Module IV - Basic Analysis

Drew Conway and Aric Hagberg

June 29, 2010

Agenda for Module IV

Loading data from multiple sources

- Local network data files
- Building directly from the Internet

Agenda for Module IV

Loading data from multiple sources

- Local network data files
- Building directly from the Internet

Brief review of Python dictionaries

- Why is the `dict` so useful?
- How `NetworkX` utilizes it?

Agenda for Module IV

Loading data from multiple sources

- Local network data files
- Building directly from the Internet

Brief review of Python dictionaries

- Why is the `dict` so useful?
- How `NetworkX` utilizes it?

Running basic centralities

- Degree, Closeness, Betweeness Eigenvector
- Calculating degree distribution
- Plotting statistics using `matplotlib`
- Calculating cliques, clustering and transitivity

Agenda for Module IV

Loading data from multiple sources
- Local network data files
- Building directly from the Internet

Brief review of Python dictionaries
- Why is the `dict` so useful?
- How `NetworkX` utilizes it?

Running basic centralities
- Degree, Closeness, Betweeness Eigenvector
- Calculating degree distribution
- Plotting statistics using `matplotlib`
- Calculating cliques, clustering and transitivity

Outputting data into multiple formats
- Writing network data
- Saving network analysis statistics

Agenda for Module IV

Loading data from multiple sources
- Local network data files
- Building directly from the Internet

Brief review of Python dictionaries
- Why is the `dict` so useful?
- How `NetworkX` utilizes it?

Running basic centralities
- Degree, Closeness, Betweeness Eigenvector
- Calculating degree distribution
- Plotting statistics using `matplotlib`
- Calculating cliques, clustering and transitivity

Outputting data into multiple formats
- Writing network data
- Saving network analysis statistics

Basic visualization
- Review of `NetworkX`'s plotting algorithms
- Adding analysis to visualization

Loading a network file

As we have seen, one of the main advantages of working with `NetworkX` is that it can read many different network formats

- For those that are unfamiliar with working at the **command-line**, however, the process can be confusing

NX syntax for loading a file

$$\underset{\text{Net variable}}{\underset{\uparrow}{>>> G}} \quad = \quad \underset{\text{NX function, file directory path}}{\underset{\uparrow}{\text{nx.read\_format(“path/to/file.txt”,}}} \qquad \underset{\text{Graph type, nodes type, etc.}}{\underset{\uparrow}{\textit{...options...})}}$$

Loading a network file

As we have seen, one of the main advantages of working with `NetworkX` is that it can read many different network formats

- For those that are unfamiliar with working at the **command-line**, however, the process can be confusing

NX syntax for loading a file

$>>> G \quad = \quad$ nx.read_format("path/to/file.txt", $\quad$ *...options...*)

$\uparrow$ Net variable

$\uparrow$ NX function, file directory path

$\uparrow$ Graph type, nodes type, etc.

Loading a network file

As we have seen, one of the main advantages of working with `NetworkX` is that it can read many different network formats

- For those that are unfamiliar with working at the **command-line**, however, the process can be confusing

NX syntax for loading a file

$$\underset{\text{Net variable}}{\underset{\uparrow}{>>> G}} \quad = \quad \underset{\text{NX function, file directory path}}{\underset{\uparrow}{\text{nx.read\_format(``path/to/file.txt'',}}} \quad \underset{\text{Graph type, nodes type, etc.}}{\underset{\uparrow}{\textit{...options...})}}$$

Loading a network file

As we have seen, one of the main advantages of working with `NetworkX` is that it can read many different network formats

- For those that are unfamiliar with working at the **command-line**, however, the process can be confusing

NX syntax for loading a file

>>> G	=	nx.read_format("path/to/file.txt",	*...options...*)
$\uparrow$		$\uparrow$	$\uparrow$
Net variable		NX function, file directory path	Graph type, nodes type, etc.

Loading a network file

As we have seen, one of the main advantages of working with `NetworkX` is that it can read many different network formats

- For those that are unfamiliar with working at the **command-line**, however, the process can be confusing

NX syntax for loading a file

$$\underset{\substack{\uparrow \\ \text{Net variable}}}{>>> G} \quad = \quad \underset{\substack{\uparrow \\ \text{NX function, file directory path}}}{\text{nx.read\_format(“path/to/file.txt”,}} \quad \underset{\substack{\uparrow \\ \text{Graph type, nodes type, etc.}}}{\ldots\textit{options}\ldots\text{)}}$$

Let's try!

