[1]:

```python
# make figures better:
import matplotlib
%matplotlib inline
font = {'weight':'normal','size':16}
matplotlib.rc('font', **font)
matplotlib.rc('figure', figsize=(9.0, 6.0))
matplotlib.rc('xtick.major', pad=8) # xticks too close to border!
matplotlib.rc('ytick.major', pad=10) # xticks too close to border!
matplotlib.rc('xtick',labelsize='small')
matplotlib.rc('ytick',labelsize='small')

import warnings
warnings.filterwarnings('ignore')

import random, math
import numpy as np
import scipy, scipy.stats
import pandas as pd

import matplotlib.pyplot as plt
nicered = "#E6072A"
niceblu = "#424FA4"
nicegrn = "#6DC048"
```

DS1 Lecture 21 - part 2

James Bagrow, james.bagrow@uvm.edu, http://bagrow.com

Graphs

Graph drawing

We want to understand, at least at a basic level, how to **draw** a graph. What does this mean exactly?

Given the graph topology (meaning the nodes and edges), we want to put those nodes into position in a graphic in a pleasing and/or meaningful way.

- We need the (x,y)-coordinates of the nodes! Obtaining these positions is the goal of drawing.

Let's build a little drawing system ourselves, to see it **firsthand**.

Preliminary tools

Before we start let's make two things:

- A function that takes a graph and the node positions and draws the network with circles and lines
- A random graph or two to play around with.

Random graph Let's start with the **random graph**. The simplest random graph is the Erdős–Rényi or ER graph:

An ER graph is specified by two parameters, N the number of nodes and p the probability for a link. What does this mean? To build it:

1. Start from an empty graph of N nodes and zero links

2. For **every** pair of nodes (i, j) $(i \neq j)$ and **flip a coin**:
   - With probability p a new link is added between i and j,
   - With probability $1 - p$ no link is added.

If $p = 0$ we're stuck with the empty graph, whereas if $p = 1$ we have a complete graph with every possible link.

ER graphs are relatively basic, and unrealistic it turns out, but they have some neat properties worth digging into if you're interested. For our purposes here, they serve as a handy way to build **fake data** for drawing.

- We will also draw a couple of other, non-random graphs.

Networkx has ER graphs built in:

[2]:

```python
import networkx as nx

G = nx.erdos_renyi_graph(30,0.1)

print(nx.info(G))
```

```
Name:
Type: Graph
Number of nodes: 30
Number of edges: 40
Average degree:   2.6667
```

Drawing function Now we turn to the drawing function. You would think this would be the whole shebang, but remember, this function takes the node positions as **input**. We still need to compute those positions somehow.

To draw a graph given node → (x, y)-coordinates:

1. Loop over all edges, and draw a line between the nodes connected by the edge. Specifically for edge (i, j), the line goes from (x_i, y_i) to (x_j, y_j) where the subscript denotes the node.

[3]:

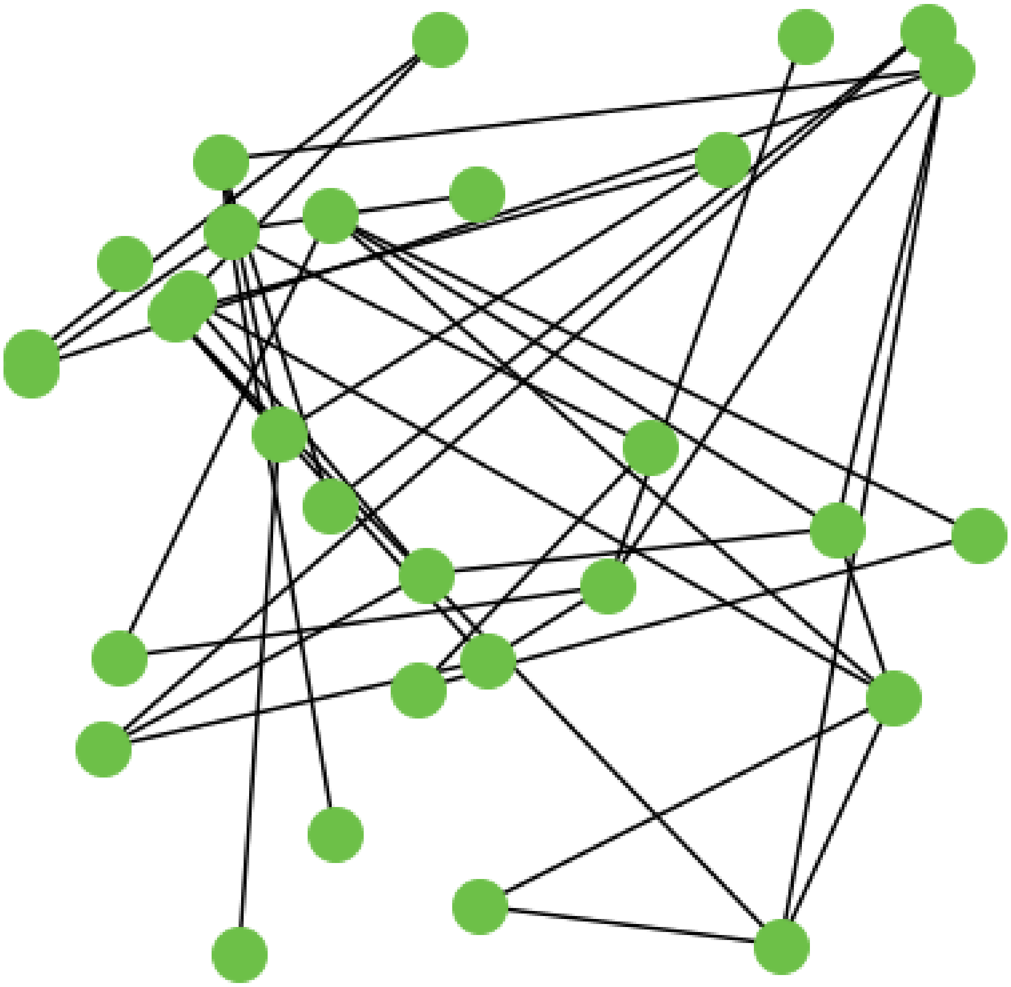
```python
def draw_graph(graph, posX,posY):
    """`graph` is a nx.Graph. `posX` and `posY` are dicts mapping node
    to x- or y-coordinate.

    Returns nothing, but plots the graph.
    """
    plt.figure(figsize=(7,7))

    # first plot edges as black lines:
    for i,j in graph.edges():
        xi = posX[i]
        xj = posX[j]
        yi = posY[i]
        yj = posY[j]

        plt.plot( [xi,xj], [yi,yj], 'k-', lw=1.5)

    # now plot nodes:
    nodes = graph.nodes()
    Xs = [ posX[n] for n in nodes ]
    Ys = [ posY[n] for n in nodes ]
    plt.plot(Xs,Ys, 'o', ms=20, c=nicegrn, clip_on=False)

    plt.axis("off")
    plt.show()
```

To test this out we need a graph (check! got the ER) and some positions (darn!).

