This tutorial is Part 2 of an introduction to social network analysis in Python. It covers how to sample and visualize network data. The primary example used for replication is Adamic and Glance's (2005) paper on relationships between political blogs ahead of the 2004 election. The paper is available here and is the source of all figures included below. We use the GML file available [here (http://www.thomaspadilla.org/data/network/polblogs/polblogs.gml)](http://www.thomaspadilla.org/data/network/polblogs/polblogs.gml) or [here (http://www-personal.umich.edu/~mejn/netdata/)](http://www-personal.umich.edu/~mejn/netdata/) ([documentation (http://www.thomaspadilla.org/data/network/polblogs/polblogs.txt)](http://www.thomaspadilla.org/data/network/polblogs/polblogs.txt)).

Table of Contents

Let's get started! First, we need to import all of the packages we'll use to do our analysis.

In [1]:
```python
import pandas as pd               # For analyzing tabular data
import numpy as np                # For working with arrays and numerical operations
import networkx as nx             # For network data specifically
from networkx.drawing.nx_agraph import graphviz_layout  # ...and graphing network data
import matplotlib.pyplot as plt   # For making plots
from pyvis import network as pv   # For making interactive plots
import random                     # For generating random numbers

%matplotlib inline
```

1 Importing external data files

We will be using a dataset from Adamic and Glance's (2005) paper: *The political blogosphere and the 2004 US election: Divided they blog.*

The paper built a dataset of political blogs as follows:

- They compiled a list of important blogs from blog curation sites (e.g. BlogCatalog).
- They crawled the front pages of these blogs and got all other linked blogs on these pages.
- If the other blog had at least 17 citations in the dataset, they kept it and got its links.
- They labeled the political orientation of the blogs using the blog curation sites and their own inspections.

The final dataset had:

- 759 liberal blogs
- 735 conservative blogs

Key findings from this analysis:

> "91% of links originating within either the conservative or liberal communities stay within that community...Conservative blogs show a greater tendency to link."

Reference: https://networkx.github.io/documentation/stable/reference/readwrite/index.html (https://networkx.github.io/documentation/stable/reference/readwrite/index.html)

First, let's practice importing files from external data sources. There are many different formats for disseminating network data, but some helpful ones supported by NetworkX include:

```
nx.read_gml(filepath)                      # GML

nx.read_edgelist(filepath, delimiter=',')  # A delimited edgelist

nx.from_pandas_edgelist(df)                # Pandas dataframe in edge list format
nx.from_pandas_adjacency(df)               # Pandas dataframe in adjacency matrix format
```

We'll get started with the dataset of blog links.

Note: Because some edges are duplicated, we had to modify the original file by adding a line to indicate that it is a multigraph. So, the dataset on this server is not perfectly identical to the original.

In [2]:
```
# Let's use NetworkX's read_gml to bring in the raw data file
G = nx.read_gml("data/polblogs.gml")
G
```

Out[2]: `<networkx.classes.multidigraph.MultiDiGraph at 0x154b76e5470>`

In [3]:
```
# As discussed, note that the graph above is a multigraph
# Let's now convert it from a multigraph to a directed graph
P = nx.DiGraph(G)
```

In [4]:
```
# And, we inspect
print(nx.info(P))
```

```
Name:
Type: DiGraph
Number of nodes: 1490
Number of edges: 19025
Average in degree:  12.7685
Average out degree:  12.7685
```

The dataset does not seem to be a perfect match for the paper. For example, we have 1940 links while they had 1494. But, it's close. We can also confirm that our degree counts roughly match those reported for Daily Kos (338), Eschelon (264), and Instapundit (277).

In [5]:
```
# Code to get all blog names containing a given keyword: [i for i in list(P.nodes) if 'instapundit' in i]

# Let's check if individual blog degrees match:
P.in_degree(['dailykos.com', 'atrios.blogspot.com', 'instapundit.com'])
```

Out[5]: `InDegreeView({'dailykos.com': 337, 'atrios.blogspot.com': 263, 'instapundit.com': 276})`

2 Downsampling graphs

We can see that the graph is (relatively) large and pretty well-connected: 1490 nodes and 19,025 edges. This is not actually "large" by network standards (Facebook has 1.7 billion users) but on my computer the plots are still running slow.

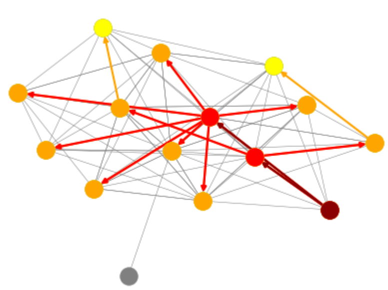
Since we will be making *many* plots, and since it's generally helpful for large graphs, let's learn to downsample. I will demonstrate two different approaches.

2.1 Snowball sample

One common method for sampling graphs is *snowball sampling*. The basic idea is as follows:

1. Start with set of seed nodes.
2. Add any friends of the seed nodes.
3. Add any friends of the friends of the seed nodes.
4. Add any friends of the friends of the (friends of the...) seed nodes.

In [6]:



We will use a few helper functions, including some from Python's `random` package:

```
nx.single_source_shortest_path(G, source, cutoff=n)  # Get all shortest paths from source node within distance n
nx.bfs_tree(G, source, depth_limit=n )               # Conduct a breadth-first search of depth n

S = G.subgraph(...)       # Select a subgraph

random.seed(n)            # Set a "seed" for the random number generator. This ensures that your code is replicable.
random.sample([...], n)   # Choose n items from a list of items
```

2.1.1 Snowball sampling with single-source-shortest-paths

Actually, last week we learned an algorithmic approach to implementing this with NetworkX. Remember that we had a function, `single_source_shortest_path` , that would give a list of all shortest paths from a source node. Well, that function takes an optional argument, `cutoff` , that limits the maximum length of the shortest paths returned.

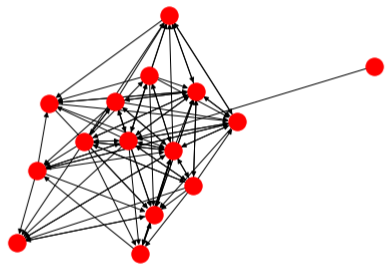
This can help us find our sample: we start from our source node, and then walk away in all directions until we hit the cutoff.

Let's try this for the blogger Matt Yglesias.

In [7]:
```
# Let's make him the path
source = "matthewyglesias.com"

# Let's find what other nodes can be reached in two steps or less
paths = nx.single_source_shortest_path(P,source,cutoff=2)

# Our sample will be all of these nodes
sample = list(paths.keys())
len(sample)
```

Out[7]: 15

In [8]:
```
# Let's take the subgraph of P containing this sample
S = P.subgraph(sample)
nx.draw(S)
```



2.1.2 Snowball sampling with breadth-first search

Note that this is equivalent to a breadth-first tree search on the graph.

In [9]:
```
# Let's do a BFS from the source with a depth of 2
sample = nx.bfs_tree(Q, source, depth_limit=2 )

# Let's collect the subgraph
S = P.subgraph(sample)

# Let's confirm that it's the same size
len(sample)
```

Out[9]:
```
15
```

2.2 Node and edge samples

Let's try another strategy, just for fun.

