# Interactive Data Visualisation

Network Visualisation

Dr Ruairi O'Reilly

## What we'll cover

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

# Network Visualisation

Network analysis and network visualization are more common now with the growth of online social networks like Twitter and Facebook, as well as social media and linked data, all of which are commonly represented with network structures. Network visualizations like the kind you'll see in this chapter, some of which are shown in the figure below, are particularly interesting because they focus on how things are related. They represent systems more accurately than the traditional flat data seen in more common data visualizations.

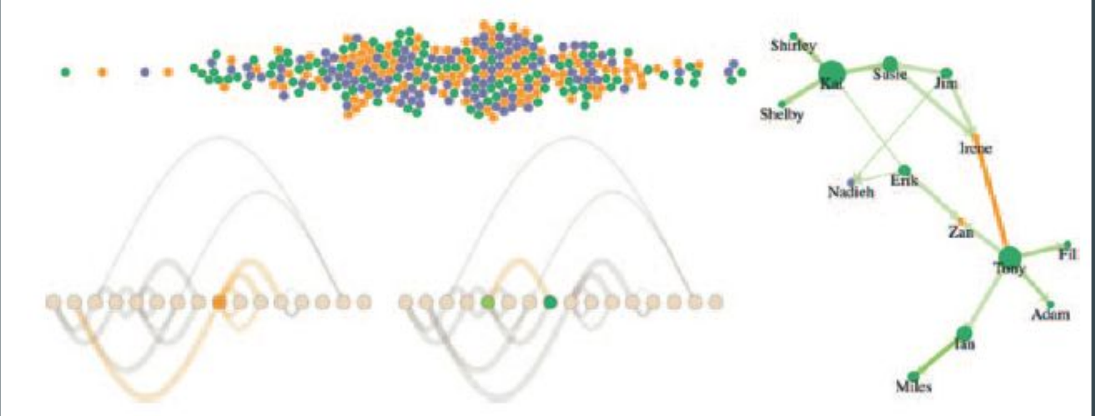


**Figure 7.1** Along with explaining the basics of network analysis (section 7.2.3), this chapter includes laying out networks using xy positioning (section 7.2.5), force-directed algorithms (section 7.2), adjacency matrices (section 7.1.2), and arc diagrams (section 7.1.3).

## Understanding networks

In general, when dealing with networks you refer to the things being connected (like people) as nodes and the connections between them (such as being a friend on Facebook) as *edges* or *links*. Networks may also be referred to as *graphs*, because that's what they're called in mathematics.

Networks aren't only a data format—they're a perspective on data. When you work with network data, you typically try to discover and display patterns of the network or of parts of the network, and not of individual nodes in the network. Although you may use a network visualization because it makes a cool graphical index, like a mind map or a network map of a website, in general you'll find that the typical information visualization techniques are designed to showcase network structure, not individual nodes.

## Static network diagrams

Network data is different from hierarchical data. Networks present the possibility of any-to-many connections, like the Sankey layout seen previously, whereas in hierarchical data a node can have many children but only one parent, like the tree and pack layouts. A network doesn’t have to be a social network. This format can represent many different structures, such as transportation networks and linked open data. We will look at four common forms for representing networks: as data, as adjacency matrices, as arc diagrams, and using force-directed network diagrams.

In each case, the graphical representation will be quite different. For instance, in the case of a force-directed layout, we’ll represent the nodes as circles and the edges as lines. But in the case of the adjacency matrix, nodes will be positioned on x- and y axes, and the edges will be filled squares. Networks don’t have a default representation, but the examples you’ll see in this lecture are the most common.

## Network data

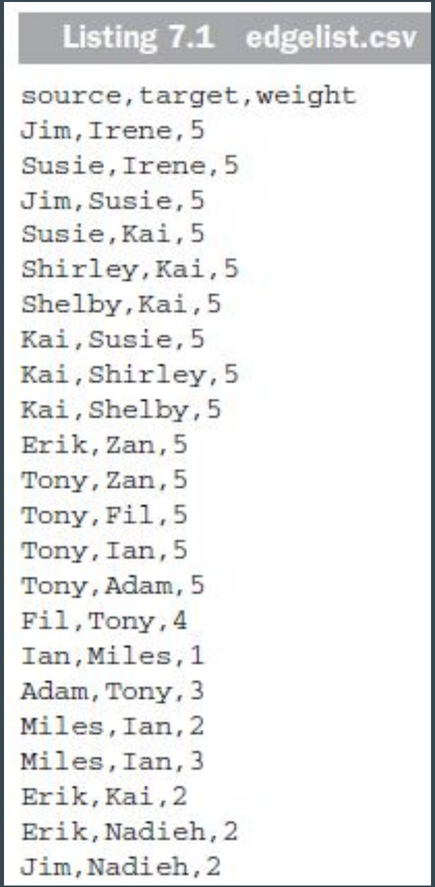
Network data stores nodes, which can be.....

For our purpose, we’re going to get into People Analytics, an exciting new trend in human resources to try to analyze and visualize data related to how organizations perform. It’s data-driven HR, and because HR is all about people, we deal with more interesting datasets than usual—such as text analysis for written reviews or, in our case, network analysis to see team dynamics.

Imagine we have three teams and a couple contractors and every six months they do a 360 review where they give feedback to the people that they worked with over the last six months. At the end of each review, the team member gives a numerical score indicating whether they have confidence in the person they’re reviewing, from 0 (indicating no confidence) to 5 (indicating total confidence). NOTE: Quite common - any examples?

# Edge list

Although you can store networks in several data formats, the most straightforward is known as an edge list. An edge list is typically represented as a CSV like that shown in listing 7.1, with a source column and a target column, and a string or number to indicate which nodes are connected, resulting in connections and networks like those described in figure 7.2. Each edge may also have other attributes, indicating the type of connection or its strength, the time period when the connection is valid, its color, or any other information you want to store about a connection. The important thing is that only the source and target columns are necessary. It's hard to indicate negative links (like people who are connected to each other by their deep and abiding hatred, like how you might connect Harry Potter and Voldemort, or Romeo and Tybalt), so we're only looking at links in our made-up people analytics network where the score is 1 or greater.