- Let's use random positions first…

[4]:

```
# build random initial positions
n2x = {} # x-coords
n2y = {} # y-coords
for n in G:
    n2x[n] = random.random()
    n2y[n] = random.random()


draw_graph( G, n2x,n2y )
```



The picture seems to be working, but is that layout (meaning the coordinates of the nodes) any good?

Drawing graphs

To make a good picture of the network we need some guidelines.

- What do we want to see?
- How can we maximize our understanding?

I propose the following:

- Nearby nodes in the graph (like neighbors) should generally be placed spatially nearby in the drawing so the graph topology is emphasized.
- Nodes should not overlap so we can see what's happening.
- Links should cross as little as possible to make it easy to visually trace out paths on the graph.

Is it possible to meet all these criteria? Probably not. But the better we do the better the graph should be...

With these guidelines and the goal of finding node coordinates, graph drawing is now a (relatively) precise problem. It has a rich history (40+ years) and there exist many methods for trying to lay out a graph.

- GD 2014, the 22nd international symposium on graph drawing: http://gd2014.informatik.uni-wuerzburg.de/
- http://www.graphdrawing.org/

Force-directed layout

We're going to build perhaps the simplest and one of the earliest layout methods.

- How does it work? *Physics!*

[To the board!]

Whew! OK to code this up is a little non-trivial. Remember how I always say how important it is to keep some paper with you for scratch work while programming? Here are my actual notes while building the layout code (most of it was written before I even programmed a thing):

I try to organize what needs to be made before I get deep into the code, working out the big picture. I also try to develop the variable names before hand so I can use simpler names.

The code

First, build an even simpler graph, a 2D lattice.

- If we can't draw this well, we are in trouble!

[5]:

```
G = nx.grid_2d_graph(5,5)
```

Now let's get random coordinates, like before:

[6]:

```
# build random initial positions
n2x = {} # x-coords
n2y = {} # y-coords
for n in G:
    n2x[n] = random.random()
    n2y[n] = random.random()
```

A function to return the spatial distance squared d_{ij}^2 between a pair of points.

- I say "spatial distance" even though that sounds redundant because we could also be talking about "graph distance" (shortest path).

[7]:

```
def distance_sqrd(pix,piy, pjx,pjy):
    # See also, math.hypot
    return (pjx-pix)**2 + (pjy-piy)**2
```

First, we need the constants and stopping criteria:

[8]:

```
k_r = 0.02 # finding these values takes some
k_a = 0.12 # back and forth tuning
k_n = 0.05

c_stop = 3e-4
t_stop = 5000
```

Now we come to it at last.

[9]:

```
t = 0
avg_change = 1.0
while avg_change > c_stop: # keep updating until
    t += 1

    # store forces for this move:
    n2Fx = { n:0.0 for n in n2x }
    n2Fy = { n:0.0 for n in n2y }

    # get net force on each node i:
    for i in G:

        # get repulsive forces:
        for j in G:
            if j == i:
                continue
            dij2 = distance_sqrd( n2x[i],n2y[i], n2x[j],n2y[j] )

            # magnitude of the force:
            Fij_r = k_r / dij2 # coulomb in da house!!

            # direction of the force:
            th_ij = math.atan2( n2y[j]-n2y[i], n2x[j]-n2x[i] ) # radians
            th_ij += math.pi # 180 degrees since repulsion

            # add onto the x and y components of the force on i:
            n2Fx[i] += math.cos(th_ij) * Fij_r
            n2Fy[i] += math.sin(th_ij) * Fij_r


        # now get attractive forces:
        for j in G.neighbors(i):
            dij2 = distance_sqrd( n2x[i],n2y[i], n2x[j],n2y[j] )

            # magnitude of the force:
            Fij_a = k_a * math.sqrt(dij2) # hooke for the win!

            # direction of the force:
            th_ij = math.atan2( n2y[j]-n2y[i], n2x[j]-n2x[i] ) # radians


            # add on the x and y components of the force:
            n2Fx[i] += math.cos(th_ij) * Fij_a
            n2Fy[i] += math.sin(th_ij) * Fij_a

    # forces found, update positions:
    total_change = 0.0
    for i in G:
        dx = k_n * n2Fx[i]
        dy = k_n * n2Fy[i]

        n2x[i] = n2x[i] + dx
        n2y[i] = n2y[i] + dy

        total_change += math.sqrt(dx**2 + dy**2)
    avg_change = total_change / len(n2Fx)

    # emergency bail out:
    if t > t_stop:
        break

print("Converged in %i steps." % t    )
```

```
Converged in 953 steps.
```

Let's take a peek at the x-coordinates

[10]:

```
n2x
```

[10]:

```
{(0, 0): 2.3373611287258362,
 (0, 1): 1.7049115586934895,
 (0, 2): 0.9346339142983456,
 (0, 3): 0.1655443255088499,
 (0, 4): -0.48142346143845915,
 (1, 0): 1.9373008282128974,
 (1, 1): 1.3493196550945494,
 (1, 2): 0.6727558516271391,
 (1, 3): -0.0010333948487539425,
 (1, 4): -0.39347298791778973,
 (2, 0): 1.129898148580826,
 (2, 1): 0.6866563012237122,
 (2, 2): 0.21171312466628253,
 (2, 3): -0.08642707930484148,
 (2, 4): 0.31561214981761054,
 (3, 0): -0.49707663124598406,
 (3, 1): -0.285835763394077757,
 (3, 2): 0.1414936526308415,
 (3, 3): 0.627319447049102,
 (3, 4): 1.192114801285551,
 (4, 0): -1.1810513935226656,
 (4, 1): -0.7251714294421193,
 (4, 2): 0.011351965056905779,
 (4, 3): 0.809371420041877,
 (4, 4): 1.4710525749116123}
```

