.weight);
```

1 How much each node pushes away each other—if set to a positive value, nodes attract each other
2 Key values for your nodes and edges will help when we update the network later

Force-directed layout

```
var nodeEnter = d3.select("svg").selectAll("g.node")        2
   .data(nodes, d => d.id)
   .enter()
   .append("g")
   .attr("class", "node");
nodeEnter.append("circle")
   .attr("r", 5)
   .style("fill", d => roleScale(d.role))
nodeEnter.append("text")
   .style("text-anchor", "middle")
   .attr("y", 15)
   .text(d => d.id);
```

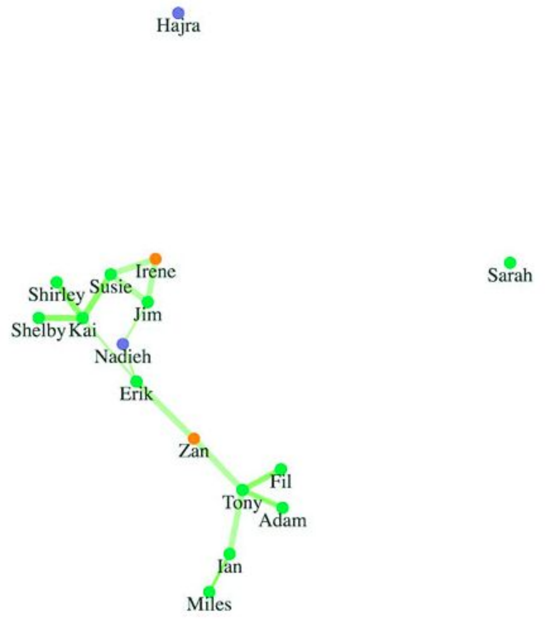
2 Key values for your nodes and edges will help when we update the network later

Force-directed layout

```
   function forceTick() {
     d3.selectAll("line.link")
        .attr("x1", d => d.source.x)
        .attr("x2", d => d.target.x)
        .attr("y1", d => d.source.y)
        .attr("y2", d => d.target.y)
     d3.selectAll("g.node")
        .attr("transform", d => `translate(${d.x},${d.y})`)
   }
}
```

Force-directed layout

A force-directed layout based on our dataset and organized graphically using default settings in the force layout. Managers are in orange, employees green, and contractors purple.



Force-directed layout

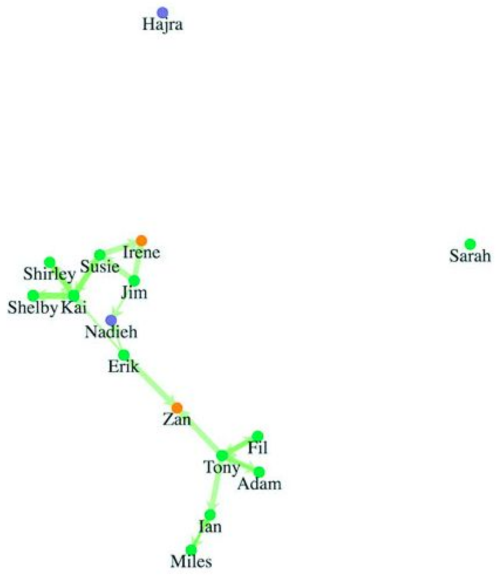
Marker definition and application