1. We will first randomly sample 500 nodes from the graph (reduce the number of nodes).
2. Then, we will randomly sample 500 edges from these nodes (reduce the number of edges).
3. Finally, we will keep only the largest connected component (reduce the number of small isolated components).

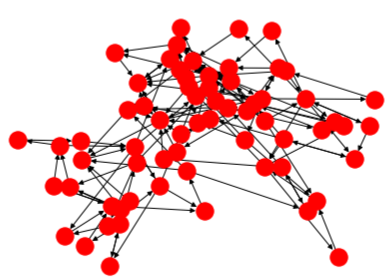
In [10]:
```
# Set the seed for a random number
random.seed(1)
```

In [11]:
```
# Step 1 -------------------------------------------------

# Choose a random sample of p's nodes
random_nodes = random.sample(P.nodes, 500)

# Get the subgraph containing of P these nodes
R = P.subgraph(random_nodes)

print(nx.info(R))
```

```
Name: 
Type: DiGraph
Number of nodes: 500
Number of edges: 1947
Average in degree:   3.8940
Average out degree:   3.8940
```

In [12]:
```
# Step 2 -------------------------------------------------

# Choose a random sample of p's edges
random_edges = random.sample(R.edges, 500)

# Get the subgraph containing of P these edges
R = P.edge_subgraph(random_edges)
```

In [13]:
```
# Step 3 -------------------------------------------------

# Get the biggest connected component
R = max(nx.strongly_connected_component_subgraphs(R), key=len)

nx.draw(R)
```



3 Plotting

Exercise: Now, it's time to see what we've got! Before we start, let's practice drawing a graph of our own. Try plotting the following adjacency matrix:

In [14]:

Out[14]:

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	1	0
2	0	1	0	0	0	0	0	0	1	0
3	1	0	0	0	0	0	0	0	0	0
4	1	0	0	1	0	0	0	0	0	0
5	0	1	0	0	1	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	1	0	1	0	0	0
8	0	0	0	0	1	0	1	0	0	0
9	0	1	0	0	0	0	0	0	0	0

3.1 Graph Layouts

Reference: [https://networkx.github.io/documentation/stable/reference/drawing.html#module-networkx.drawing.layout (https://networkx.github.io/documentation/stable/reference/drawing.html#module-networkx.drawing.layout)](https://networkx.github.io/documentation/stable/reference/drawing.html#module-networkx.drawing.layout)

A basic challenge for plotting networks is how to lay out the nodes and edges.

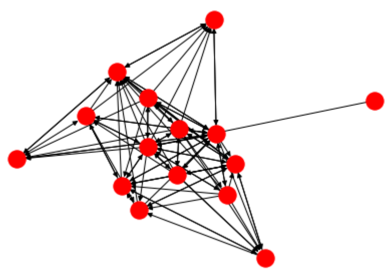
- One option is to use a **fixed layout** design, such as circular layouts (which put all nodes in a circle) or geographic layouts (which put nodes at their geographic location on a map).
- An alternative is to use an **algorithm** to determine the layout of nodes. Common classes of algorithms are:
  - **Force-directed layouts:** These algorithms generally balance two forces: a baseline repulsion between all nodes, and a countervailing attraction between connected nodes.
  - **Spectral layouts:** These algorithms perform dimensionality reduction to divide the graph into different clusters.

You might have to experiment a bit to get the layout you want. Let's try it!

3.1.1 Default (spring) layout

In [15]:

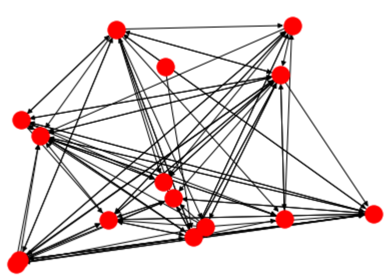
```
# We start with the default graph.
nx.draw(S)
```



3.1.2 Random layout

In [16]:

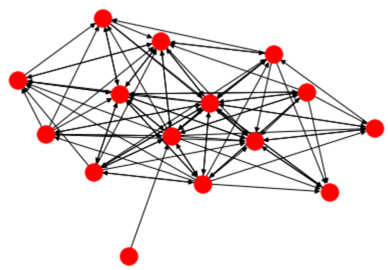
```
# Actually, it's not bad! If we drew it with a random layout, it would look worse...
nx.draw_random(S)
```



3.1.3 Kamada-Kawai (force-directed) layout

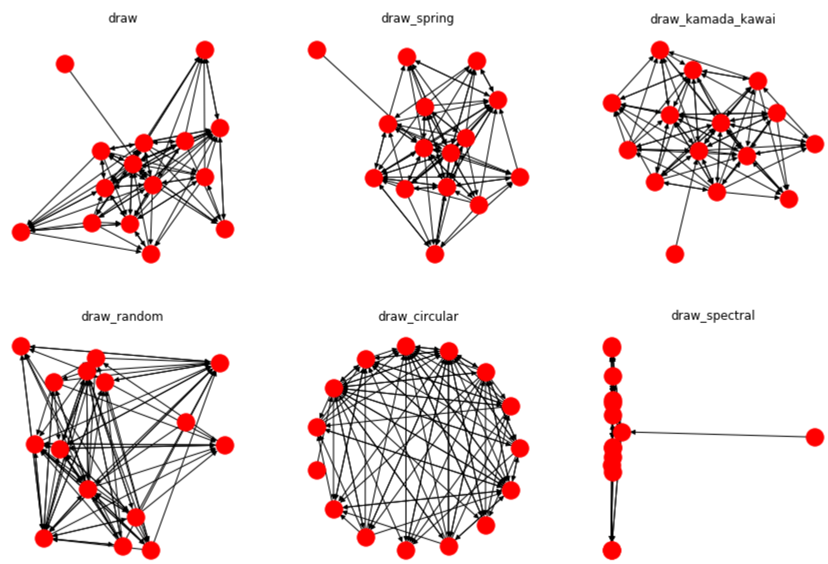
In [17]:

```
# Maybe there are some other alternatives?
nx.draw_kamada_kawai(S)
```



3.1.4 Layouts overview

In [18]:



3.1.5 Tuning graph layouts

We can also tweak individual algorithms. Before we start with that, though, we need to understand a little bit about NetworkX.

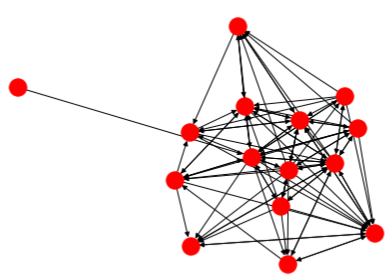
- Above, we have seen NetworkX's `draw` commands that directly incorporate the desired layout, e.g. `draw_spring` .
- However, we can also plot graphs with a given layout in two steps:

  1. Request the layout positions.
  2. Pass the layout positions to the `draw` command.

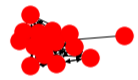
To tweak the positions, you'll need to read the documentation to see what parameters are available to you. For example, let's look into the [spring_layout (https://networkx.github.io/documentation/stable/reference/generated/networkx.drawing.layout.spring_layout.html)](https://networkx.github.io/documentation/stable/reference/generated/networkx.drawing.layout.spring_layout.html).