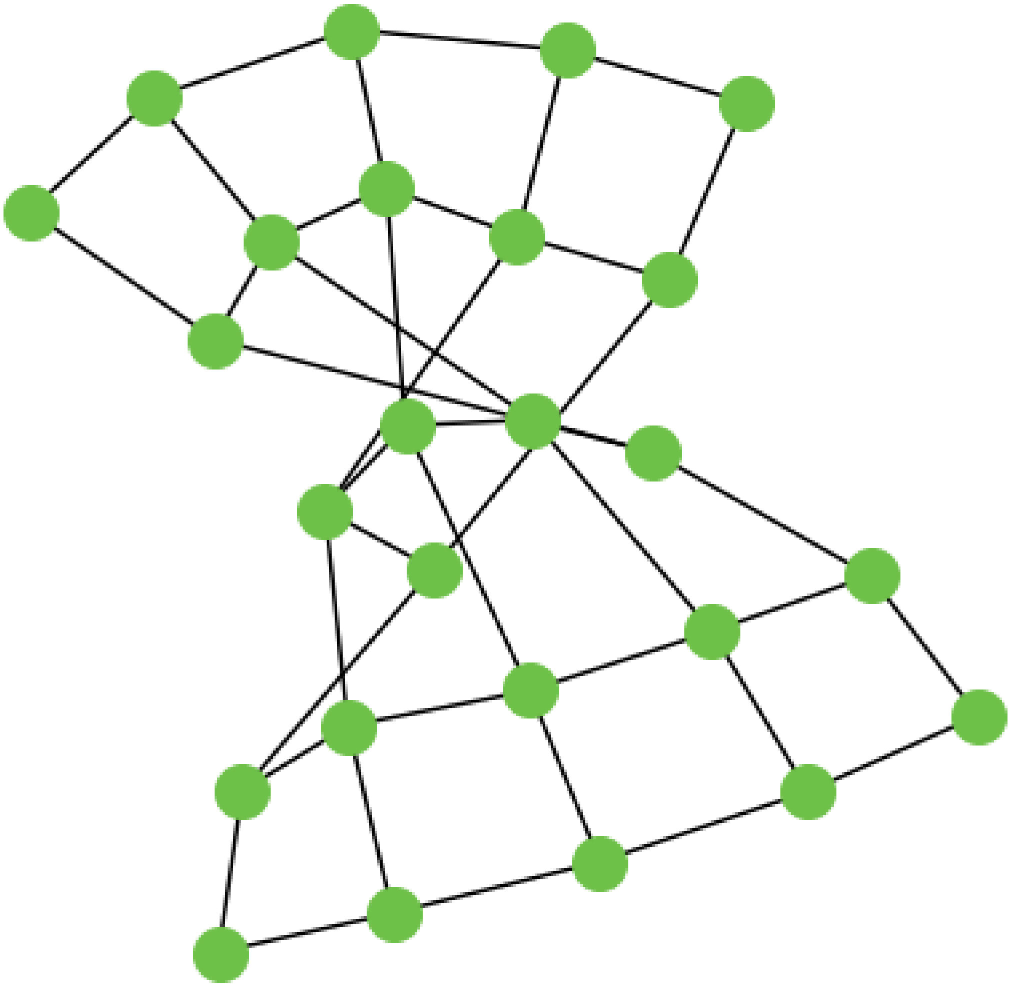
The initial values of x were inside $[0, 1]$, so it looks like nothing **flew off towards infinity**.

- This can actually happen, so taking a peek at the coordinates is a useful debugging step!

OK, the moment of truth:

[11]:

```
draw_graph(G, n2x,n2y)
```



And there was much rejoicing!

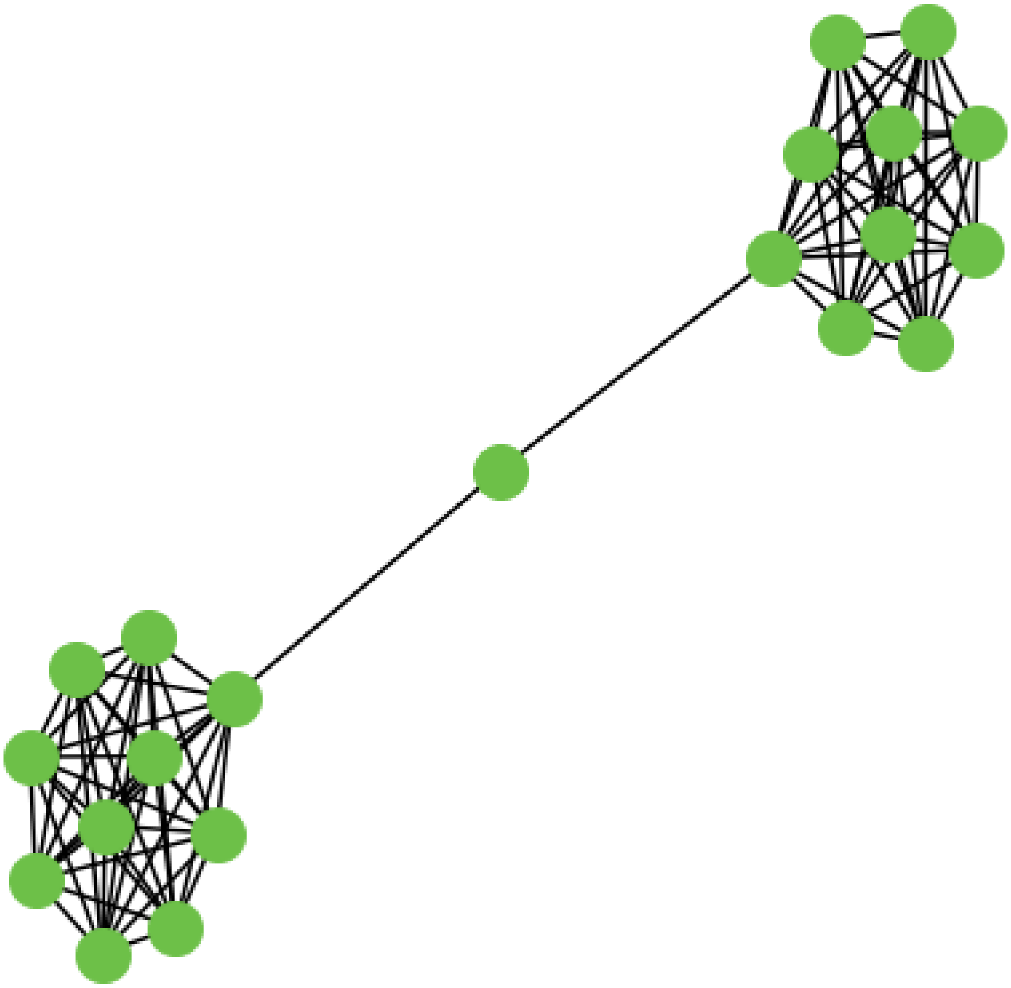
Let's bundle the drawing up into a reusable function:

- There are a lot of parameters, so a good docstring is extra important!