```
var marker = d3.select("svg").append('defs')
   .append('marker')
   .attr("id", "triangle")
   .attr("refX", 12)
   .attr("refY", 6)
   .attr("markerUnits", 'userSpaceOnUse')                    1
   .attr("markerWidth", 12)
   .attr("markerHeight", 18)
   .attr("orient", 'auto')
   .append('path')
   .attr("d", 'M 0 0 12 6 0 12 3 6');
d3.selectAll("line").attr("marker-end", "url(#triangle)");    2
```

1 The default setting for markers bases their size off the stroke-width of the parent, which in our case would result in difficult-to-read markers
2 A marker is assigned to a line by setting the marker-end, marker-start, or marker-mid attribute to point to the marker

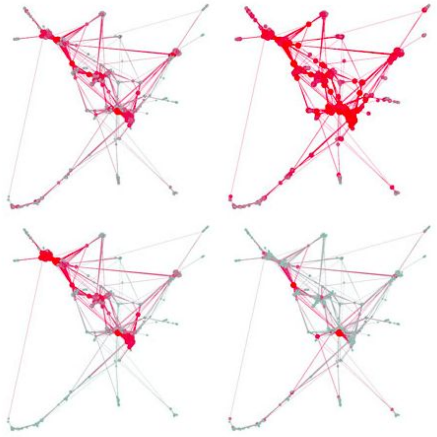
Force-directed layout

Edges now display markers (arrowheads) indicating the direction of connection. Notice that all the arrowheads are the same size. You can control the color of the arrowheads by using CSS rules such as marker > path {fill: # 93C464;}.



Force-directed layout

The same network measured using degree centrality (top left), closeness centrality (top right), eigenvector centrality (bottom left), and betweenness centrality (bottom right). We'll only see degree centrality, but you can explore the others with libraries like jsnetworkx.js. More-central nodes are larger and bright red, whereas less-central nodes are smaller and gray. Notice that although some nodes are central according to all measures, their relative centrality varies, as does the overall centrality of other nodes.



Force-directed layout

You can calculate degree centrality easily by filtering the edges array to show only links that involve that node:

```
nodes.forEach(d => {
  d.degreeCentrality = edges.filter(
    p => p.source === d || p.target === d).length
})
```

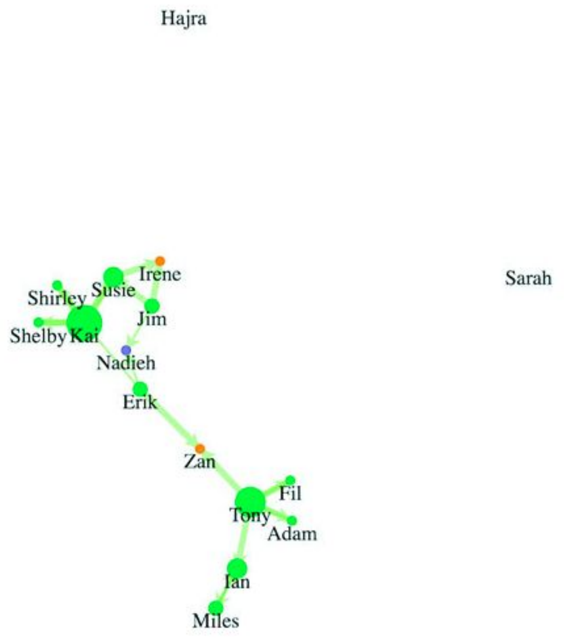
Force-directed layout

let’s add a button that resizes the nodes based on their weight attribute:

```
d3.select("#controls").append("button")
  .on("click", sizeByDegree).html("Degree Size")
function sizeByDegree() {
  simulation.stop()
  simulation.force("charge", d3.forceManyBody()
    .strength(d => -d.degreeCentrality * 20))
  simulation.restart()
  d3.selectAll("circle")
    .attr("r", d => d.degreeCentrality * 2)
}
```

Force-directed layout

Sizing nodes by weight indicates the number of total connections for each node by setting the radius of the circle equal to the weight times 2.



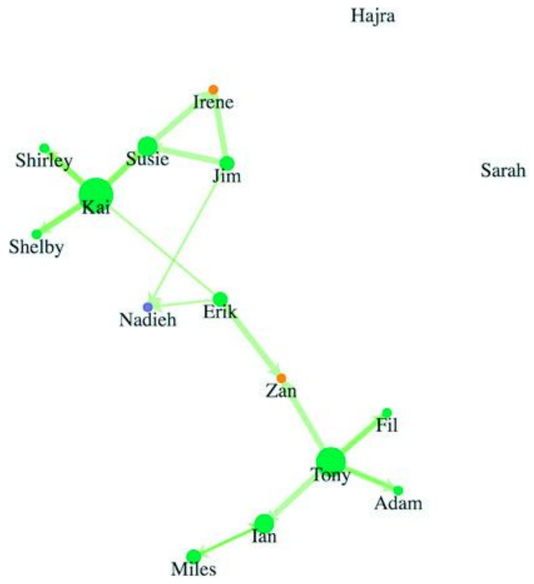
Force-directed layout

If your network has stronger and weaker links, as our example does, then it makes sense to have those edges exert stronger and weaker effects on the controlling nodes.