- We will load the edge list of Hartford drug users network
- Specify that the network be a directed graph, and the nodes be integers
- Use `nx.info()` to check that data has been loaded correctly

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> hartford=nx.read_edgelist("../../data/hartford_drug.txt",create_using=nx.DiGraph(),nodetype=int)
>>> nx.info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> hartford=nx.read_edgelist("../../data/hartford_drug.txt",create_using=nx.DiGraph(),nodetype=int)
>>> nx.info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

What did we just do?

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> hartford=nx.read_edgelist("../../data/hartford_drug.txt",create_using=nx.DiGraph(),nodetype=int)
>>> nx.info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

What did we just do?

- Used the `read_edgelist` function to load EL file

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> hartford=nx.read_edgelist("../../data/hartford_drug.txt",create_using=nx.DiGraph(),nodetype=int)
>>> nx.info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

What did we just do?

- Used the `read_edgelist` function to load EL file
- Specified path to Hartford drug users file

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> hartford=nx.read_edgelist("../../data/hartford_drug.txt",create_using=nx.DiGraph(),nodetype=int)
>>> nx.info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

What did we just do?

- Used the `read_edgelist` function to load EL file
- Specified path to Hartford drug users file
- Used the `create_using` option to force NX to create as a directed graph

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> hartford=nx.read_edgelist("../../data/hartford_drug.txt",create_using=nx.DiGraph(),nodetype=int)
>>> nx.info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

What did we just do?

- Used the `read_edgelist` function to load EL file
- Specified path to Hartford drug users file
- Used the `create_using` option to force NX to create as a directed graph
- Used the `nodetype` option to force NX to store nodes as integers