**Listing 7.1 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
```

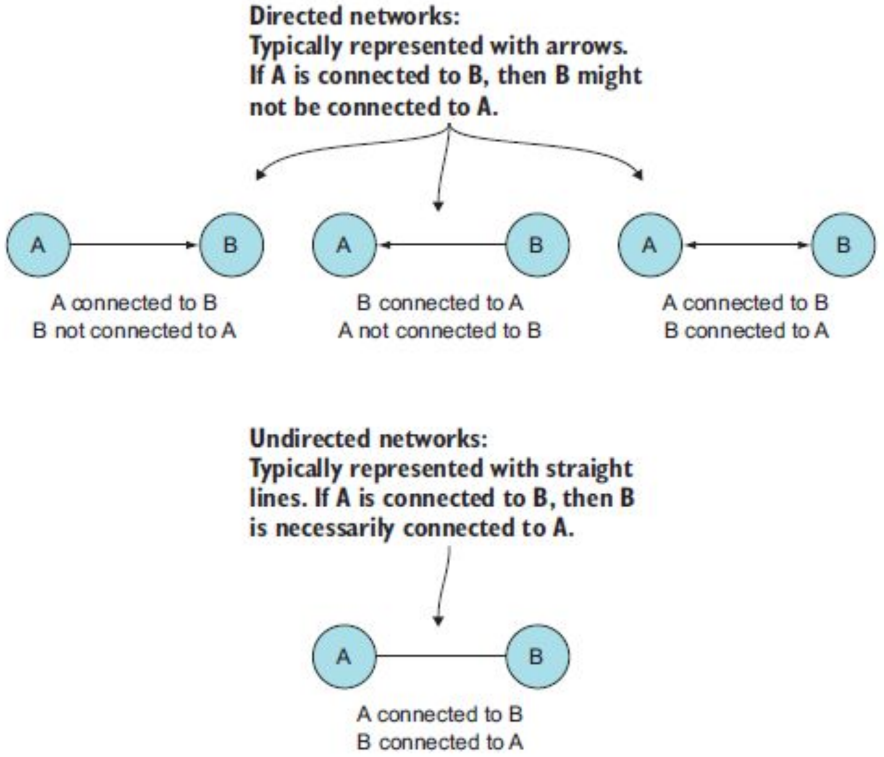
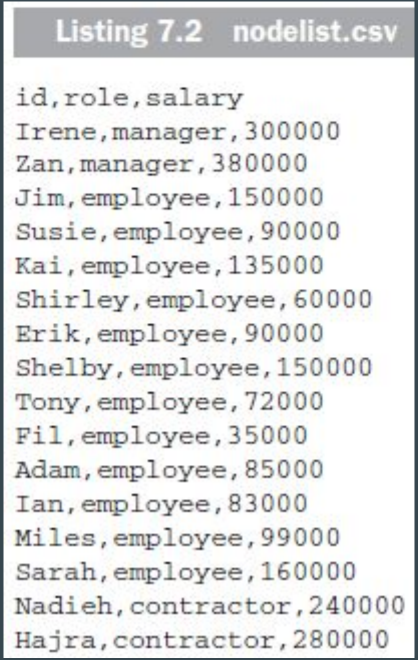

Directed networks:
Typically represented with arrows.
If A is connected to B, then B might not be connected to A.
A
B
A connected to B
B not connected to A
A
B
B connected to A
A not connected to B
A
B
A connected to B
B connected to A
Undirected networks:
Typically represented with straight lines. If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

**Figure 7.2** Some basic kinds of network connections (directed, reciprocated, and undirected) that show up in basic networks like simple directed and undirected networks

# Edge list

Reading this dataset, you can see that Jim and Susie both have total confidence in Irene, whereas Irene either gave anyone connected to her a 0 or didn’t get around to doing her 360 reviews (which happens a lot, and the lack of connection is itself something key to visualize with datasets like this). This is a weighted network because the edges have a value. It’s a directed network because the edges have direction. Therefore, we have a weighted directed network, and we need to account for both weight and direction in our network visualizations.

Technically, you only need an edge list to create a network, because you can derive a list of nodes from the unique values in the edge list. This is done by traditional network analysis software packages like Gephi. Although you can derive a node list with JavaScript, it’s more common to have a corresponding node list that provides more information about the nodes in your network, like we have in the following listing.



Listing 7.2 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
```

## More info...

As these are employees, we have a bit more information about them besides their links—in this case, their role and their salary. As with the edge list, it's not necessary to have more than an ID. But having access to more data gives you the chance to modify your network visualization to reflect the node attributes.

We'll use role to color the later networks (managers in orange, employees in green, and contractors in purple). How you represent a network depends on its size and nature. If a network doesn't represent discrete connections between similar things, but rather the flow of goods or information or traffic, then you could use a Sankey diagram.

Recall that the data format for the Sankey is exactly the same as what we have here: a table of nodes and a table of edges. The Sankey diagram is only suitable for specific kinds of network data. Other chart types, such as an adjacency matrix, are more generically useful for network data.

Before we get started with code to create a network visualizations, let's put together a CSS page so that we can set color based on class and use inline styles as little as possible. Listing 7.3 gives the CSS necessary for all the examples. (Some exceptions)

**Listing 7.3 networks.css**

```
.grid {
  stroke: #9A8B7A;
  stroke-width: 1px;
  fill: #CF7D1C;
}
.arc {
  stroke: #9A8B7A;
  fill: none;
}
.node {
  fill: #EBD8C1;
  stroke: #9A8B7A;
  stroke-width: 1px;
}
circle.active {
  fill: #FE9922;
}
path.active {
  stroke: #FE9922;
}
circle.source {
  fill: #93C464;
}
circle.target {
  fill: #41A368;
}
```

**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**

## Adjacency matrix

As you see more and more networks represented graphically, it seems like the only way to represent a network is with a circle or square that represents the node and a line (whether straight or curvy) that represents the edge. It may surprise you that one of the most effective network visualizations has no connecting lines at all. Instead, the adjacency matrix uses a grid to represent connections between nodes, with the graphical rules of the chart as described in figure 7.3.

The principle of an adjacency matrix (a two-node example is seen in the figure) is simple: you place the nodes along the x-axis and then place the same nodes along the y-axis. If two nodes are connected, then the corresponding grid square is filled; otherwise, it's left blank. In our case, because it's a directed network, the nodes along the y-axis are considered the source, and the nodes along the x-axis are considered the target, as you'll see in a few pages. Because our people analytics network is also weighted, we'll use saturation to indicate weight, with lighter colors indicating a weaker connection and darker colors indicating a stronger connection.

## Adjacency matrix

The only problem with building an adjacency matrix in D3 is that it doesn't have an existing layout, which means you have to build it by hand like we did with the bar chart, scatterplot, and boxplot. Nice example from Mike B.

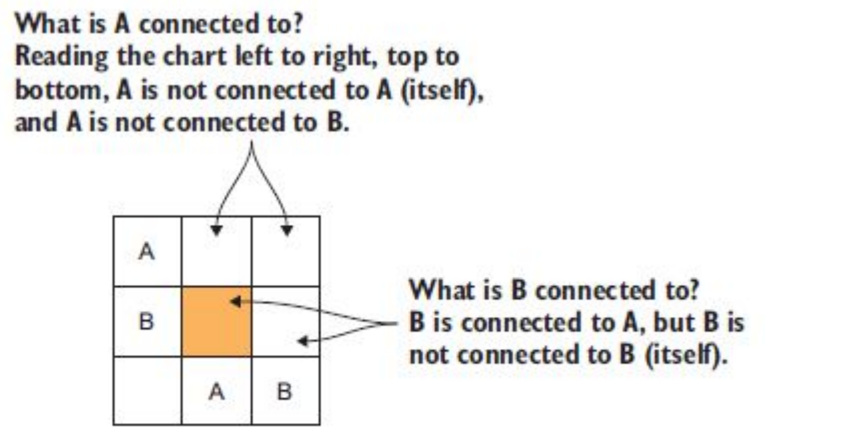

What is A connected to?
Reading the chart left to right, top to bottom, A is not connected to A (itself), and A is not connected to B.
A
B
A
B
What is B connected to?
B is connected to A, but B is not connected to B (itself).

**Figure 7.3** 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.

You can make something that's functional without too much code, which we'll do with the function in listing 7.4. In doing so, though, we need to process the two arrays of JavaScript objects that are created from our CSVs and format the data so that it's easy to work with. This is getting close to writing our own layout, something we'll do in chapter 10, and a good idea generally.