```
var linkForce = d3.forceLink().strength(d => d.weight * .1)

var simulation = d3.forceSimulation()
  .force("charge", d3.forceManyBody().strength(-500))
  .force("x", d3.forceX(250))
  .force("y", d3.forceY(250))
```

Force-directed layout

By basing the strength of the attraction between nodes on the strength of the connections between nodes, you see a dramatic change in the structure of the network. The weaker connections between x and y allow that part of the network to drift away.



Force-directed layout

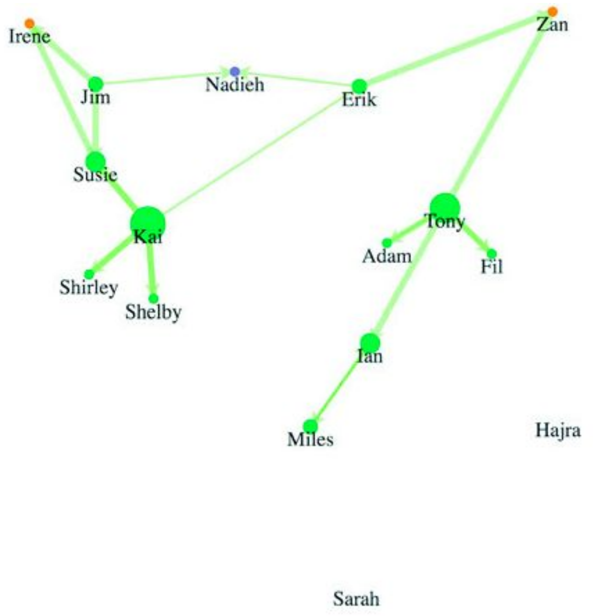
Setting up drag for networks

```
var drag = d3.drag()
drag
     .on("drag", dragging)                    1

function dragging(d) {
    var e = d3.event                          2
    d.fx = e.x                            3
    d.fy = e.y
    if (simulation.alpha() < 0.1) {
      simulation.alpha(0.1)                   4
      simulation.restart()
    }
}

d3.selectAll("g.node").call(drag);         5
```

1 The drag behavior also exposes a “start” and “end” event
2 This will give us the event so we can get the current x and y coordinates
3 Setting fx or fy sets the fixed position of the node
4 If the simulation has cooled down enough, heat it back up and restart it
5 Assign the drag behavior to the nodes

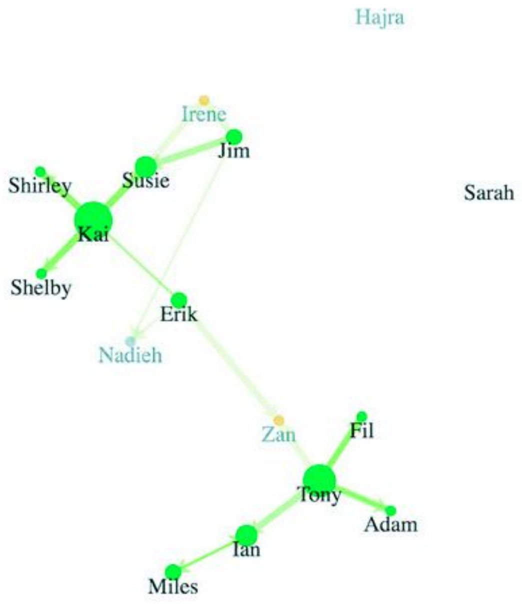
Force-directed layout

The two nodes representing managers have been dragged to the top corners, allowing the rest of the nodes to take their positions based on the forces of the simulation (being dragged toward the center along with being dragged toward the fixed nodes).



Force-directed layout

The network has been filtered to only show nodes that are not managers or contractors. This figure catches two processes in midstream, the transition of nodes from full to 0 opacity, and the removal of edges.



Force-directed layout

Filtering a network