Loading the Hartford drug users network

Starting NetworkX and loading data

```
>>> hartford=nx.read_edgelist("../../data/hartford_drug.txt",create_using=nx.DiGraph(),nodetype=int)
>>> nx.info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

What did we just do?

- Used the `read_edgelist` function to load EL file
- Specified path to Hartford drug users file
- Used the `create_using` option to force NX to create as a directed graph
- Used the `nodetype` option to force NX to store nodes as integers
- Used the `info` function to check that it all worked

Loading the Hartford drug users network

Starting NetworkX and loading data

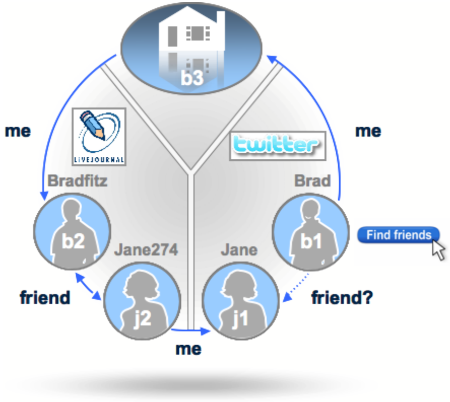
```
>>> hartford=nx.read_edgelist("../../data/hartford_drug.txt",create_using=nx.DiGraph(),nodetype=int)
>>> nx.info(hartford)
Name:
Type:                   DiGraph
Number of nodes:        212
Number of edges:        337
Average in degree:      1.5896
Average out degree:     1.5896
```

What did we just do?

- Used the `read_edgelist` function to load EL file
- Specified path to Hartford drug users file
- Used the `create_using` option to force NX to create as a directed graph
- Used the `nodetype` option to force NX to store nodes as integers
- Used the `info` function to check that it all worked

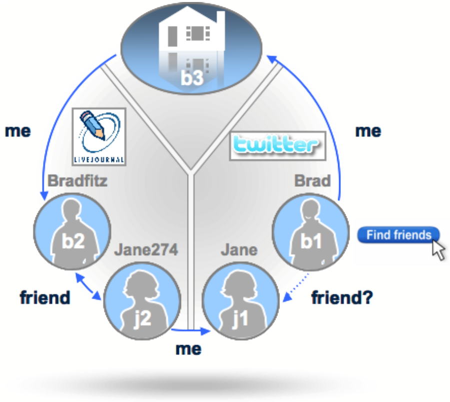
Some formats may have more or less options, **always check the documentations!**

Building the social network among LiveJournal users



Perhaps the most powerful aspect of `NetworkX` is its ability to work in Python to generate networks from live-streaming data

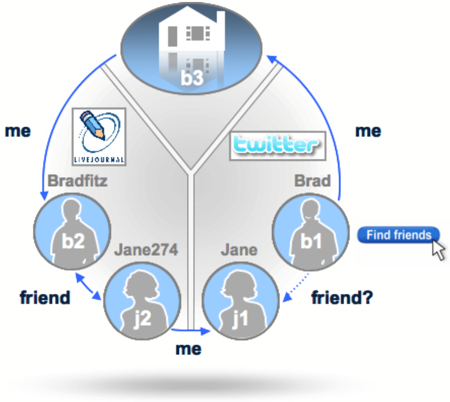
Building the social network among LiveJournal users



Perhaps the most powerful aspect of `NetworkX` is its ability to work in Python to generate networks from live-streaming data

- In Python, use `NetworkX`, `cjson` and a other standard scientific libraries to parse Google's SocialGraph data

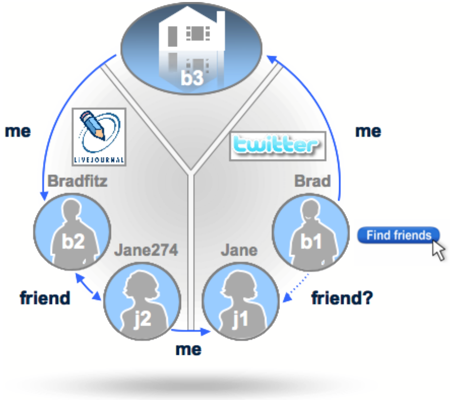
Building the social network among LiveJournal users



Perhaps the most powerful aspect of `NetworkX` is its ability to work in Python to generate networks from live-streaming data

- In Python, use `NetworkX`, `cjson` and a other standard scientific libraries to parse Google's SocialGraph data
- Using a "seed" user, we will build out a network

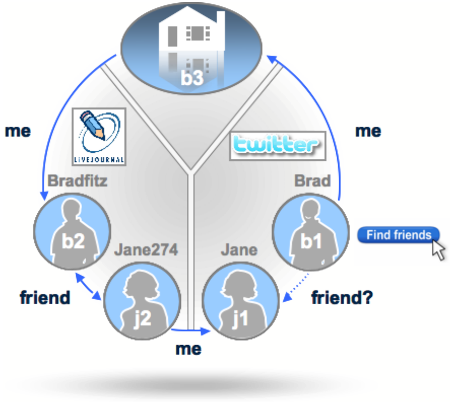
Building the social network among LiveJournal users



Perhaps the most powerful aspect of `NetworkX` is its ability to work in Python to generate networks from live-streaming data

- In Python, use `NetworkX`, `cjson` and a other standard scientific libraries to parse Google's SocialGraph data
- Using a "seed" user, we will build out a network
- Through a process called "k-snowball searching" $seed \longrightarrow friend \longrightarrow \cdots \longrightarrow friend_k$

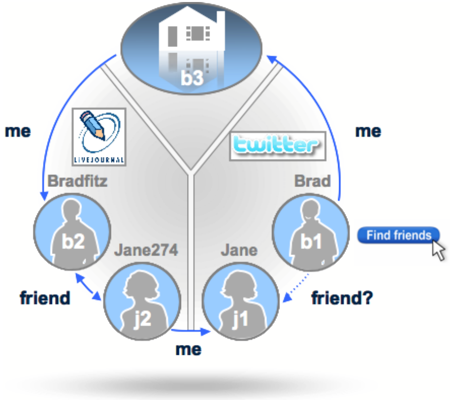
Building the social network among LiveJournal users



Perhaps the most powerful aspect of `NetworkX` is its ability to work in Python to generate networks from live-streaming data

- In Python, use `NetworkX`, `cjson` and a other standard scientific libraries to parse Google's SocialGraph data
- Using a "seed" user, we will build out a network
- Through a process called "k-snowball searching" $seed \longrightarrow friend \longrightarrow \cdots \longrightarrow friend_k$
  - Seed: imichaeldotorg.livejournal.com
  - $k = 3$

Building the social network among LiveJournal users



Perhaps the most powerful aspect of `NetworkX` is its ability to work in Python to generate networks from live-streaming data

- In Python, use `NetworkX`, `cjson` and a other standard scientific libraries to parse Google's SocialGraph data
- Using a "seed" user, we will build out a network
- Through a process called "k-snowball searching" $seed \longrightarrow friend \longrightarrow \cdots \longrightarrow friend_k$
  - Seed: imichaeldotorg.livejournal.com
  - $k = 3$
- Note the low value of k

The code, part 1

Loading the libraries and scraping egonet