In [19]:

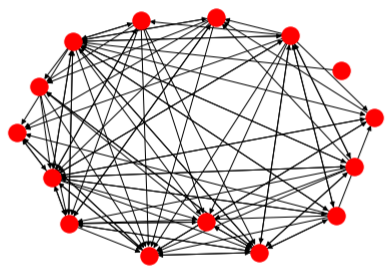
```
# To tweak the layout, we'll need to work in two steps
pos = nx.spring_layout(S)        # First, we run the algorithm to get the positions
nx.draw(S, pos=pos)              # Next, we plot the graph using those positions
```



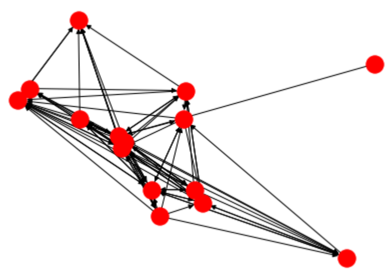
In [20]:
```python
# For example, we can scale it down in size ...
pos = nx.spring_layout(S, scale=0.01)
nx.draw(S, pos=pos)
```



In [21]:
```python
# ... we can push nodes apart ...
pos = nx.spring_layout(S, k=5)
nx.draw(S, pos=pos)
```



In [22]:
```python
# ...or pull them together
pos = nx.spring_layout(S, k=0.01)
nx.draw(S, pos=pos)
```



3.2 Styling the plot aesthetic

For now, let's stick with the default Kamada-Kawai layout. We will set this one time and then use it subsequently below.

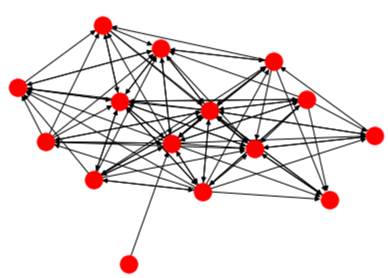
In [23]:
```python
# Set the positions for the rest of this section
pos = nx.kamada_kawai_layout(S)
```

3.2.1 Plotting the default

Below, we can see the default plot that we've grown used to generating. But now, it takes the `pos` argument which determines the node positions.

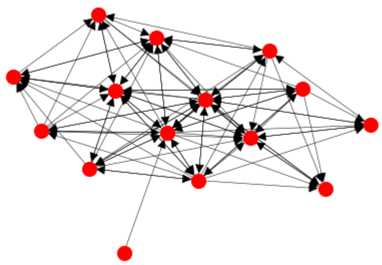
We can pass other arguments to the command to change other aspects of the graph styling:

In [24]:
```python
# Draw the default
nx.draw(S, pos=pos)
```

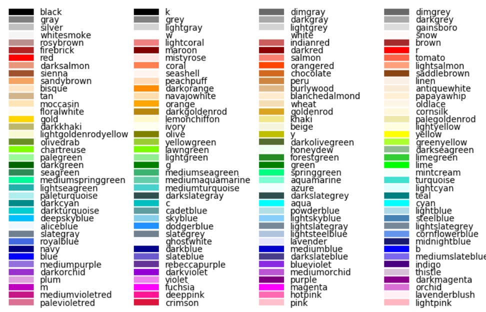


3.2.2 Change sizes of nodes, edges, or arrows

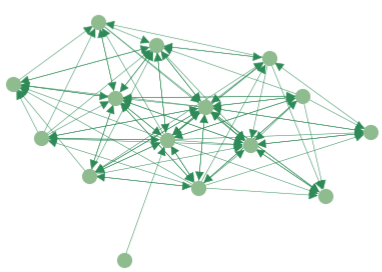
In [25]:
```
nx.draw(S, pos=pos,
        node_size=200,       # Change size of node
        width=0.5,           # Change width of edge
        arrowsize=20         # Change size of arrow
       )
```



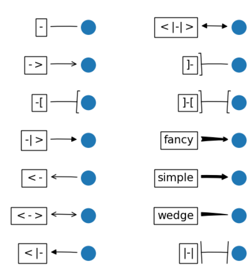
3.2.3 Change colors of nodes or edges

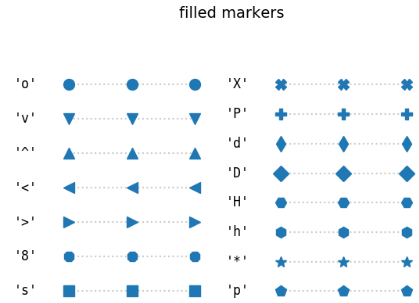


In [26]:
```
nx.draw(S,  pos=pos,
        node_size=200,
        width=0.5,
        arrowsize=20,
# ----------------------
        node_color="darkseagreen",
        edge_color='seagreen'
       )
```

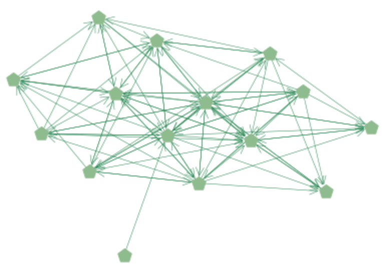


3.2.4 Change styling of nodes or arrows



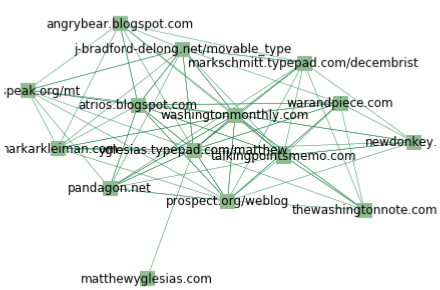


In [27]:
```
nx.draw(S, pos=pos,
        node_size=200,
        width=0.5,
        arrowsize=20,
        node_color="darkseagreen",
        edge_color='seagreen',
# ----------------------
        node_shape='p',
        arrowstyle='->'
       )
```



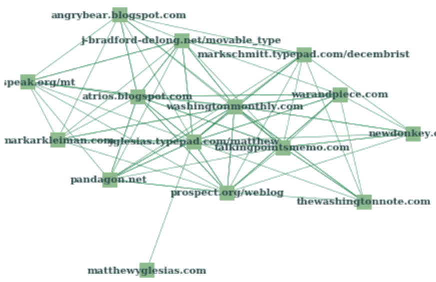
3.2.5 Turn plot features on and off

In [28]:
```
nx.draw(S, pos=pos,
        node_size=200,
        width=0.5,
        node_color="darkseagreen",
        edge_color='seagreen',
        node_shape='s',
# ----------------------
        arrows=False,          # Control whether arrow is shown
        with_labels=True       # Control whether plot is labeled
       )
```



3.2.6 Adding text

In [29]:
```python
nx.draw(S, pos=pos,
        node_size=200,
        width=0.5,
        node_color="darkseagreen",
        edge_color='seagreen',
        node_shape='s',
        arrows=False,
        with_labels=True,
# ----------------------
        font_size=10,
        font_color='darkslategrey',
        font_weight='bold',
        font_family='serif'
       )
```



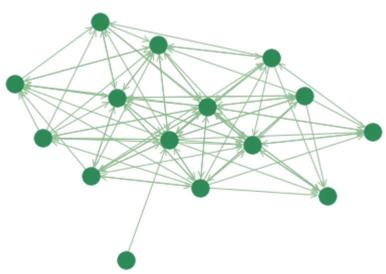
3.3 Drawing attention to selected features

Now, we'll show how you can select and emphasize certain features. For example, you might want to tell a story about a given node.