## Promises

One thing you'll notice in listing 7.4 that might intimidate you is the Promise API. Promises are asynchronous functions that fire a resolve or reject event when the asynchronous call finishes. We're not using them for fancy async behavior—we're using them so that we can fire *Promise.all*, which lets us pass an array of promises and only fires a function once all those promises have been resolved or one of them has been rejected. The simple promise wrapper we see in the listing 7.4 is how you might wrap a callback function like *d3.csv* so that it resolves as a promise. It's better to use core ES6 functionality like this, which you will run into in industry, than helper libraries like, say, *d3.queue*. I decided to use promises for any example where we need to wait for the asynchronous behavior of two or more functions because I think it's going to serve you best to get exposed to and familiar with promises rather than a D3-specific approach.

**Listing 7.4 The adjacency matrix function**

```
function adjacency() {
  var PromiseWrapper = d => new Promise(resolve => d3.csv(d, p => resolve(p)))

    Promise.all([PromiseWrapper("nodelist.csv"),
           PromiseWrapper("edgelist.csv")])
        .then(resolve => {

          createAdjacencyMatrix(resolve[0], resolve[1])
        })

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

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

We need to wrap our calls in promises to use promise.all

Promise.all returns an array of results in the order of the promises that were sent

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

Creates all possible source-target connections

Sets the xy coordinates based on the source-target array positions

If there's a corresponding edge in our edge list, give it that weight

```
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)
```

**Creates horizontal labels from the nodes**

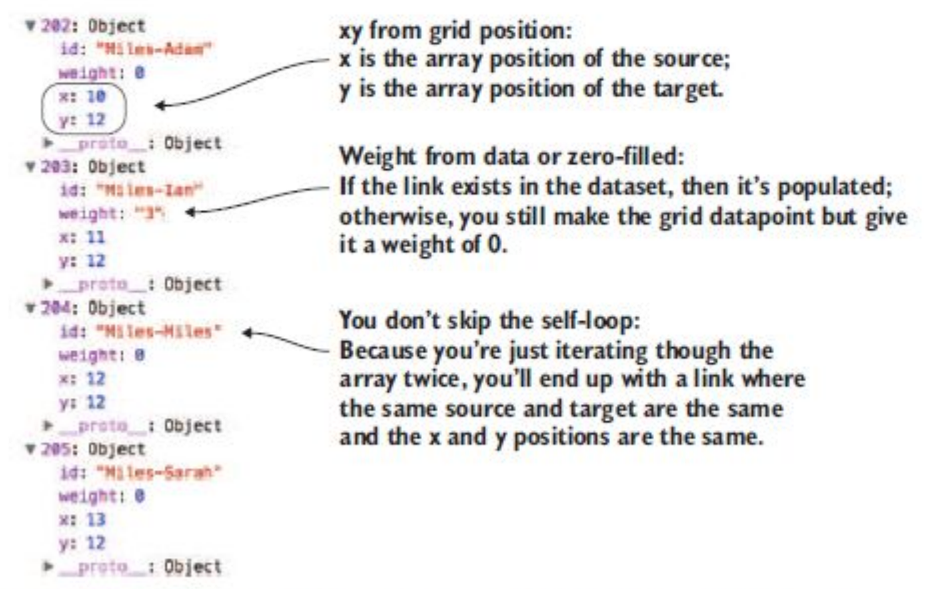
```
d3.select("svg")
  .append("g")
  .attr("transform", "translate(50,45)")
  .selectAll("text")
  .data(nodes)
  .enter()
  .append("text")
  .attr("x", (d,i) => i * 25 + 12.5)
  .text(d => d.id)
  .style("text-anchor", "middle")
```

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

```
d3.select("svg")
  .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")
  };
};
```

# Matrix Array

We're building this matrix array of objects that may seem obscure. But if you examine it in your console, you'll see, as in figure 7.4, that it's a list of every possible connection and the strength of that connection, if it exists.



```
▼ 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.

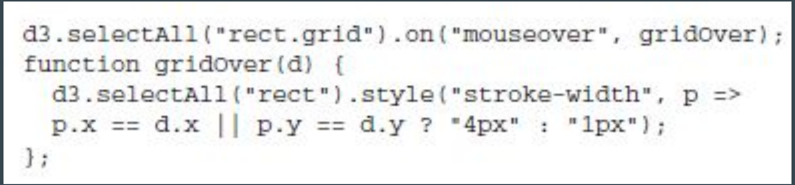
You don't skip the self-loop:
Because you're just iterating though the array twice, you'll end up with a link where the same source and target are the same and the x and y positions are the same.

Figure 7.4 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.

# Adjacency matrix

Figure 7.5 shows the resulting adjacency matrix based on the node list and edge list. You'll notice in many adjacency matrices that the square indicating the connection from a node to itself is always filled. In network parlance this is a self-loop, and it occurs when a node is connected to itself. In our case, it would mean that someone gave themselves positive feedback, and fortunately no one in our dataset is a big enough loser to do that.

If we want, we can add interactivity to help make the matrix more readable. 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. All we have to do is add a mouseover event listener that fires a gridOver function to highlight all rectangles that have the same x or y value:



```
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");
};
```

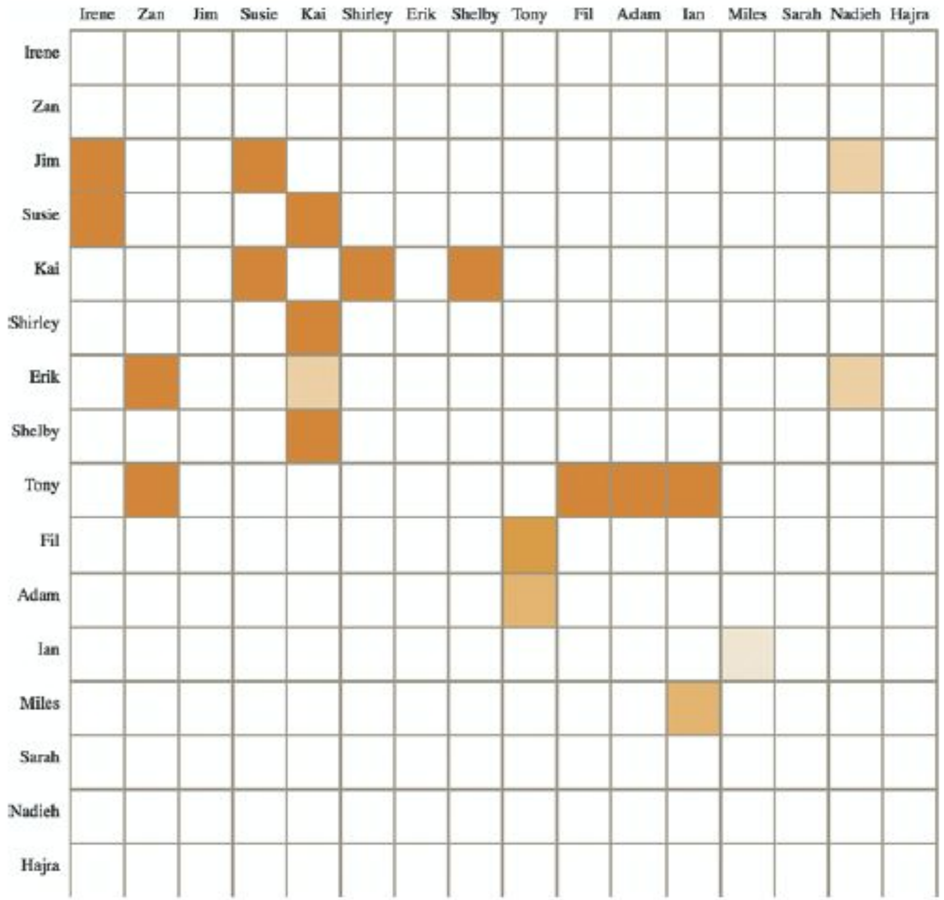


**Figure 7.5 A weighted, directed adjacency matrix where lighter orange indicates weaker connections and darker orange indicates stronger connections. The source is on the y-axis, and the target is on the x-axis. The matrix shows that Sarah, Nadieh, and Hajra didn't give anyone feedback, whereas Kai gave Susie feedback, and Susie gave Kai feedback (what we call a *reciprocated tie* in network analysis).**