```
function filterNetwork() {
   simulation.stop()
   var originalNodes = simulation.nodes()                                   1
   var originalLinks = simulation.force("link").links()                     1

   var influentialNodes = originalNodes.filter(d => d.role === "employee") 2
   var influentialLinks = originalLinks.filter(d =>
        influentialNodes.includes(d.source) &&
        influentialNodes.includes(d.target))                               2

   d3.selectAll("g.node")
      .data(influentialNodes, d => d.id)
      .exit()
      .transition()
      .duration(4000)
      .style("opacity", 0)
      .remove()
```

1 Accesses the current array of nodes and array of links associated with the force layout
2 Makes an array of nodes and links only out of those that reference existing nodes

Force-directed layout

Filtering a network

```
   d3.selectAll("line.link")
      .data(influentialLinks, d => `${d.source.id}-${d.target.id}`)
      .exit()
      .transition()
      .duration(3000)
      .style("opacity", 0)                                   3
      .remove()

   simulation
      .nodes(influentialNodes)                                  4
   simulation.force("link")
      .links(influentialLinks)
   simulation.alpha(0.1)
   simulation.restart()
}
```

3 By setting a transition on the exit(), it applies the transition only to those nodes being removed and waits until the transition is finished to remove them
4 Reinitialize the simulation with only the existing nodes and edges and reheat it by setting the alpha and restart the network

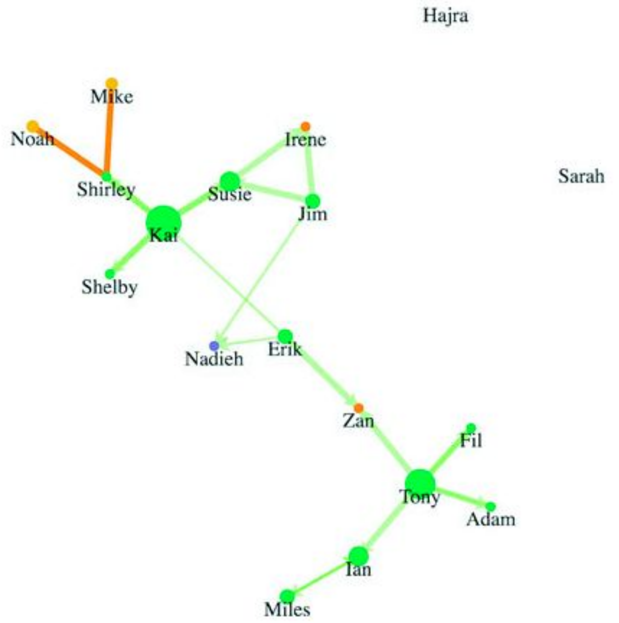
Force-directed layout

A function for adding edges

```
function addEdge() {
    simulation.stop()
    var links = simulation.force("link").links()
    var nodes = simulation.nodes()
    var newEdge = {source: nodes[0], target: nodes[8], weight: 5}
    links.push(newEdge)
    simulation.force("link").links(links)
    d3.select("svg").selectAll("line.link")
    .data(links, d => `${d.source.id}-${d.target.id}`)
    .enter()
    .insert("line", "g.node")
    .attr("class", "link")
    .style("stroke", "#FE9922")
    .style("stroke-width", 5)

    simulation.alpha(0.1)
    simulation.restart()
}
```

Force-directed layout

Network with two new nodes added (Mike and Noah), both with links to Shirley



Force-directed layout

Function for adding nodes and edges