Essentially, the easiest way to do this is:

1. Get the positions of the graph layout.
2. Plot the underlying graph and nodes.
3. Layer your styling on top.

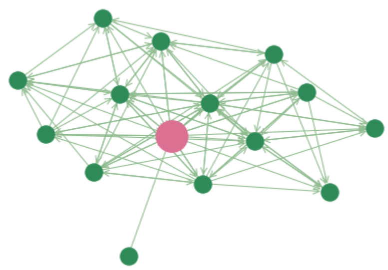
In [30]:
```python
# Let's start with a simple baseline graph
nx.draw(S, pos=pos,
        node_size=300,
        node_color="seagreen",
        edge_color='darkseagreen',
        arrowstyle='->',
        arrowsize=15
       )
```



3.3.1 Drawing selected nodes

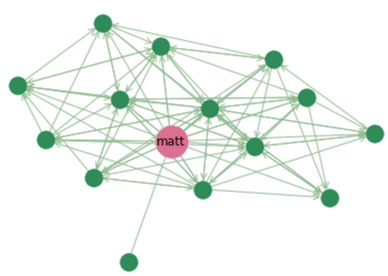
In [31]:
```python
# Let's plot the underlying graph
nx.draw(S, pos=pos,
        node_size=300,
        node_color="seagreen",
        edge_color='darkseagreen',
        arrowstyle='->',
        arrowsize=15
       )

# -----------------------------------------------------
# We can choose a list of nodes we want to emphasize (in this case, one of Matt Yglesias' pages)
selected_nodes = ['yglesias.typepad.com/matthew']

# Let's resize and recolor the node(s) of interest
nx.draw_networkx_nodes(S, pos=pos,
                      nodelist=selected_nodes,         # List of nodes to alter with new styling
                      node_color = 'palevioletred',
                      node_size=1000)
```

Out[31]: <matplotlib.collections.PathCollection at 0x154bbf7f470>



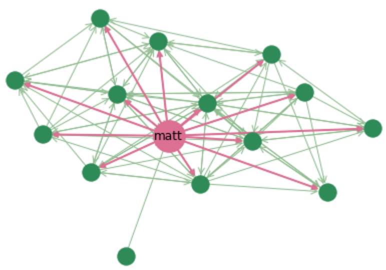
3.3.2 Drawing selected node labels

In [32]:
```python
# Let's plot the underlying graph
nx.draw(S, pos=pos,
        node_size=300,
        node_color="seagreen",
        edge_color='darkseagreen',
        arrowstyle='->',
        arrowsize=15
       )

# We can choose a list of nodes we want to emphasize (in this case, one of Matt Yglesias' pages)
selected_nodes = ['yglesias.typepad.com/matthew']

# Let's resize and recolor the node(s) of interest
nx.draw_networkx_nodes(S, pos=pos,
                      nodelist=selected_nodes,         # List of nodes to alter with new styling
                      node_color = 'palevioletred',
                      node_size=1000)
# -----------------------------------------------------

# We can create a dictionary of nodes to label, and the labels we want
selected_labels = {'yglesias.typepad.com/matthew': "matt"}

# Let's add these labels
nx.draw_networkx_labels(S, pos=pos,
                        labels=selected_labels,       # Dictionary of nodes to label
                        font_size=12)

plt.show()
```



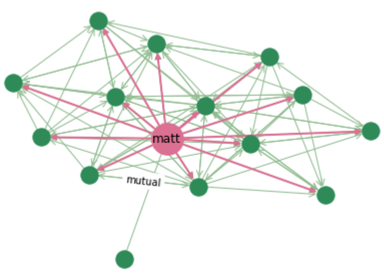
3.3.3 Drawing selected edges

In [33]:
```python
# Let's plot the underlying graph
nx.draw(S, pos=pos,
        node_size=300,
        node_color="seagreen",
        edge_color='darkseagreen',
        arrowstyle='->',
        arrowsize=15
       )

# We can choose a list of nodes we want to emphasize (in this case, one of Matt Yglesias' pages)
selected_nodes = ['yglesias.typepad.com/matthew']

# Let's resize and recolor the node(s) of interest
nx.draw_networkx_nodes(S, pos=pos,
                      nodelist=selected_nodes,         # List of nodes to alter with new styling
                      node_color = 'palevioletred',
                      node_size=1000)

# We can create a dictionary of nodes to label, and the labels we want
selected_labels = {'yglesias.typepad.com/matthew': "matt"}

# Let's add these labels
nx.draw_networkx_labels(S, pos=pos,
                        labels=selected_labels,       # Dictionary of nodes to label
                        font_size=12)


# -----------------------------------------------------
# We can create a list of edges we want to plot <- in this case, all of Matt's
selected_edges = list(S.edges(['yglesias.typepad.com/matthew']))

# We can plot only these edges
nx.draw_networkx_edges(S, pos=pos,
                      edgelist=selected_edges,        # List of edges to restyle
                      edge_color = 'palevioletred',
                      width=2)

plt.show()
```



3.3.4 Drawing selected edge labels

In [34]:
```python
# Let's plot the underlying graph
nx.draw(S, pos=pos,
        node_size=300,
        node_color="seagreen",
        edge_color='darkseagreen',
        arrowstyle='->',
        arrowsize=15
       )

# We can choose a list of nodes we want to emphasize (in this case, one of Matt Yglesias' pages)
selected_nodes = ['yglesias.typepad.com/matthew']

# Let's resize and recolor the node(s) of interest
nx.draw_networkx_nodes(S, pos=pos,
                      nodelist=selected_nodes,          # List of nodes to alter with new styling
                      node_color = 'palevioletred',
                      node_size=1000)

# We can create a dictionary of nodes to label, and the labels we want
selected_labels = {'yglesias.typepad.com/matthew': "matt"}

# Let's add these labels
nx.draw_networkx_labels(S, pos=pos,
                        labels=selected_labels,       # Dictionary of nodes to label
                        font_size=12)


# We can create a list of edges we want to plot <- in this case, all of Matt's
selected_edges = list(S.edges(['yglesias.typepad.com/matthew']))

# We can plot only these edges
nx.draw_networkx_edges(S, pos=pos,
                      edgelist=selected_edges,         # List of edges to restyle
                      edge_color = 'palevioletred',
                      width=2)

# ------------------------------------------------------
# We can create a dictionary of edges we want to label, and their labels
selected_edge_labels = {("pandagon.net", "prospect.org/weblog"): "mutual"}

nx.draw_networkx_edge_labels(S, pos=pos,
                            edge_labels = selected_edge_labels) # Dictionary of edges to label

plt.show()
```

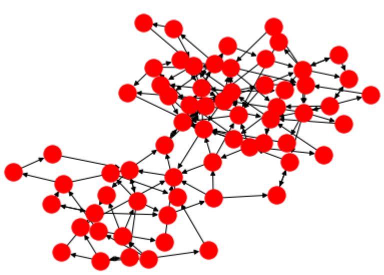


3.4 Illustrating discrete attributes

Of course, we might want to systematically render attributes of our graph. For example, in our dataset, there is an attribute called "value", which is set to 0 if the blog is liberal and 1 if it is conservative.

Let's drop Matt's blog network and switch back to the sample we previously made, R.

In [35]:
```python
# We'll plot R with the default Kamada Kawai layout
pos = nx.kamada_kawai_layout(R)
nx.draw(R, pos=pos)
```



Exercise: Let's tidy up the graph a bit. How would you...