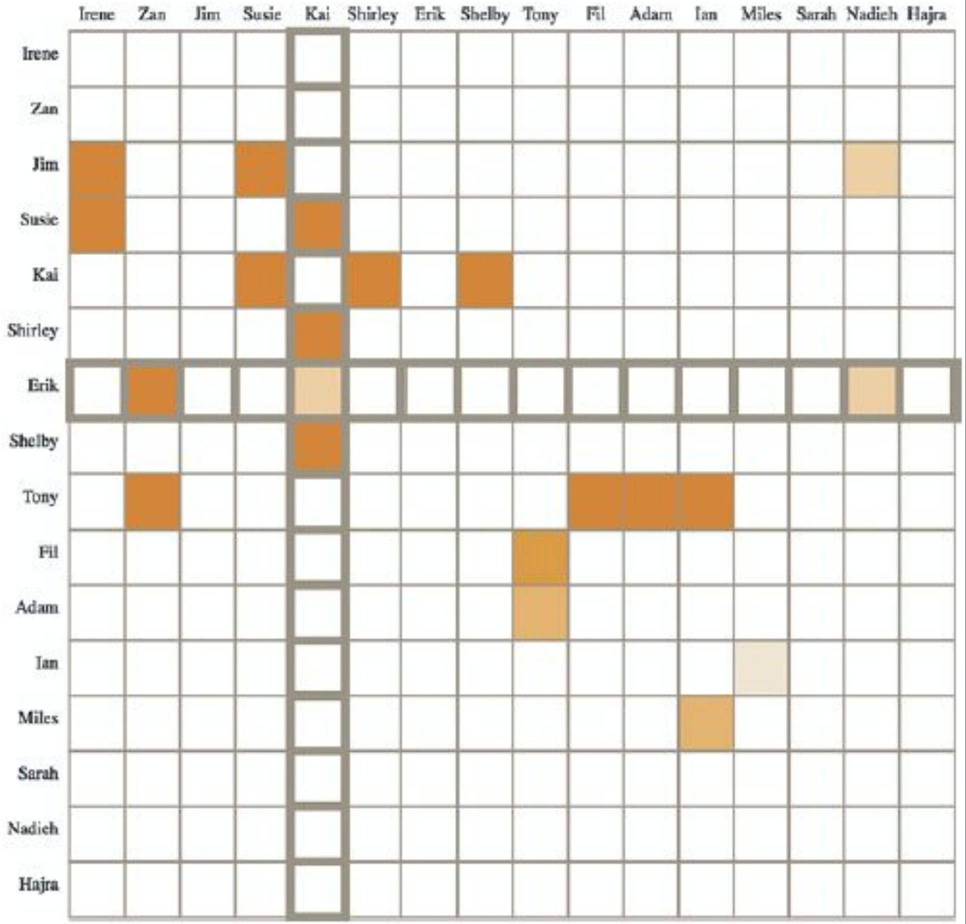


**Figure 7.6** Adjacency highlighting of the column and row of the grid square. In this instance, the mouse is over the Erik-to-Kai edge, and as a result highlights the Erik row and the Kai column. You can see that Erik gave feedback to three people, whereas Kai received feedback from four people.

## That’ll do folks!

## Arc diagram

Another way to graphically represent networks is by using an arc diagram. An arc diagram arranges the nodes along a line and draws the links as arcs above and/or below that line (as seen in figure 7.7). Whereas adjacency matrices let you see edge dynamics quickly, arc diagrams let you see node dynamics quickly. You can see which nodes are isolated and which nodes have many connections, as well as get a ready sense of the directionality of those connections.

Again, there isn't a layout available for arc diagrams, and there are even fewer examples, but the principle is rather simple after you see the code. We build another pseudo-layout like we did with the adjacency matrix, but this time we need to process the nodes as well as the links, as shown in listing 7.5.

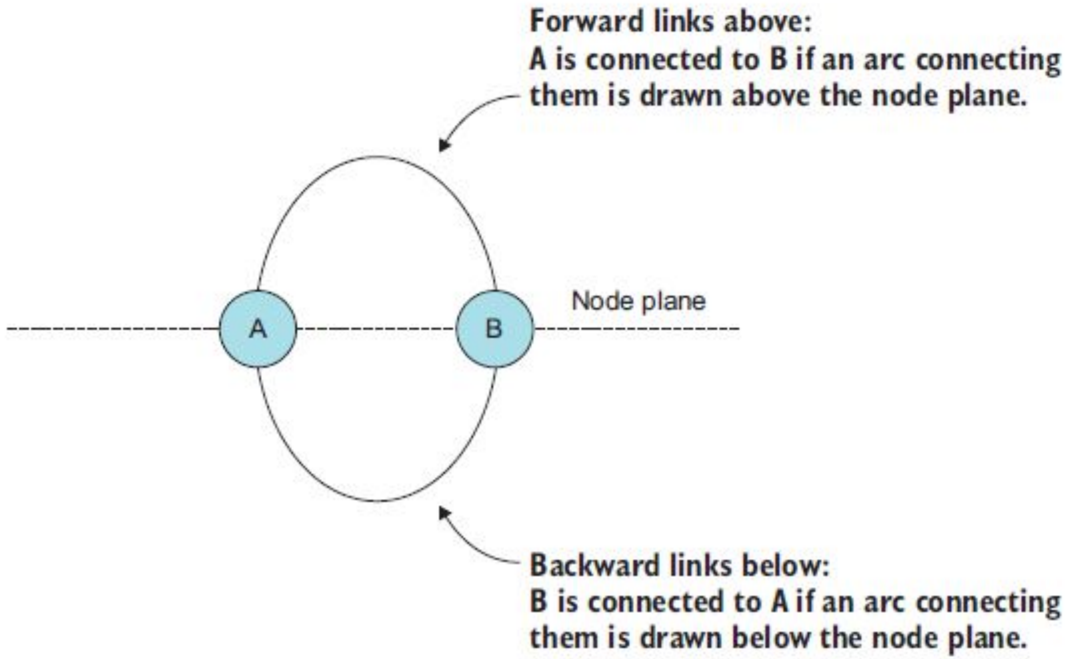

Forward links above:
A is connected to B if an arc connecting them is drawn above the node plane.
Node plane
A
B
Backward links below:
B is connected to A if an arc connecting them is drawn below the node plane.