```
function addNodesAndEdges() {
   simulation.stop()
   var oldEdges = simulation.force("link").links()
   var oldNodes = simulation.nodes()
   var newNode1 = {id: "Mike", role: "contractor", team: "none"}          1
   var newNode2 = {id: "Noah", role: "contractor", team: "none"}          1
   var newEdge1 = {source: oldNodes[5], target: newNode1, weight: 5}      2
   var newEdge2 = {source: oldNodes[5], target: newNode2, weight: 5}      2
   oldEdges.push(newEdge1,newEdge2)                                       3
   oldNodes.push(newNode1,newNode2)                                       3
   simulation.force("link").links(oldEdges)
   simulation.nodes(oldNodes)
   d3.select("svg").selectAll("line.link")
      .data(oldEdges, d => d.source.id + "-" + d.target.id)
    .enter()
    .insert("line", "g.node")
    .attr("class", "link")
    .style("stroke", "#FE9922")
    .style("stroke-width", 5)
```

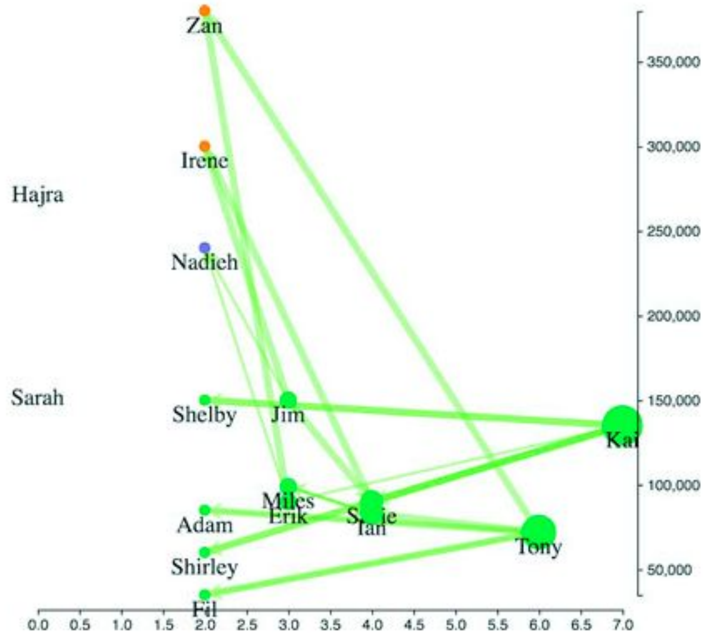
1 A node is always an object, and we want the new nodes to have the same structure as our original nodes
2 The edges reference an original node and a new node
3 We need to add the new nodes and edges to our existing arrays for the simulation to be aware of them

Force-directed layout

```
 var nodeEnter = d3.select("svg").selectAll("g.node")
    .data(oldNodes, d => d.id)
    .enter()
    .append("g")
    .attr("class", "node")
 nodeEnter.append("circle")
    .attr("r", 5)
    .style("fill", "#FCBC34")
 nodeEnter.append("text")
    .style("text-anchor", "middle")
    .attr("y", 15)
    .text(d => d.id)
    simulation.alpha(0.1)
    simulation.restart()
}
```

Force-directed layout

When the network is represented as a scatterplot, the links increase the visual clutter. It provides a useful contrast to the force-directed layout, but can be hard to read on its own.



Force-directed layout

Moving our nodes manually

```
function manuallyPositionNodes() {
   var xExtent = d3.extent(simulation.nodes(), d =>
   parseInt(d.degreeCentrality))
   var yExtent = d3.extent(simulation.nodes(), d => parseInt(d.salary))
1
   var xScale = d3.scaleLinear().domain(xExtent).range([50,450])
   var yScale = d3.scaleLinear().domain(yExtent).range([450,50])
   simulation.stop()
   d3.selectAll("g.node")
       .transition()
       .duration(1000)
       .attr("transform", d => `translate(${xScale(d.degreeCentrality)
                      },${yScale(d.salary) })`)                    2
```

1 We need to parseInt the salary value because d3.csv brings everything in as strings
2 Manually move the nodes and links

Force-directed layout

```
   d3.selectAll("line.link")
       .transition()
       .duration(1000)
       .attr("x1", d => xScale(d.source.degreeCentrality))            3
       .attr("y1", d => yScale(d.source.salary))
       .attr("x2", d => xScale(d.target.degreeCentrality))
       .attr("y2", d => yScale(d.target.salary))
   var xAxis = d3.axisBottom().scale(xScale).tickSize(4)
   var yAxis = d3.axisRight().scale(yScale).tickSize(4)
   d3.select("svg").append("g").attr("transform",
           "translate(0,460)").call(xAxis)
   d3.select("svg").append("g").attr("transform",
           "translate(460,0)").call(yAxis)
   d3.selectAll("g.node").each(d => {
     d.x = xScale(d.degreeCentrality)                             4
     d.vx = 0                                               5
     d.y = yScale(d.salary)                                   4
     d.vy = 0                                               5
    })
}
```

3 Manually move the nodes and links
4 Update the xy coordinates of the nodes to match their new screen coordinates
5 Zero out the velocity so that when you restart the simulation they don’t have their old velocity in play