- make the size of the nodes to be 100
- make the color of the edges to be grey
- make the style of arrows to be a "v" instead of a triangle?

Answer: pos = nx.kamada_kawai_layout(R) nx.draw(R, pos=pos, node_size=100, edge_color='gray', arrowstyle="->")

3.4.1 Coloring nodes by label

Let's highlight groups of nodes by changing the label color.

In [36]:
```python
# We can see that each node has an attribute caleld "value" with political affiliation
# It also has information about the source of the blog
R.nodes.data()
```

Out[36]:
```
NodeDataView({'jquinton.com': {'value': 1, 'source': 'LabeledManually'}, 'cbcbcbcb.blogspot.com': {'value': 1, 'source': 'Blogarama'}, 'timblair.net': {'value': 1, 'source':
'LabeledManually'}, 'mhking.mu.nu': {'value': 1, 'source': 'BlogPulse,eTalkingHead'}, 'demagogue.blogspot.com': {'value': 0, 'source': 'LeftyDirectory,eTalkingHead'}, 'crank
yneocon.com/crankyneocon': {'value': 1, 'source': 'LabeledManually'}, 'homespunbloggers.blogspot.com': {'value': 1, 'source': 'LabeledManually'}, 'mudvillegazette.com': {'va
lue': 1, 'source': 'LabeledManually'}, 'pajamaeditors.blogspot.com': {'value': 1, 'source': 'LabeledManually'}, 'inthebullpen.com': {'value': 1, 'source': 'LabeledManuall
y'}, 'talkleft.com': {'value': 0, 'source': 'BlogPulse,LeftyDirectory,eTalkingHead,CampaignLine'}, 'decision08.blogspot.com': {'value': 1, 'source': 'Blogarama'}, 'chrenkof
f.blogspot.com': {'value': 1, 'source': 'LabeledManually'}, 'vodkapundit.com': {'value': 1, 'source': 'eTalkingHead'}, 'washingtonmonthly.com': {'value': 0, 'source': 'BlogP
ulse,LeftyDirectory'}, 'thespoonsexperience.com': {'value': 1, 'source': 'eTalkingHead'}, 'incite1.blogspot.com': {'value': 1, 'source': 'Blogarama,eTalkingHead'}, 'evangeli
caloutpost.com': {'value': 1, 'source': 'eTalkingHead'}, 'littlegreenfootballs.com/weblog': {'value': 1, 'source': 'LabeledManually'}, 'alphapatriot.com': {'value': 1, 'sour
ce': 'Blogarama'}, 'discerningtexan.blogspot.com': {'value': 1, 'source': 'eTalkingHead'}, 'politicalstrategy.org': {'value': 0, 'source': 'CampaignLine'}, 'roxanne.typepad.
com': {'value': 0, 'source': 'eTalkingHead'}, 'emergingdemocraticmajorityweblog.com/donkeyrising': {'value': 0, 'source': 'LabeledManually'}, 'michellemalkin.com': {'value':
1, 'source': 'BlogPulse'}, 'shakespearessister.blogspot.com': {'value': 0, 'source': 'Blogarama,BlogCatalog'}, 'sayanythingblog.com': {'value': 1, 'source': 'BlogCatalog,eTa
lkingHead'}, 'gevkaffeegal.typepad.com/the_alliance': {'value': 1, 'source': 'LabeledManually'}, 'patriotboy.blogspot.com': {'value': 0, 'source': 'LeftyDirectory'}, 'all-en
compassingly.com': {'value': 1, 'source': 'Blogarama,eTalkingHead'}, 'laughingwolf.net': {'value': 1, 'source': 'eTalkingHead'}, 'professorbainbridge.com': {'value': 1, 'sou
rce': 'BlogPulse,eTalkingHead'}, 'deanesmay.com': {'value': 1, 'source': 'BlogPulse'}, 'aboutpolitics.blogspot.com': {'value': 0, 'source': 'Blogarama,eTalkingHead'}, 'kimdu
toit.com/dr/weblog.php': {'value': 1, 'source': 'LabeledManually'}, 'txfx.net': {'value': 1, 'source': 'BlogCatalog'}, 'reachm.com/amstreet': {'value': 0, 'source': 'Blogara
ma'}, 'elemming2.blogspot.com': {'value': 0, 'source': 'Blogarama,LeftyDirectory'}, 'papadoc.net/pinkflamingobar.html': {'value': 1, 'source': 'eTalkingHead'}, 'snunes.blogs
pot.com': {'value': 0, 'source': 'Blogarama,LeftyDirectory'}, 'answerguy.blogspot.com': {'value': 0, 'source': 'Blogarama'}, 'rhetoricrhythm.blogspot.com': {'value': 0, 'sou
rce': 'LeftyDirectory,eTalkingHead'}, 'liberaloasis.com': {'value': 0, 'source': 'LeftyDirectory,LabeledManually,CampaignLine'}, 'thisliberal.com': {'value': 1, 'source': 'B
logarama'}, 'burntorangereport.com': {'value': 0, 'source': 'BlogPulse,CampaignLine'}, 'ruminatethis.com': {'value': 0, 'source': 'LeftyDirectory'}, 'celluloid-wisdom.com/p
```

In [37]:
```python
# Let's extract political affiliation as a dictionary
political_affil = nx.get_node_attributes(R, 'value')
political_affil
```

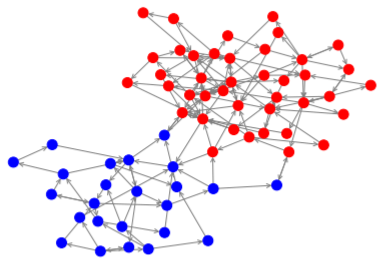
Out[37]:
```
{'aboutpolitics.blogspot.com': 0,
 'all-encompassingly.com': 1,
 'alphapatriot.com': 1,
 'answerguy.blogspot.com': 0,
 'beldar.blogs.com/beldarblog': 1,
 'bensworld.patriotforum.org': 1,
 'burntorangereport.com': 0,
 'cbcbcbcb.blogspot.com': 1,
 'celluloid-wisdom.com/pw': 1,
 'chrenkoff.blogspot.com': 1,
 'commonsenserunswild.typepad.com': 1,
 'crankyneocon.com/crankyneocon': 1,
 'crookedtimber.org': 0,
 'daisycutter.blogspot.com': 1,
 'deanesmay.com': 1,
 'decision08.blogspot.com': 1,
 'demagogue.blogspot.com': 0,
 'discerningtexan.blogspot.com': 1,
 'dneiwert.blogspot.com': 0,
 'ejectejecteject.com': 1,
 'elemming2.blogspot.com': 0,
 'emergingdemocraticmajorityweblog.com/donkeyrising': 0,
 'evangelicaloutpost.com': 1,
 'gcruse.typepad.com': 1,
 'gevkaffeegal.typepad.com/the_alliance': 1,
 'hillcountryviews.blogspot.com': 1,
 'homespunbloggers.blogspot.com': 1,
 'iddybud.blogspot.com': 0,
 'incite1.blogspot.com': 1,
 'inthebullpen.com': 1,
 'john.hoke.org': 0,
 'jquinton.com': 1,
 'kimdutoit.com/dr/weblog.php': 1,
 'laughingwolf.net': 1,
 'liberaloasis.com': 0,
 'littlegreenfootballs.com/weblog': 1,
 'mahablog.com': 0,
 'mhking.mu.nu': 1,
 'michellemalkin.com': 1,
 'mudvillegazette.com': 1,
 'pajamaeditors.blogspot.com': 1,
 'papadoc.net/pinkflamingobar.html': 1,
 'pardonmyenglish.com': 1,
 'patriotboy.blogspot.com': 0,
 'patterico.com': 1,
 'politicalstrategy.org': 0,
 'professorbainbridge.com': 1,
 'reachm.com/amstreet': 0,
 'redstate.org': 1,
 'rhetoricrhythm.blogspot.com': 0,
 'roxanne.typepad.com': 0,
 'ruminatethis.com': 0,
 'sayanythingblog.com': 1,
 'secureliberty.org/': 1,
 'shakespearessister.blogspot.com': 0,
 'snunes.blogspot.com': 0,
 'talkleft.com': 0,
 'the-goddess.org': 0,
 'the-hamster.com': 0,
 'thespoonsexperience.com': 1,
 'thisliberal.com': 1,
 'timblair.net': 1,
 'txfx.net': 1,
 'vodkapundit.com': 1,
 'washingtonmonthly.com': 0}
```