**Figure 7.7** 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.

**Listing 7.5 Arc diagram code**

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

 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)

  function arc(d,i) {
    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]])
  }
}
```

Takes the results of the same Promise.all as adjacencyMatrix

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

Replaces the string ID of the node with a pointer to the JSON object

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

## Arc diagram

Notice that the edges array that we built uses a hash with the ID value of our edges to create object references. By building objects that have references to the source and target nodes, we can easily calculate the graphical attributes of the *<line>* or *<path>* element we're using to represent the connection. This is the same method used in the force layout that we'll look at later. The result of the code is your first arc diagram, shown in figure 7.8.

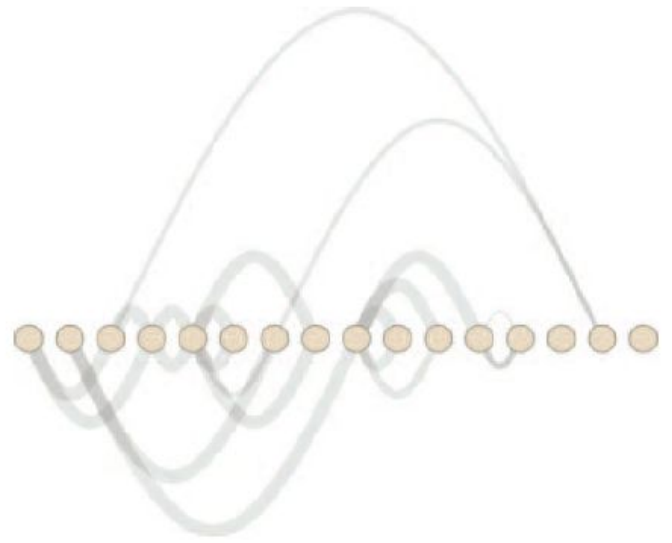


**Figure 7.8** An arc diagram, with connections between nodes represented as arcs above and below the nodes. We can see the first (left) two nodes have no outgoing links, and the rightmost three nodes also have no outgoing links. The length of the arcs is meaningless and based on how we've laid the nodes out (nodes that are far away will have longer links), but the width of the arcs is based on the weight of the connection.

## Arc diagram

With abstract charts like these, you're getting to the point where interactivity is no longer optional. Even though the links follow rules, and you're not dealing with too many nodes or edges, it can be hard to make out what's connected to what and how. You can add useful interactivity by having the edges highlight the connecting nodes on mouseover. You can also have the nodes highlight connected edges on mouseover by adding two new functions, as shown in listing 7.6, with the results in figure 7.9.

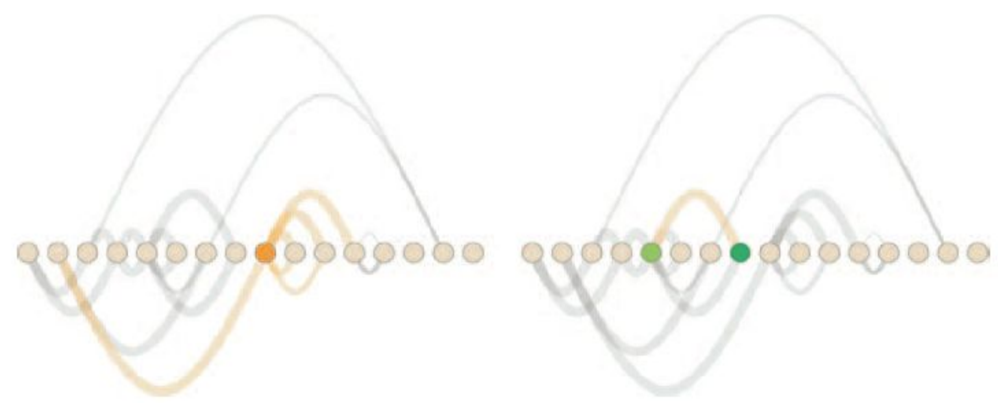
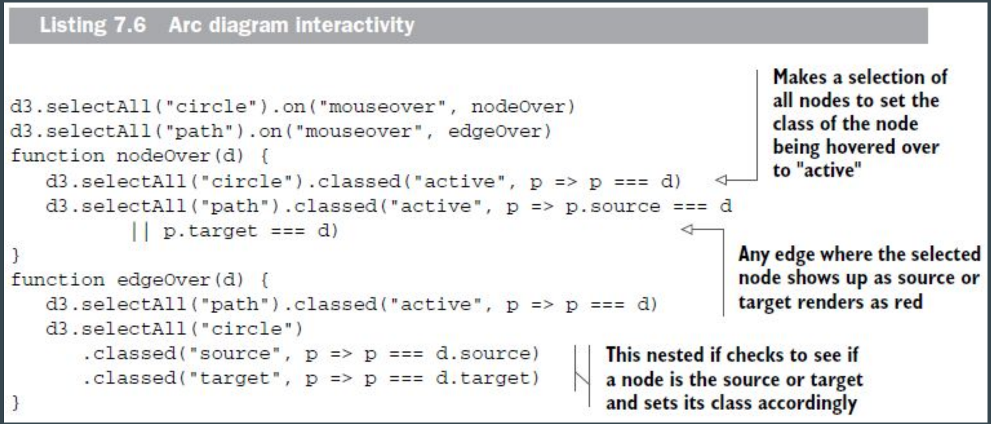


**Figure 7.9** Mouseover behavior on edges (right), with the edge being moused over in orange, the source node in light green, and the target node in dark green. Mouseover behavior on nodes (left), with the node being moused over in orange and the connected edges in light orange.

## Interactivity


**Listing 7.6 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)
   d3.selectAll("path").classed("active", p => p.source === d
          || p.target === d)
}
function edgeOver(d) {
   d3.selectAll("path").classed("active", p => p === d)
   d3.selectAll("circle")
      .classed("source", p => p === d.source)
      .classed("target", p => p === d.target)
}
```

Makes a selection of all nodes to set the class of the node being hovered over to "active"

Any edge where the selected node shows up as source or target renders as red

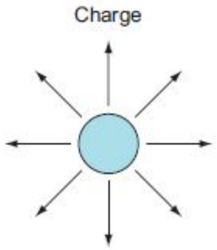
This nested if checks to see if a node is the source or target and sets its class accordingly

If you're interested in exploring arc diagrams further and want to use them for larger datasets, you'll also want to look into hive plots, which are arc diagrams arranged on spokes. We won't deal with hive plots in this book, but there's a plugin layout for hive plots that you can see at link. Both the adjacency matrix and arc diagram benefit from the control you have over sorting and placing the nodes, as well as the linear manner in which they're laid out. The next method for network visualization, which is our focus for the rest of the chapter, uses entirely different principles for determining how and where to place nodes and edges.

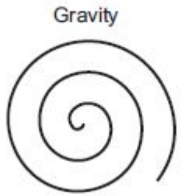
## Force-directed layout

The force layout gets its name from the method by which it determines the most optimal graphical representation of a network (yet another instance of bad naming in data visualization). Like the word cloud and the Sankey diagram, the force() layout dynamically updates the positions of its elements to find the best fit. Unlike those layouts, it does it continuously in real time rather than as a preprocessing step before rendering. The principle behind a force layout is the interplay between three forces, shown in figure 7.10. These forces push nodes away from each other, attract connected nodes to each other, and keep nodes from flying out of sight.

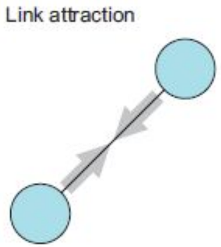
In this section, you'll learn how force-directed layouts work, how to make them, and general principles from network analysis that will help you better understand them. You'll also learn how to add and remove nodes and edges, as well as adjust the settings of the layout on the fly.



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.