[12]:
```
def layout_graph(graph, k_r=0.02, k_a=0.12, k_n=0.05,
                        c_stop=3e-4, t_stop=5000  ):
    """Get the coordinates for a 2D drawing of a network.

    Inputs:
        graph  - a nx.Graph()
        k_r    - repulsive force constant (coulomb), default 0.02
        k_a    - attractive force constant (hooke), default 0.12
        k_n    - distance a node moves in one time step for a
                 one-unit net force, default 0.05
        c_stop - stop updating layout if average change in distance
                 per node is <= c_stop, default 3e-4
        t_stop - stop updating layout if number of timesteps on
                 update > t_stop, default 5000.

    Output:
        two dicts, mapping node to x- or y-coordinate.

    WARNING: make sure `graph` is globally connected.
    """

    # begin with random coordinates:
    n2x = {} # x-coords
    n2y = {} # y-coords
    for n in graph:
        n2x[n] = random.random()
        n2y[n] = random.random()

    t = 0
    avg_change = 1.0
    while avg_change > c_stop:
        t += 1

        # store forces:
        n2Fx = { n:0.0 for n in n2x }
        n2Fy = { n:0.0 for n in n2y }

        # get net force on each node:
        for i in graph:

            # get repulsive forces:
            for j in graph:
                if j == i:
                    continue
                dij2 = distance_sqrd( n2x[i],n2y[i], n2x[j],n2y[j] )

                # magnitude of the force:
                Fij_r = k_r / dij2 # coulomb in da house
                # direction of the force:
                th_ij = math.atan2( n2y[j]-n2y[i], n2x[j]-n2x[i] ) # radians
                th_ij += math.pi # 180 degrees since repulsion

                # add on the x and y components of the force:
                n2Fx[i] += math.cos(th_ij) * Fij_r
                n2Fy[i] += math.sin(th_ij) * Fij_r


            # get attractive forces:
            for j in graph.neighbors(i):

                dij2 = distance_sqrd( n2x[i],n2y[i], n2x[j],n2y[j] )

                # magnitude of the force:
                Fij_a = k_a * math.sqrt(dij2) # hooke for the win!

                # direction of the force:
                th_ij = math.atan2( n2y[j]-n2y[i], n2x[j]-n2x[i] ) # radians


                # add on the x and y components of the force:
                n2Fx[i] += math.cos(th_ij) * Fij_a
                n2Fy[i] += math.sin(th_ij) * Fij_a

        # update positions:
        total_change = 0.0
        for i in graph:
            dx = k_n * n2Fx[i]
            dy = k_n * n2Fy[i]

            n2x[i] = n2x[i] + dx
            n2y[i] = n2y[i] + dy

            total_change += math.sqrt(dx**2 + dy**2)
        avg_change = total_change / len(n2Fx)

        # emergency bail out:
        if t > t_stop:
            break

    return n2x,n2y
```

Now we can pretty productive. Let's check out some more examples:

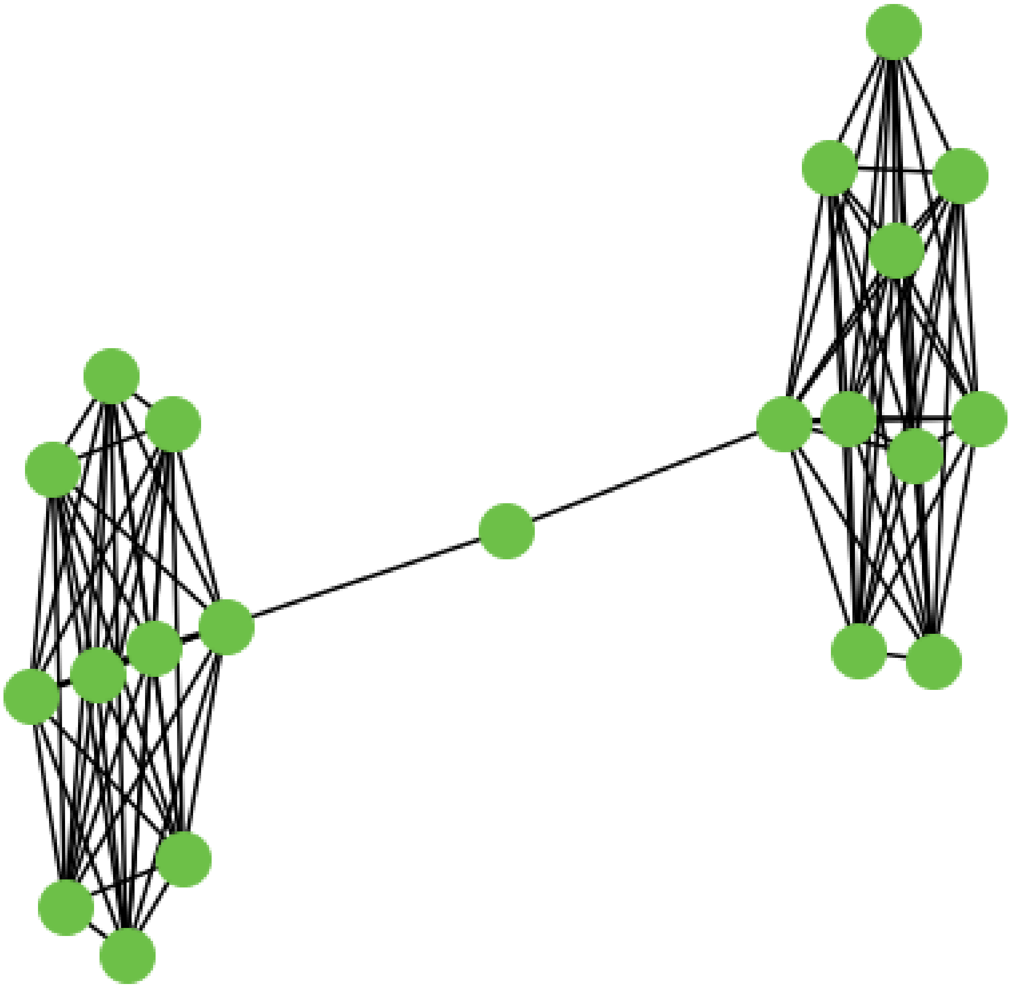
[13]:
```
G = nx.barbell_graph(10,1)

posX,posY = layout_graph(G)
draw_graph(G, posX,posY)
```



Now we can actually see why this is called a “barbell graph”.