In [38]:
```python
# Let's make a list of node_ids and an empty list of node colors
node_ids = list(R.nodes)
node_colors = []

# Now, for each node, get its affiliation
for n in node_ids:
    if political_affil[n] == 0:       # 0 = liberal, let's color it blue
        node_colors.append("blue")
    elif political_affil[n] == 1:     # 1 = conservative, let's color it red
        node_colors.append("red")

# Let's inspect
print(node_ids[:10])
print(node_colors[:10])
```

```
['jquinton.com', 'cbcbcbcb.blogspot.com', 'timblair.net', 'mhking.mu.nu', 'demagogue.blogspot.com', 'crankyneocon.com/crankyneocon', 'homespunbloggers.blogspot.com', 'mudvill
egazette.com', 'pajamaeditors.blogspot.com', 'inthebullpen.com']
['red', 'red', 'red', 'red', 'blue', 'red', 'red', 'red', 'red', 'red']
```

In [39]:
```python
# There is a one-line alternative:
# node_colors = ["blue" if political_affil[n]==0 else "red" for n in node_ids]
```

In [40]:
```python
nx.draw(R, pos=pos,
        node_size=100,
        edge_color='gray',
        arrowstyle="->", fontsize=7,
        # -------------------------
        nodelist = node_ids,       # Now we just supply the ids
        node_color = node_colors  # And the list of colors corresponding to them
       )
```



3.4.2 Coloring edges by node label

Now, let's try to replicate Adamic and Glance's classic figure on our graph subset. We want to use the same intuition: make a list of edges, and then color the edges.

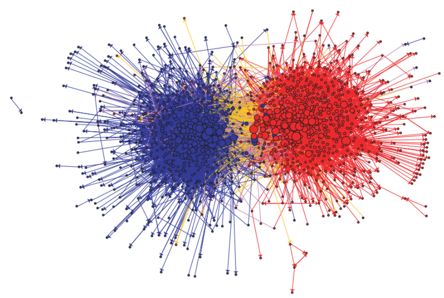
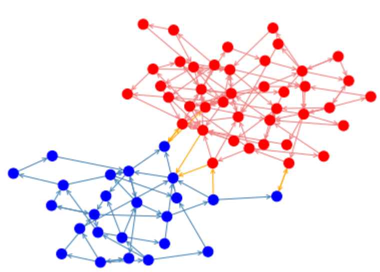


Figure 1: Community structure of political blogs (expanded set), shown using utilizing a GEM layout [11] in the GUESS[3] visualization and analysis tool. The colors reflect political orientation, red for conservative, and blue for liberal. Orange links go from liberal to conservative, and purple ones from conservative to liberal. The size of each blog reflects the number of other blogs that link to it.

In [41]:
```python
# Let's make a list of edge_ids and an empty list of edge colors
edge_ids = list(R.edges)
edge_colors = []

for e in edge_ids:

    n1,n2 = e

    # Liberal links to liberal
    if political_affil[n1] == 0 and political_affil[n2] == 0:
        edge_colors.append("steelblue")

    # Conservative links to conservative
    elif political_affil[n1] == 1 and political_affil[n2] == 1:
        edge_colors.append("lightcoral")

    # Links across parties
    elif political_affil[n1] != political_affil[n2]:
        edge_colors.append("orange")
```

In [42]:
```python
nx.draw(R, pos=pos,
        node_size=100,
        arrowstyle="->", fontsize=7,
        nodelist = node_ids,
        node_color = node_colors,
        # -------------------------
        edgelist = edge_ids,          # Now we just supply the edge ids
        edge_color = edge_colors      # and the list of colors corresponding to them
       )
```



Exercise: We almost replicated the graph, but not exactly. We use orange links to represent any cross-party ties, but in Adamic and Glance, the orange links only go from liberal to conservative. They use purple links to indicate the opposite direction. How would we modify the code to match their coloring scheme?

Answer: edge_ids = list(R.edges) # Let's make a list of edge_ids and an empty list of edge colors edge_colors = [] for e in edge_ids: n1,n2 = e if political_affil[n1] == 0 and political_affil[n2] == 0: edge_colors.append("steelblue") # Liberal links to liberal elif political_affil[n1] == 1 and political_affil[n2] == 1: edge_colors.append("lightcoral") # Conservative links to conservative elif political_affil[n1] == 0 and political_affil[n2] == 1: edge_colors.append("orange") # Liberal to conservative elif political_affil[n1] == 1 and political_affil[n2] == 0: edge_colors.append("purple") # Conservative to liberal nx.draw(R, pos=pos, node_size=100, arrowstyle="->", fontsize=7, nodelist = node_ids, node_color = node_colors, edgelist = edge_ids, # Now we just supply the edge ids edge_color = edge_colors # and the list of colors corresponding to them)

3.5 Illustrating continuous attributes

We can also use size to inform us about relevant quantities in the graph. For example, we might want to make important nodes larger, or plot the weights on edges.

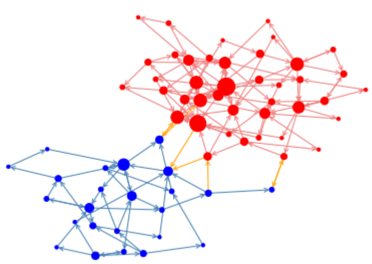
3.5.1 Sizing nodes by value

Note: When sizing nodes by value, it sometimes helps to scale the value in order to draw a more dramatic contrast. We can do this by multiplying the raw value or exponentiating. You can experiment with the scaling constants until you get the appearance you want.