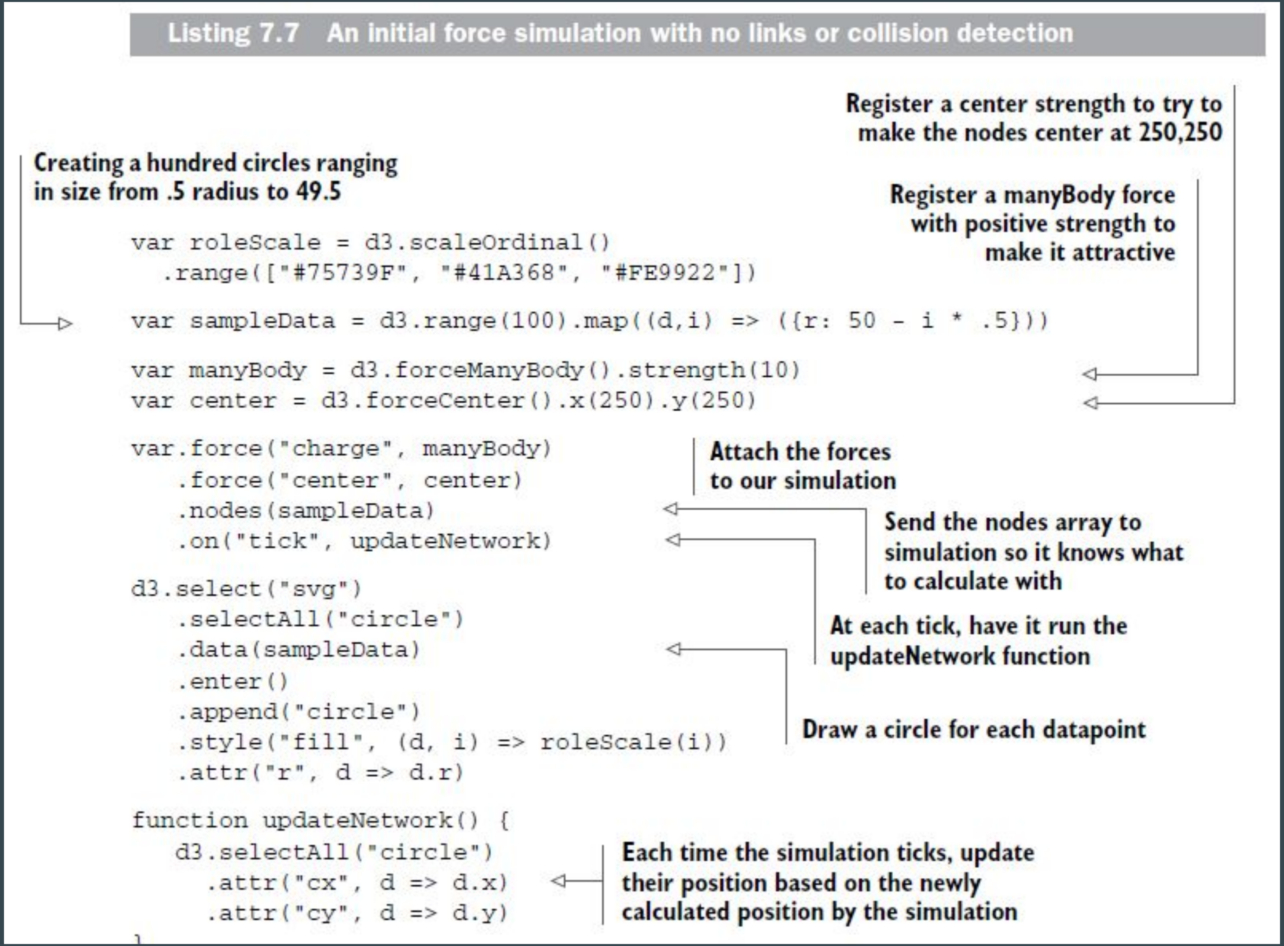


Nodes that are connected to each other are pulled toward each other. Sometimes, this force is based on the strength of connection, so that more strongly connected nodes are closer. In D3, this is defined using d3.forceCenter for the "link" force.

**Figure 7.10** 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.

# Playing with forces

Your first implementation of forceSimulation is going to be pretty unimpressive, though, with results looking something like figure 7.11. The circles will bounce around a bit and finally settle on top of each other—which, if you think about it, is exactly what you might expect if you made all the circles attractive to each other.



**Listing 7.7 An initial force simulation with no links or collision detection**

Register a center strength to try to make the nodes center at 250,250

Creating a hundred circles ranging in size from .5 radius to 49.5

Register a manyBody force with positive strength to make it attractive

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

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

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

var.force("charge", manyBody)
   .force("center", center)
   .nodes(sampleData)
   .on("tick", updateNetwork)

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

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

Attach the forces to our simulation

Send the nodes array to simulation so it knows what to calculate with

At each tick, have it run the updateNetwork function

Draw a circle for each datapoint

Each time the simulation ticks, update their position based on the newly calculated position by the simulation

## Force simulation - attraction

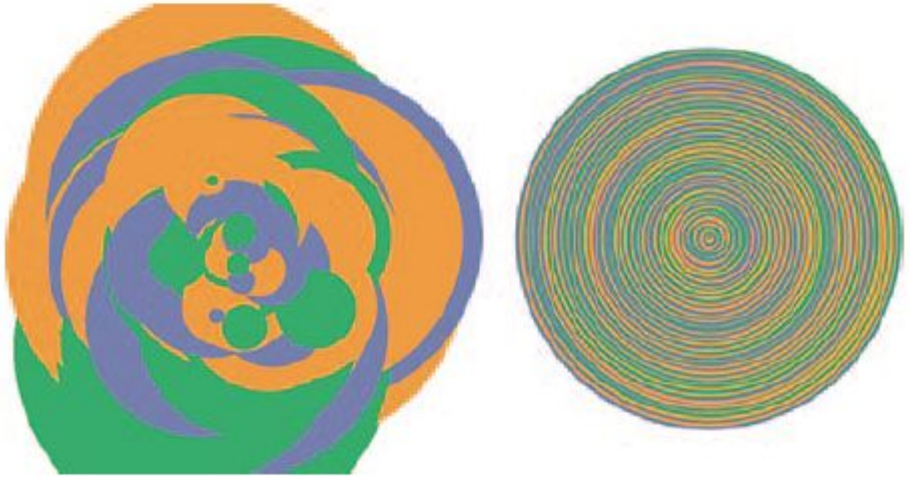


**Figure 7.11** The results of a force simulation where the only force acting on the nodes is attraction

## Force simulation - collision

To make this a bit more interesting, let’s register a “collide” force using *d3.force-Collide* and setting it to base the collision detection off the size of each node (its .r attribute):

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

With this in place, we get a simple bubble chart of our data, as we see in figure 7.12.