Here’s the same graph but drawn with a **stronger attractive force**:

[14]:
```
posX,posY = layout_graph(G, k_a=0.5)
draw_graph(G, posX,posY)
```



Finally, let's get back to the ER graphs we looked at in the beginning:

[15]:
```
G = nx.erdos_renyi_graph(30,0.1) # another realization
print(nx.info(G))
```

```
Name:
Type: Graph
Number of nodes: 30
Number of edges: 50
Average degree:   3.3333
```

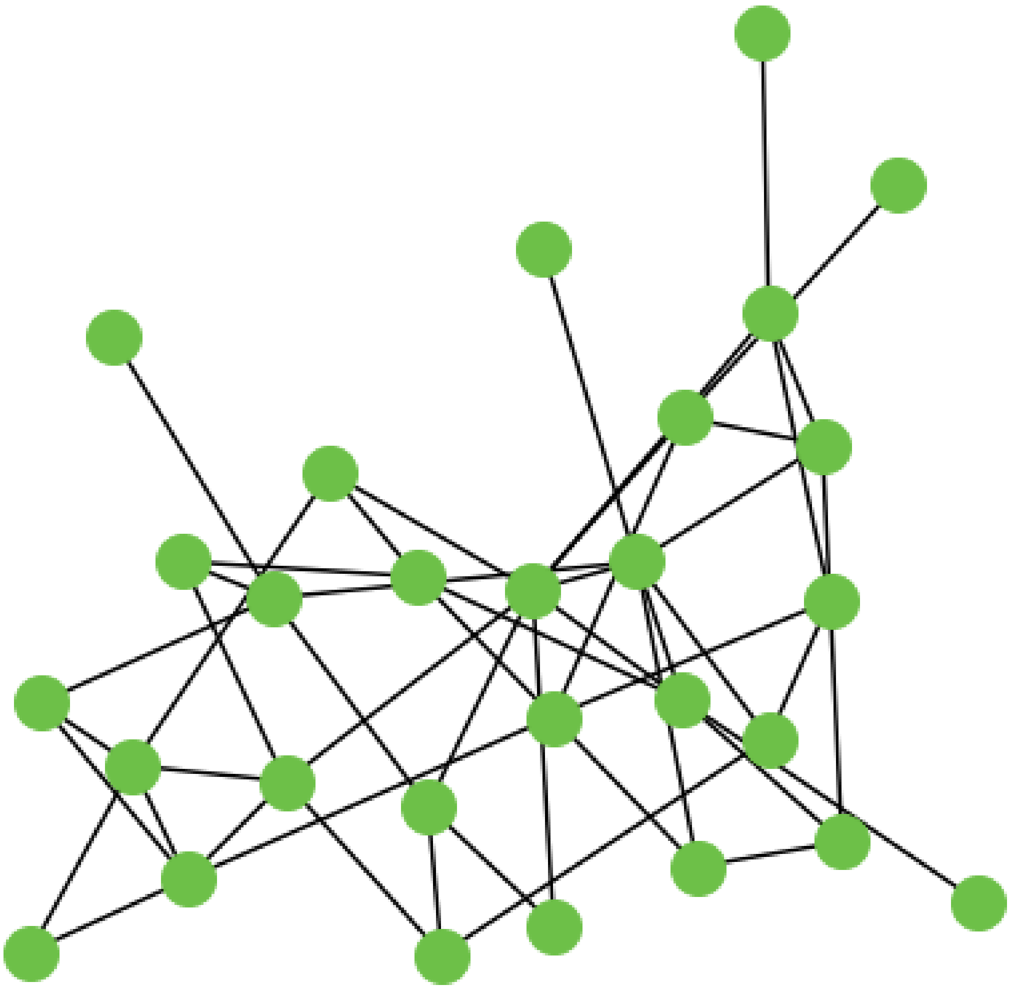
Now our code is pretty basic. It's not clear what will happen if the graph is disconnected. Depending on the value of p relative to N this can happen. So I'm going to cheat a little bit and cut out all the smaller components (if any):

[16]:
```
S = [G.subgraph(c).copy() for c in nx.connected_components(G)]
G = max(S, key=len)
print(nx.info(G))
```

```
Name:
Type: Graph
Number of nodes: 28
Number of edges: 50
Average degree:   3.5714
```

[17]:

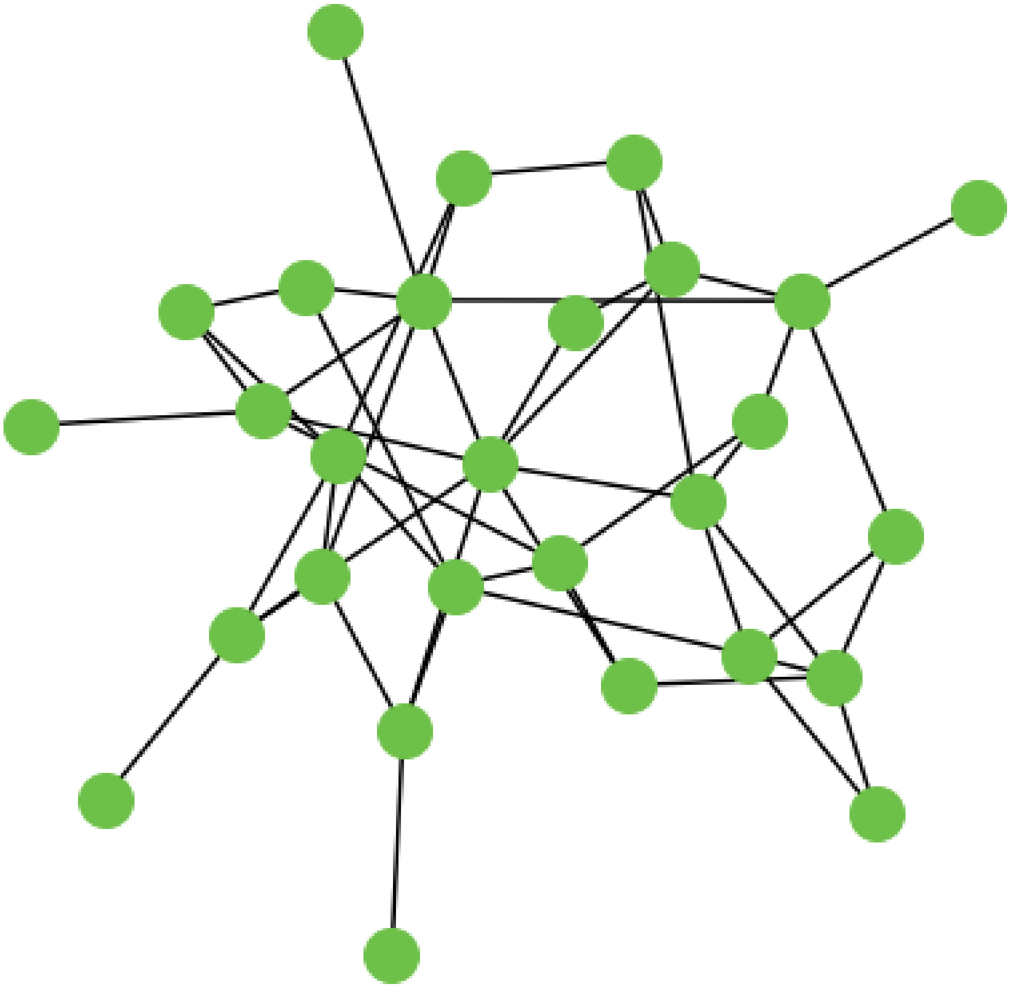
```
posX,posY = layout_graph(G)
draw_graph(G, posX,posY)
```



Our graph layout begins with random initial positions of nodes, so if we run the exact same code again, we may get a very different layout:

[18]:

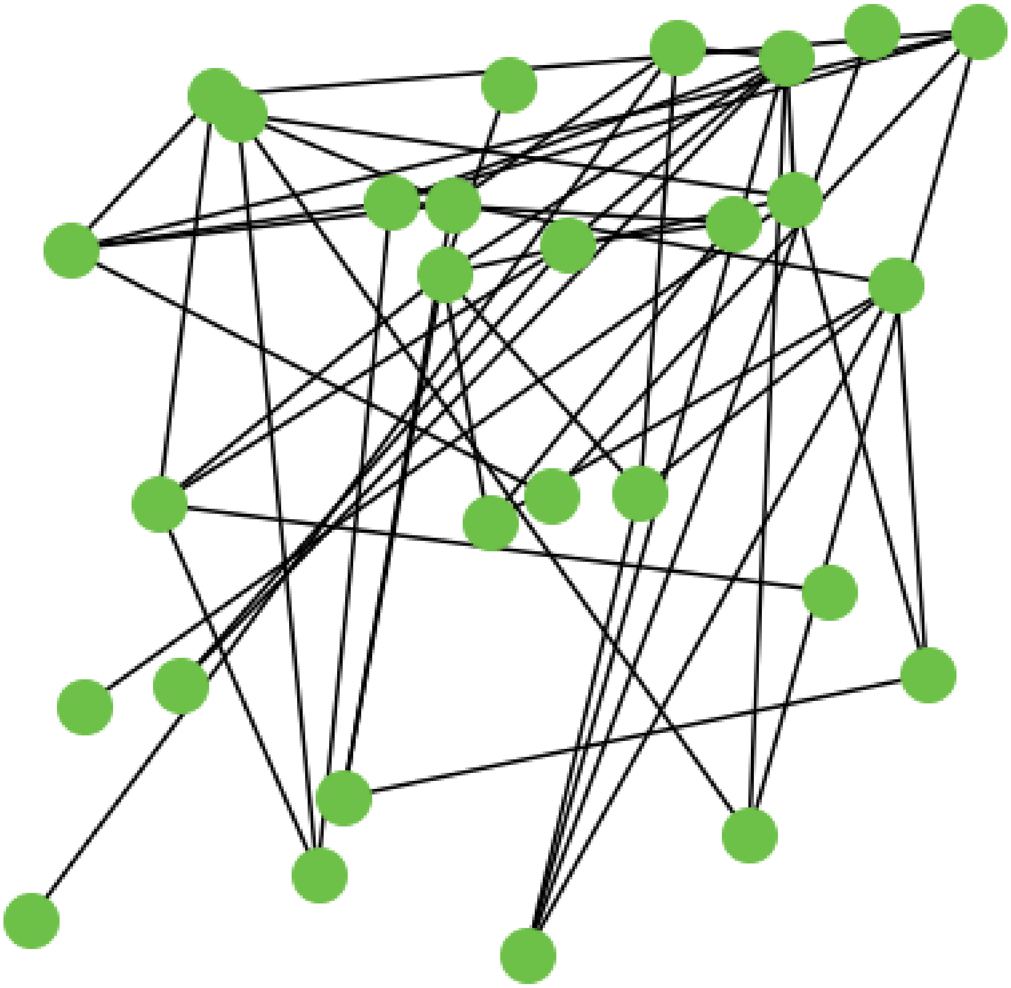
```
posX,posY = layout_graph(G)
draw_graph(G, posX,posY)
```



This may not be earth shattering, but it's certainly better than random:

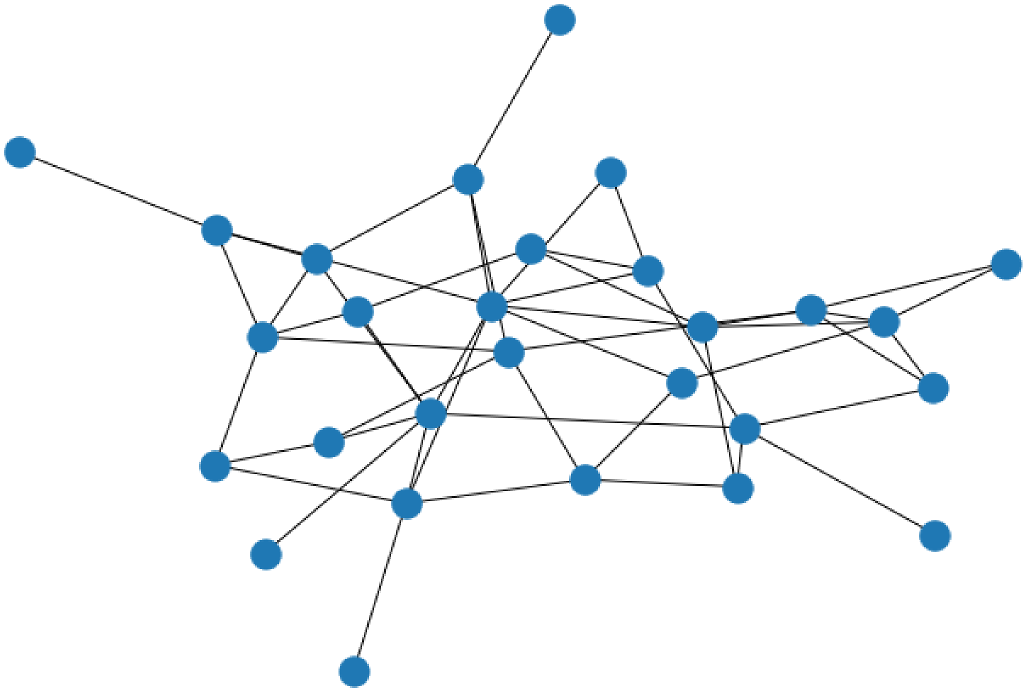
[19]:

```
# build random initial positions
n2x = {} # x-coords
n2y = {} # y-coords
for n in G:
    n2x[n] = random.random()
    n2y[n] = random.random()


draw_graph( G, n2x,n2y )
```



How does this compare to NX's built in layout mechanism?

[20]:
```
nx.draw_spring(G)
```



NX looks a little less cramped than ours. We may be able to tune our force constants to spread the nodes out a little bit, but much of that is also due to our drawing function using larger circles and thicker lines.

Of course, this can be animated or done dynamically, especially if it was implemented efficiently.

- Other interesting force-based graph visualizations

Practical drawing

Unfortunately, the above code is not very stable. Choosing bad values for the constants can lead to graphs that just **bounce around** forever, never settling down.

Likewise, we may need to carefully tune the constants. Good values for a lattice may look very bad on an ER graph.

Finally, the code (as implemented above) is relatively inefficient because we computed the repulsive force on a node due to every other node. This will become **too costly** when there are many nodes.

- Many practical force-directed implementation actually approximate the repulsive forces by only considering nearby nodes, and use fancy data structures to find the nearby nodes efficiently.

So in practice you will likely be working with pre-existing drawing tools.

- Normally I like to automate workflows, avoiding manual steps so that I can avoid mistakes, work faster, and keep provenance. But graph drawing is a very visual process, and many algorithms (all?) don't work perfectly. So it's useful to have a graphical (GUI) application where I can reposition some of the nodes following a layout, to tweak it.

- There are a number of these GUI applications. One nice choice is Gephi. It's cross platform and free/open.

To interface our code with an external program like gephi, we should have a file format to export the graph.

- Fortunately, NX has our backs once again!

[21]:

```
G = nx.barabasi_albert_graph(200,2) # bigger!

nx.write_graphml(G, "BA_graph.graphml")
```

Getting Fancy

Remember those dreaded bezier curves? Well, they're useful here as well:

(Actually can use force-directed method to layout the curve as well!)

- Flavor network and the principles of food pairing, *Nature Scientific Reports*, 2011

Tips for good visualizations

- Run a good layout algorithm, but be prepared to tweak by hand, sometimes extensively
- Use variable-size nodes. Map a network property to node size. Even something potentially redundant like node degree can still help emphasize the network's hubs. Here's an example mapping an error value for each node to the size of that node:
- Colors for nodes can help emphasize categories. (Recall all our work with colormaps.)
- Thicker or thinner lines for edges can be used on weighted networks.
- Try annotating the network:
- Don't make the lines too thin.

Generally, the best network visualizations augment the circles and lines with color, variable sizes, words or annotations, etc.

Interesting example application

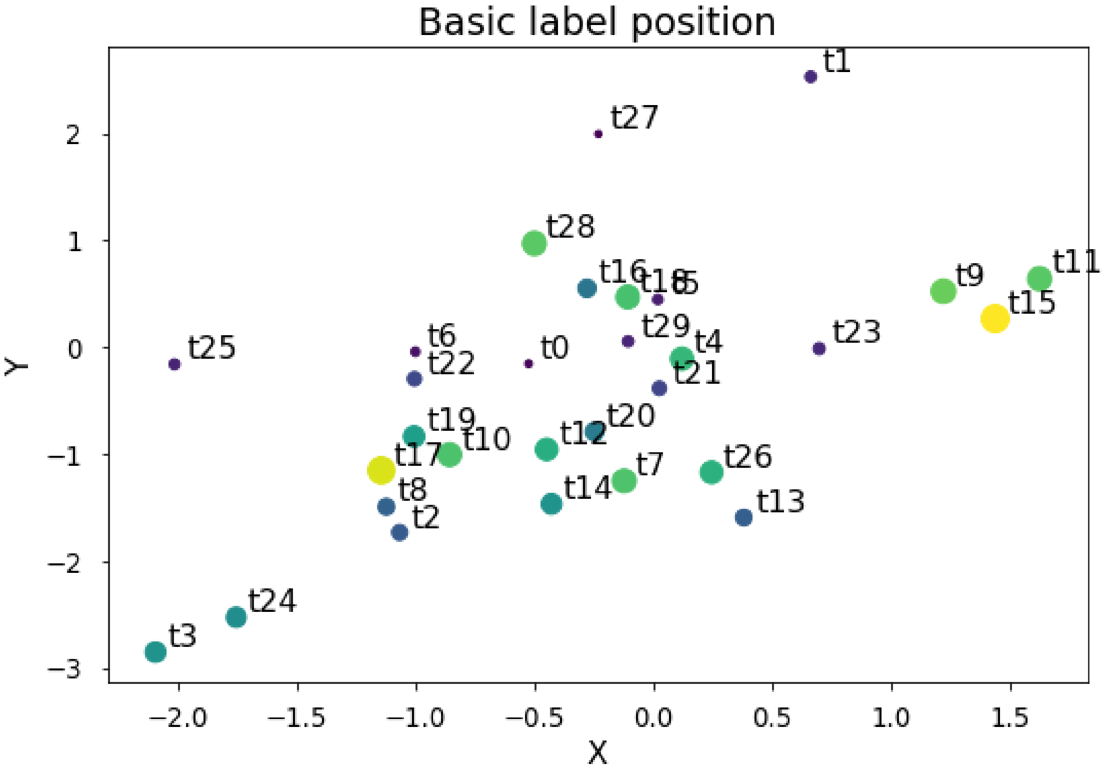
Here's something fun to do using network layouts as part of a non-network visualization: **label placement**