3.5.1.1 Degree

In [43]:
```
# Let's make a list of node_ids and an empty list of node sizes
node_ids = list(R.nodes)
node_sizes = []

# Add the scaled degrees for each node
for n in node_ids:
    node_sizes.append((2*R.degree(n))**1.75)   # Experiment with scaling here
```

In [44]:
```
# Plot
nx.draw(R, pos=pos,
        node_size=node_sizes,                  # Supply a list of node sizes
        arrowstyle="->",
        fontsize=7,
        node_color = node_colors,
        edgelist = edge_ids,
        edge_color = edge_colors,
        nodelist = node_ids
       )
```

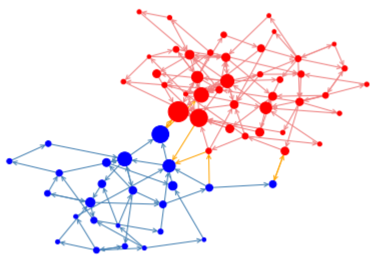


3.5.1.2 Pagerank

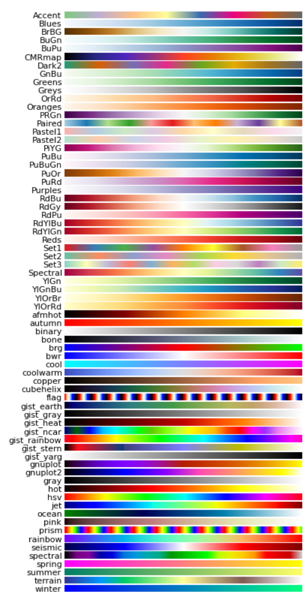
We can use the same general formula for any calculated value. For example, we can size nodes by PageRank:

In [45]:
```
# Get a dictionary of page ranks
pr = nx.pagerank(R)

# Let's make a list of node_ids and an empty list of node sizes
node_ids = list(R.nodes)
node_sizes = []

# Calculate the node sizes
for n in node_ids:
    node_sizes.append(4000*pr[n])              # Pagerank values are small (they sum to 1), so we scale aggressively
```

In [46]:
```
# Plot
nx.draw(R, pos=pos,
        node_size=node_sizes,                  # Supply a list of node sizes
        arrowstyle="->",
        fontsize=7,
        node_color = node_colors,
        edgelist = edge_ids,
        edge_color = edge_colors,
        nodelist = node_ids
       )
```



3.6 Coloring nodes by value



We might want to show multiple features. In that case, we can also use color as an informative dimension. Let's keep the node ids and node sizes from above, but add a color dimension that captures in-degree:

In [47]:
```
# Initialize a list to store the variable associated with the color
node_color_values = []

# Add the (numeric) values to the list
for n in node_ids:
    node_color_values.append(R.in_degree(n))
```

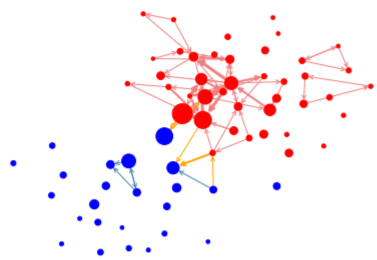
In [48]:
```
nx.draw(R, pos=pos,
        node_size=node_sizes,
        arrowstyle="->",
        fontsize=7,
        node_color = node_color_values,   # Supply the (numeric) data for the color variable
        cmap = plt.get_cmap('viridis'),   # Map the numeric data to a color scheme
        edgelist = edge_ids,
        edge_color = 'lightgrey',
        nodelist = node_ids,
        arrows=False
       )
```

Note: We can see that in-degree does not perfectly correspond with PageRank. The nodes with the highest in degree (the lightest) are not always the ones with the highest PageRank (the biggest). PageRank depends on the weight of inbound links (and not just the number of them).

3.7 Sizing edges by value

Finally, we might want to use weight values to change the edges sizing. Here, let's weight a tie by how many neighbors its endpoints have in common. By now, you can probably guess how to do this:

In [49]:
```
# Initialize a list of edge weights
edge_weights = []

# For each edge, append a weight
for n1,n2 in edge_ids:
    neighbors = nx.common_neighbors(nx.Graph(R),n1,n2)   # Get common neighbors of the end points
    n_neighbors = len(list(neighbors))                   # Count how many of these neighbors they have in common
    edge_weights.append(n_neighbors)                     # Add to the list (+1, in case no common neighbors)

# Plot
nx.draw(R, pos=pos,
        node_size=node_sizes,
        arrowstyle="->",
        fontsize=7,
        node_color = node_colors,
        edgelist = edge_ids,
        edge_color = edge_colors,
        nodelist = node_ids,
        width=edge_weights,                              # Supply the edge weights
        edge_labels = [str(i) for i in edge_weights]
       )
```



Exercise:

- Why are some edgs missing?
- How would you fix the code so that even edges with no common neighbors are drawn?

Answer: edge_weights = [] for n1,n2 in edge_ids: neighbors = nx.common_neighbors(nx.Graph(R),n1,n2) # Get common neighbors of the end points n_neighbors = len(list(neighbors)) # Count how many of these neighbors they have in common edge_weights.append(1+n_neighbors) # Add to the list (+1, in case no common neighbors) nx.draw(R, pos=pos, node_size=node_sizes, arrowstyle="->", fontsize=7, node_color = node_colors, edgelist = edge_ids, edge_color = edge_colors, nodelist = node_ids, width=edge_weights, # Supply the edge weights edge_labels = [str(i) for i in edge_weights])

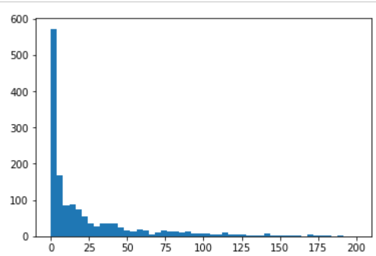
Exercise: How would you change the code so that random weights are drawn?

Note You can draw random numbers between zero and one using `random.random()` .

Answer: edge_weights = [] for e in edge_ids: edge_weights.append(2*random.random()) # Add a random weight nx.draw(R, pos=pos, node_size=node_sizes, arrowstyle="->", fontsize=7, node_color = node_colors, edgelist = edge_ids, edge_color = edge_colors, nodelist = node_ids, width=edge_weights, # Supply the edge weights edge_labels = [str(i) for i in edge_weights])

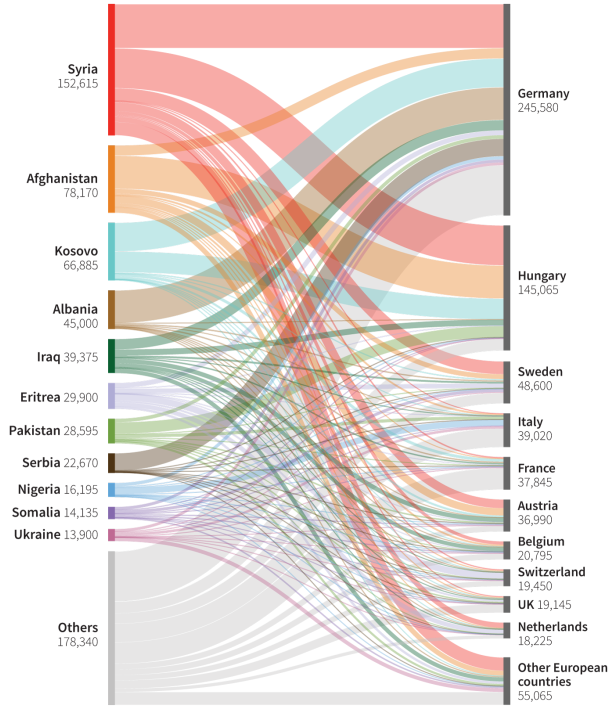
3.7.1 Degree histogram

In [50]:
```
# Get a list of all degrees
degrees = list(dict(P.degree()).values())

# Plot a histogram of the degree distribution
plt.hist(degrees, bins=50, range = [0,200])
plt.show()
```