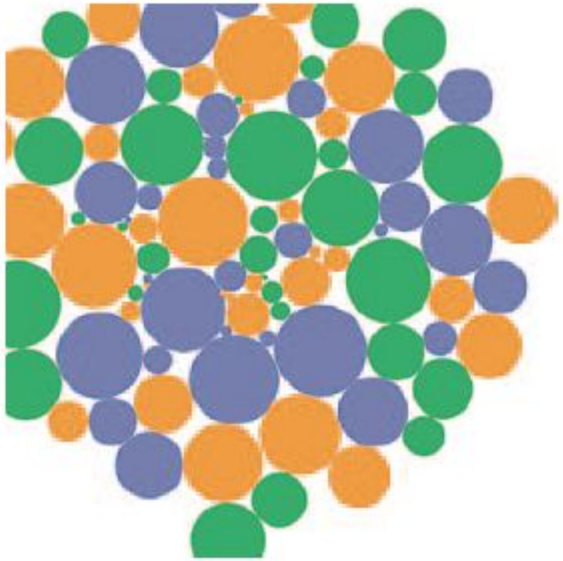
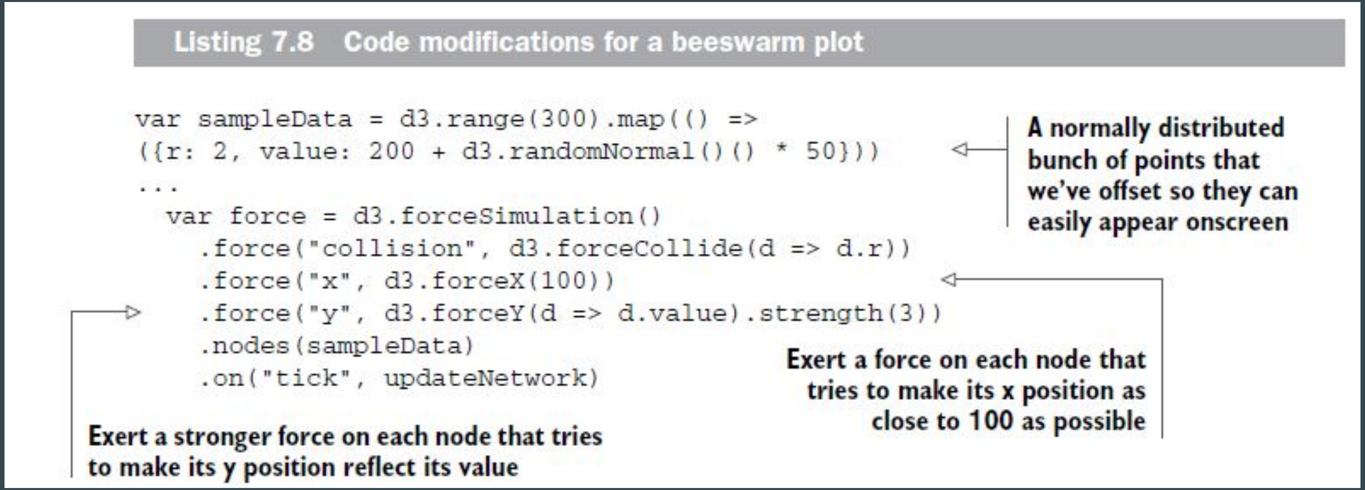


**Figure 7.12** **Our sample node data laid out with collision detection. This is one way to create a simple bubble chart.**

## Beeswarm plot

The last thing we want to look at is using x- and y-constraints to lay out our nodes. If we replace our random data with normally distributed random data and add an x-constraint to keep it in a line and a y-constraint to have its y position correspond to its value, we can produce a beeswarm plot, as in the following listing.



**Listing 7.8 Code modifications for a beeswarm plot**

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

A normally distributed bunch of points that we've offset so they can easily appear onscreen

Exert a force on each node that tries to make its x position as close to 100 as possible

Exert a stronger force on each node that tries to make its y position reflect its value

## Beeswarm plot

The result of your simulation this time tries to arrange each node in a way that they're laid out along a shared x-axis but positioned to show their value. This beeswarm plot, as you see in figure 7.13, is pretty popular and allows you to show distributions while maintaining individual sample points.

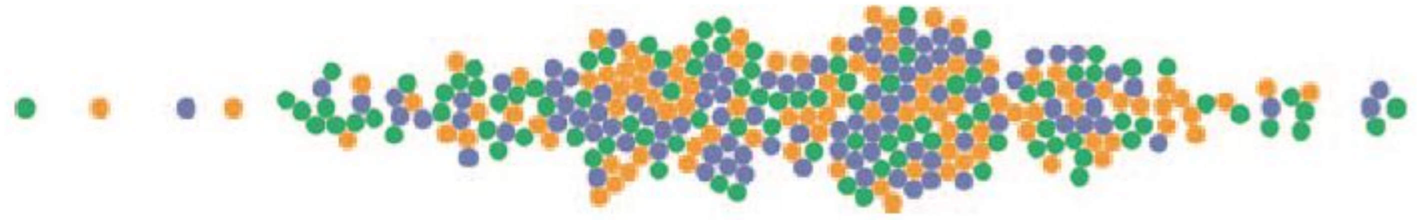


**Figure 7.13 A beeswarm plot created with our code (rotated to better fit on the page)**

## Creating a force-directed network diagram

The forceSimulation()layout that you've been using and which you see initialized in listing 7.9 has even more settings. The nodes() method that we already used is similar to the one you saw in the Sankey layout in chapter 5, but links in forceSimulation are registered with a "link" force that takes, as you'd expect, the settings for how the links describe source and target as well as an array of those links. We need to take the links from edges.csv and change the source and target into objects like we did with the arc diagram. That's the formatting that forceSimulation() expects. It also accepts integer values where the integer values correspond to the array position of a node in the nodes array, like the formatting of data for the Sankey diagram links array from chapter 5. Other than the links force, the only new setting we have is to use forceManyBody with a negative value, meaning that nodes will push each other away. This will result in connected nodes attracted to connected nodes and create the kind of network diagram people are familiar with.

**Listing 7.9 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()

    var simulation = d3.forceSimulation()
     .force("charge", d3.forceManyBody().strength(-40))
     .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}`)
      .enter()
      .append("line")
      .attr("class", "link")
      .style("opacity", .5)
      .style("stroke-width", d => d.weight);

   var nodeEnter = d3.select("svg").selectAll("g.node")
      .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);