Some X,Y data to plot:

[22]:

```
"""
Rough improvement of scatter point label placement using
force-directed graph layout.

See also Stack Overflow: http://bit.ly/1NXgB3o
"""

# some fake data:
N = 30
X = np.random.randn(N)
Y = X*0.5 + np.random.randn(N)
C = np.random.rand(N)

plt.scatter(X,Y,c=C, s=C*200)
```

```
for i in range(N):
    d_str = "d%i" % i
    t_str = "t%i" % i
    plt.annotate(t_str,
                 xy=(X[i],Y[i]),
                 xytext=(X[i]+0.05,Y[i]+0.05),
                 )
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Basic label position")
plt.show()
```



Build an invisible "network" on top of the scatter plot, where each link is a scatter point at one end and a text label at the other

- Crucially, *freeze* the scatter point end so that it cannot move, only the text endpoint can move

Use graph layout algorithms to get a first pass at a better placement of the text labels

[23]:

```
# these constants likely require tuning for different data:
dx,dy = 0.05,0.05 # initial offset of labels from points
dX,dY = 0.8, 0.8  # expand the frame of the plot to make room for stretched labels
WEIGHT = 10.0 # strength of links, how far from points can text be?
K = 0.3 # preferred distance between points
label_repulsive_weight = 0.0005 # labels push away from each other
```

[24]:

```
# build the network:
from itertools import combinations

d_nodes, t_nodes = [], []
G = nx.Graph()
node2coord = {}
for i in range(N):
    x, y = X[i], Y[i]
    d_str = "d%i" % i
    t_str = "t%i" % i

    d_nodes.append(d_str)
    t_nodes.append(t_str)

    G.add_edge(d_str, t_str, weight=WEIGHT)

    node2coord[d_str] = (x, y)
    node2coord[t_str] = (x+dx, y+dy)

# "t" nodes are self-repulsive:
for ni,nj in combinations(t_nodes,2):
    G.add_edge(ni,nj, weight=-label_repulsive_weight*WEIGHT)
```

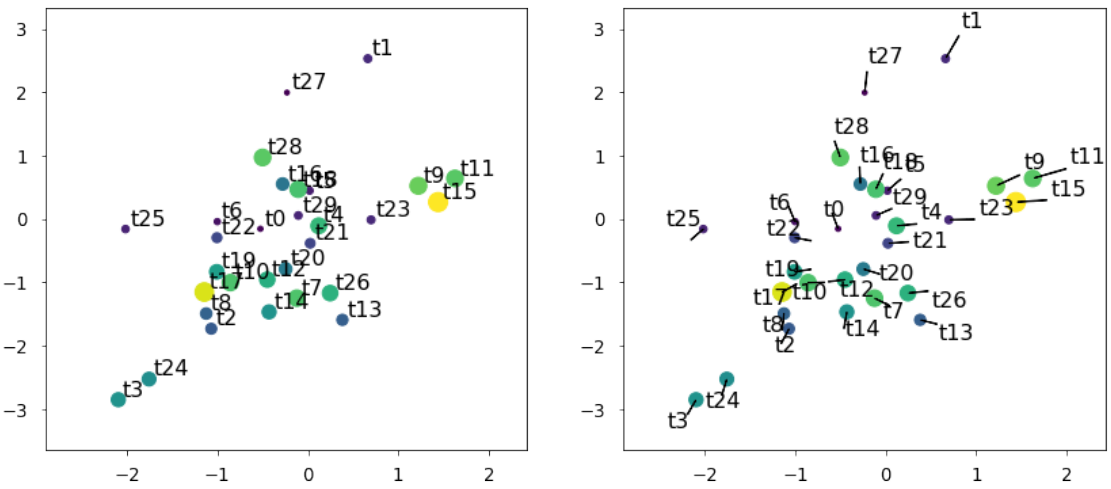
[25]:

```
# compute new layout, only "t" nodes can move:
node2coord_springs = nx.spring_layout(G, k=K, pos=node2coord, fixed=d_nodes)
```

And plot:

[26]:

```
def draw_and_label(coords, arrows=False):
    ax = plt.gca()
    ax.scatter(X, Y, c=C, s=C*200)

    arrowprops = None
    if arrows:
        arrowprops = dict(shrink=0.05, width=0.5, headwidth=0.5)
    for i in range(N):
        d_str = "d%i" % i
        t_str = "t%i" % i
        ax.annotate(t_str,
                    xy=coords[d_str],
                    xytext=coords[t_str],
                    arrowprops=arrowprops,
                    )
    ax.set_xlim(X.min()-dX, X.max()+dX)
    ax.set_ylim(Y.min()-dY, Y.max()+dY)


plt.figure(figsize=(14,6))
plt.subplot(121)
draw_and_label(node2coord)
plt.subplot(122)
draw_and_label(node2coord_springs, arrows=True)

plt.show()
```



Not perfect, but as a starting point for a first pass, not bad!

- Note the emphasis in these plots is on the labels not the points, but in practice the opposite would be true. As these stand now, the labels *overwhelm* the data, something you should avoid. Likely better to use a lighter color for the labels and/or not label every point
- Omitting the connector lines between the points and the labels could also clean things up

Summary

- Network layout or graph drawing is a non-trivial problem.
  - Simple methods called Force-Directed use a physics-based approach.
- You will likely use a pre-existing toolbox or GUI application such as Gephi.
- To make a good visualization takes care and attention, experimentation is important.