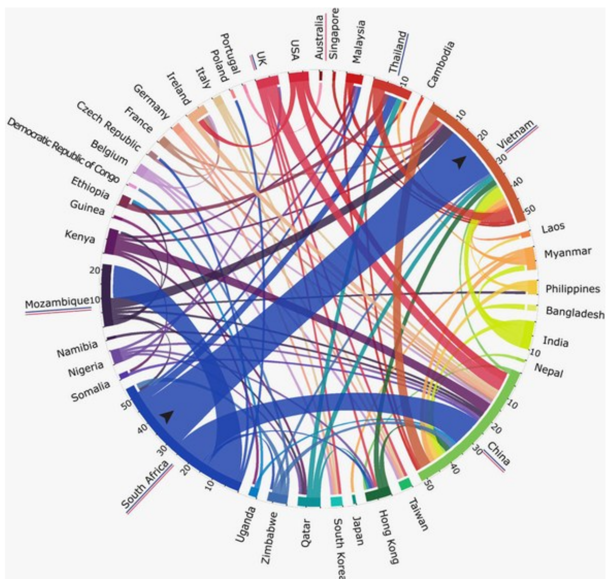


Today we have focused on simple network visualization charts, which are an easy and intuitve way to visualize and explore networks. However, it's worth noting that there are many different options.

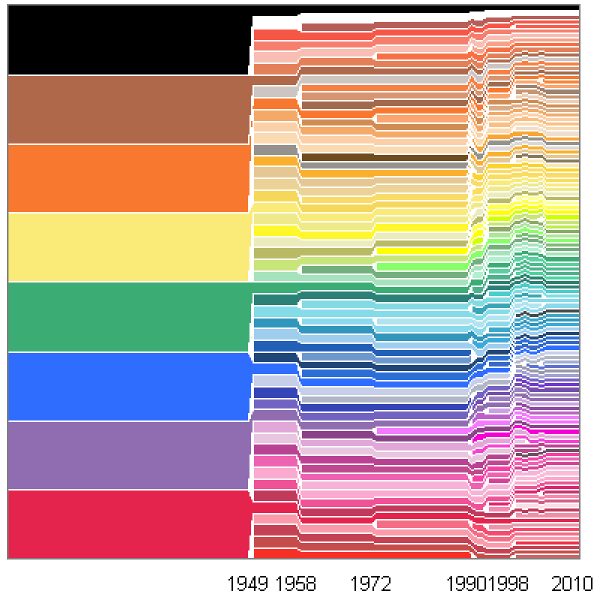
- **Sankey diagrams** are good for visualizing bipartite networks, such as [this visualization of European asylum-seekers (http://graphics.thomsonreuters.com/15/migrants/index.html#section-asylum)](http://graphics.thomsonreuters.com/15/migrants/index.html#section-asylum):



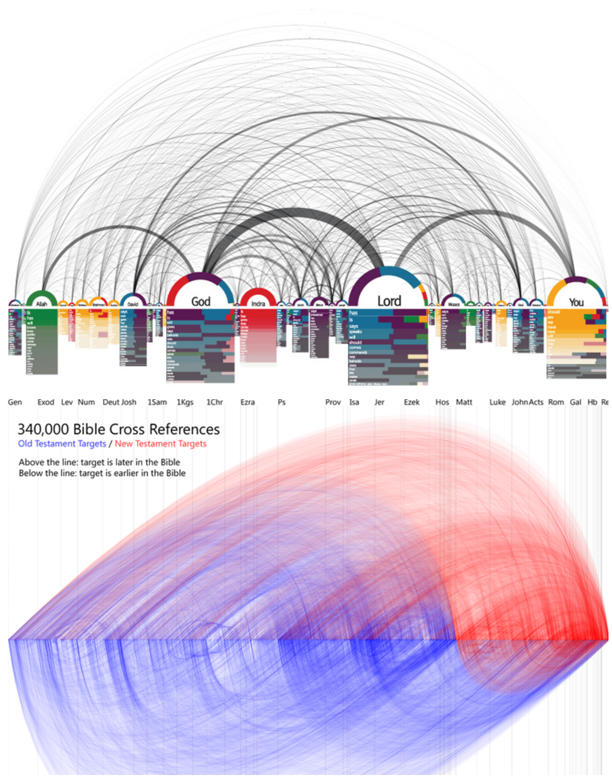
- **Chord charts** are helpful for visualizing flows, such as [this graphic on illegal wildlife trading (https://www.wired.com/2015/06/using-news-reports-track-wildlife-black-markets/)](https://www.wired.com/2015/06/using-news-reports-track-wildlife-black-markets/):



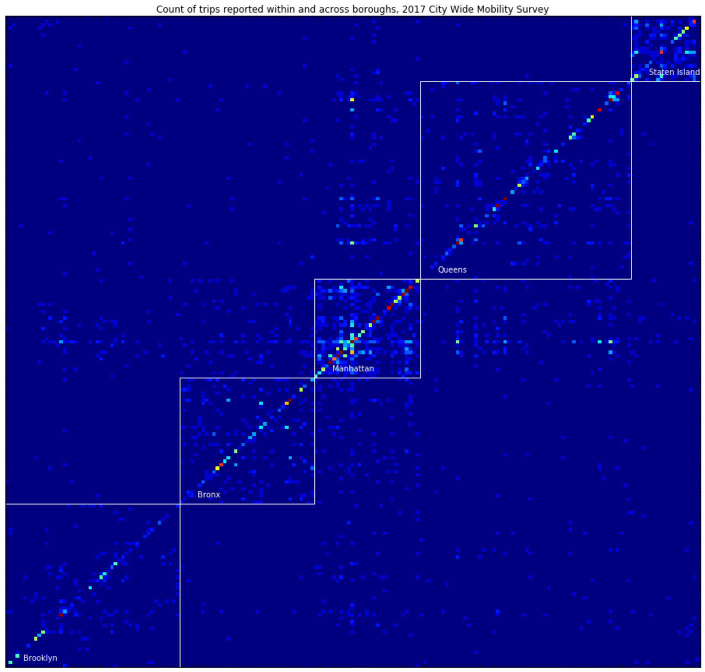
- **Trees** and dendrograms can be used for hierarchical data, such as [this illustration of Crayola crayon color evolution in time (https://blog.revolutionanalytics.com/2010/01/crayola-crayon-colors-1949present.html)](https://blog.revolutionanalytics.com/2010/01/crayola-crayon-colors-1949present.html):



- **Arc Diagrams** can help with linearly sequenced data, such as [these networks of biblical and religious references (https://www.theguardian.com/news/datablog/gallery/2013/sep/05/holy-infographics-bible-visualised)](https://www.theguardian.com/news/datablog/gallery/2013/sep/05/holy-infographics-bible-visualised):



- **Similarity matrices** can help to highlight clustering in densely connected networks:



More examples:

- https://python-graph-gallery.com (https://python-graph-gallery.com)
- https://flowingdata.com/category/visualization/network-visualization/ (https://flowingdata.com/category/visualization/network-visualization/)

The right visualization depends on the type of data you have, how much of it there is, and the key messages you want to highlight.