   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})`)
   }
}
```

**How much each node pushes away each other—if set to a positive value, nodes attract each other**

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

The animated nature of the force layout is lost on the page, but you can see in figure 7.14 general network structure that's less prominent in an adjacency matrix or arc diagram. It's readily apparent that there are dense and sparse parts of the network, with key brokers like Zan who connect two different groups. We can also see that two people aren't connected to anyone, having neither given nor received feedback. The only reason those nodes are still onscreen is because the layout's gravity pulls unconnected pieces toward the center.

We can see that our two managers both gave feedback to only two people, but that they have different positions in the structure of our two teams. If Irene quit tomorrow, there wouldn't be much change in this network, but if Zan quit, then the two teams wouldn't have any communication with each other.

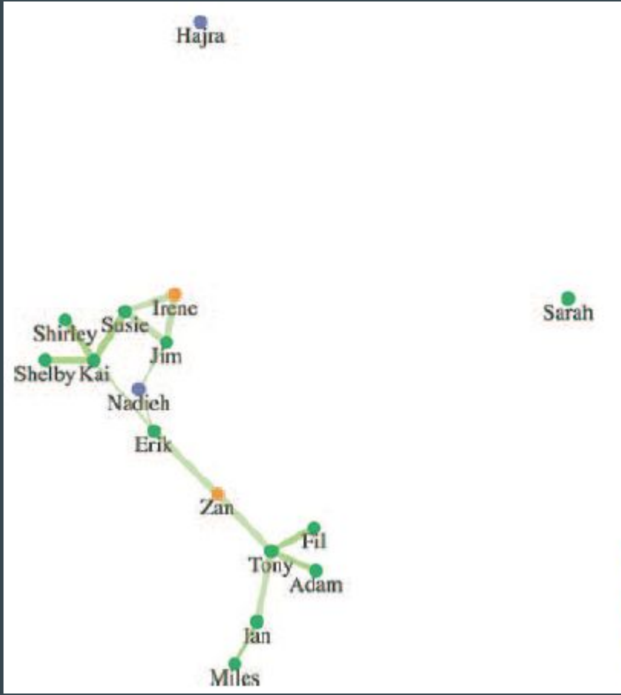


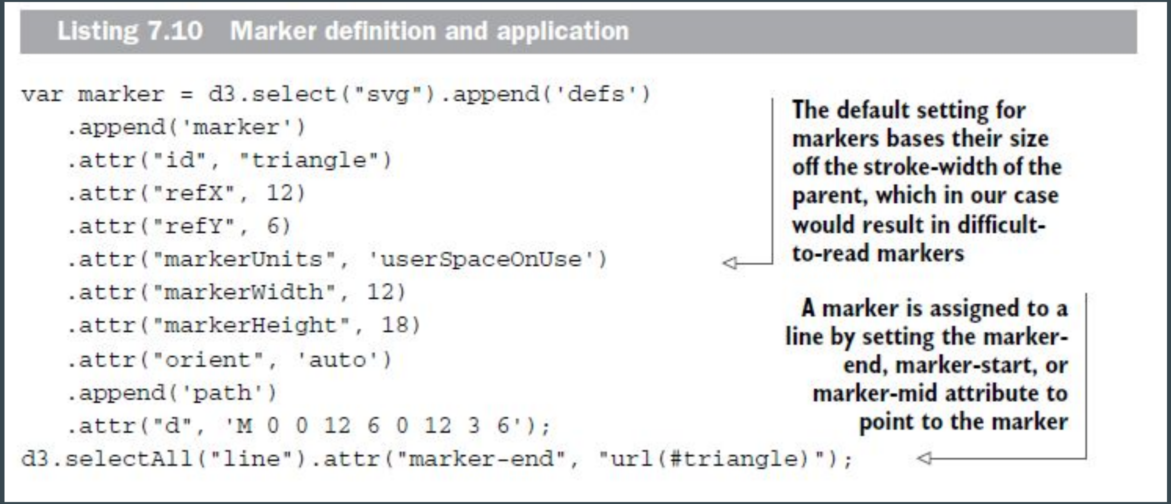
**Figure 7.14** A force-directed layout based on our dataset and organized graphically using default settings in the force layout. Managers are in orange, employees green, and contractors purple.

## SVG markers

The thickness of the lines corresponds to the strength of connection. But although we have edge strength, we’ve lost the direction of the edges in this layout. You can tell that the network is directed only because the links are drawn as semitransparent, so  you can see when two links of different weights overlap each other. We need to use a method to show if these links are to or from a node. One way to do this is to turn our lines into arrows using SVG markers.

Sometimes you want to place a symbol, such as an arrowhead, on a line or path that you’ve drawn. In that case, you have to define a marker in your svg:defs and then associate that marker with the element on which you want it to draw. You can define your marker statically in HTML or create it dynamically like any SVG element, as we’ll do next. The marker we define can be any sort of SVG shape, but we’ll use a path because it lets us draw an arrowhead. A marker can be drawn at the start, end, or middle of a line, and has settings to determine its direction relative to its parent element. See the following listing.

# SVG Markers


**Listing 7.10 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')
    .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)");
```

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

A marker is assigned to a line by setting the marker-end, marker-start, or marker-mid attribute to point to the marker

With the markers defined in listing 7.10, you can now read the network (as shown in figure 7.15) more effectively. You see how the nodes are connected to each other and you can spot which nodes have reciprocal ties with each other (where nodes are connected in both directions). Reciprocation is important to identify, because there's a big difference between people who favorite Katy Perry's tweets and people whose tweets are favorited by Katy Perry (the current Twitter user with the most followers). Direction of edges is important, but you can represent direction in other ways, such as using curved edges or edges that grow fatter on one end than the other. To do something like that, you'd need to use a *<path>* rather than a *<line>* for the edges like we did with the Sankey layout or the arc diagram.

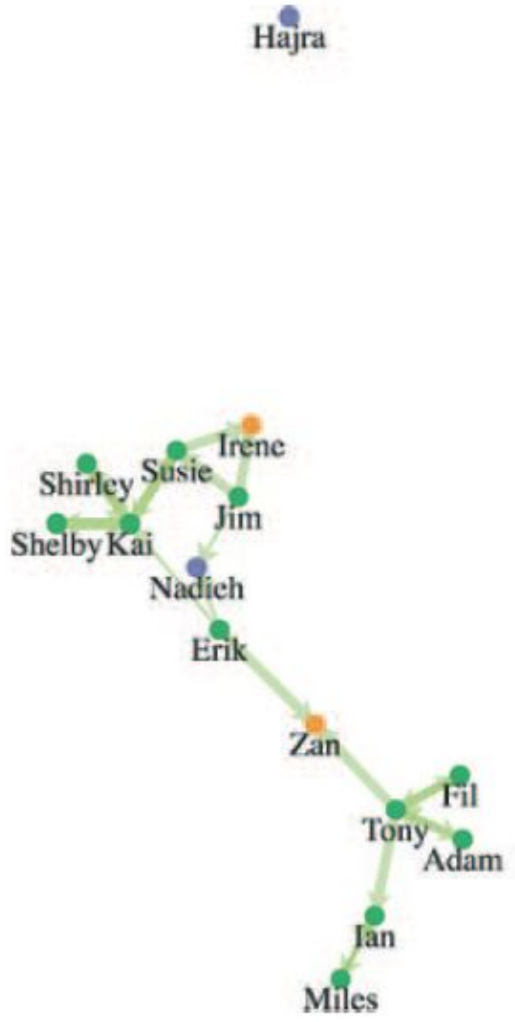


**Figure 7.15 Edges now display markers (arrowheads) indicating the direction of connection. Notice that all the arrowheads are the same size. You can control the color of the arrowheads by using CSS rules such as `marker > path {fill: # 93C464;}`.**

## Basemap

If you’ve run this code on your own, your network should look exactly like figure 7.15. That’s because even though network visualizations created with force-directed layouts are the result of the interplay of forces, D3’s force simulation is deterministic as long as the inputs don’t change. However, if your network inputs are constantly changing, one way to help your readers is to generate a network using a forcedirected layout and then fix it in place to create a network basemap. You can then apply any later graphical changes to that fixed network. The concept of a basemap comes from geography and in network visualization refers to the use of the same layout with differently sized and/or colored nodes and edges. It allows readers to identify regions of the network that are significantly different according to different measures. You can see this concept of a basemap in use in figure 7.16, which shows how one network can be measured in multiple ways.

The force-directed layout provides the added benefit of seeing larger structures. Depending on the size and complexity of your network, they may be enough. But you may need to represent other network measurements when working with network data.

## Infoviz term: hairball

Network visualizations are impressive, but they can also be so complex that they’re unreadable. For this reason, you’ll encounter critiques of networks that are too dense to be readable. These network visualizations are often referred to as hairballs due to extensive overlap of edges that make them resemble a mass of unruly hair.

If you think a force-directed layout is hard to read, you can pair it with another network visualization such as an adjacency  matrix and highlight both as the user navigates either visualization. You’ll see techniques for pairing visualizations like this in chapter 11.

## References

All content is taken verbatim from Chapter 7 of D3.js in Action 2nd Edition