```
from cjson import *
from urllib import *
from time import *
from scipy import array,unique
...
if __name__ == "__main__":
    seed="imichaeldotorg"
    seed_url="http://"+seed+".livejournal.com"
    # 3.1 Scrape, parse and build seed's ego net
    sg=get_sg(seed_url)
    net,newnodes=create_egonet(sg)
    nx.write_pajek(net,"../../data/"+seed+"_ego.net")
    nx.info(net)
```

```
def get_sg(seed_url):
  sgapi_url="http://socialgraph.apis.google.com/lookup?q="+seed_url+"&edo=1&edi=1&fme=1&pretty=0"
  try:
      furl=urlopen(sgapi_url)
      fr=furl.read()
      furl.close()
      return fr
  except IOError:
      print "Could not connect to website"
      print sgapi_url
      return
```

The code, part 1

Loading the libraries and scraping egonet

```
from cjson import *
from urllib import *
from time import *
from scipy import array,unique
...
if __name__ == "__main__":
    seed="imichaeldotorg"
    seed_url="http://"+seed+".livejournal.com"
    # 3.1 Scrape, parse and build seed's ego net
    sg=get_sg(seed_url)
    net,newnodes=create_egonet(sg)
    nx.write_pajek(net,"../../data/"+seed+"_ego.net")
    nx.info(net)
```

```
def get_sg(seed_url):
  sgapi_url="http://socialgraph.apis.google.com/lookup?q="+seed_url+"&edo=1&edi=1&fme=1&pretty=0"
  try:
      furl=urlopen(sgapi_url)
      fr=furl.read()
      furl.close()
      return fr
  except IOError:
      print "Could not connect to website"
      print sgapi_url
      return
```

The code, part 1

Loading the libraries and scraping egonet

```
from cjson import *
from urllib import *
from time import *
from scipy import array,unique
...
if __name__ == "__main__":
    seed="imichaeldotorg"
    seed_url="http://"+seed+".livejournal.com"
    # 3.1 Scrape, parse and build seed's ego net
    sg=get_sg(seed_url)
    net,newnodes=create_egonet(sg)
    nx.write_pajek(net,"../../data/"+seed+"_ego.net")
    nx.info(net)
```

```
def get_sg(seed_url):
  sgapi_url="http://socialgraph.apis.google.com/lookup?q="+seed_url+"&edo=1&edi=1&fme=1&pretty=0"
  try:
      furl=urlopen(sgapi_url)
      fr=furl.read()
      furl.close()
      return fr
  except IOError:
      print "Could not connect to website"
      print sgapi_url
      return
```

The code, part 1

Loading the libraries and scraping egonet

```
from cjson import *
from urllib import *
from time import *
from scipy import array,unique
...
if __name__ == "__main__":
    seed="imichaeldotorg"
    seed_url="http://"+seed+".livejournal.com"
    # 3.1 Scrape, parse and build seed's ego net
    sg=get_sg(seed_url)
    net,newnodes=create_egonet(sg)
    nx.write_pajek(net,"../../data/"+seed+"_ego.net")
    nx.info(net)
```

```
def get_sg(seed_url):
  sgapi_url="http://socialgraph.apis.google.com/lookup?q="+seed_url+"&edo=1&edi=1&fme=1&pretty=0"
  try:
      furl=urlopen(sgapi_url)
      fr=furl.read()
      furl.close()
      return fr
  except IOError:
      print "Could not connect to website"
      print sgapi_url
      return
```

The code, part 1

Loading the libraries and scraping egonet

```
from cjson import *
from urllib import *
from time import *
from scipy import array,unique
...
if __name__ == "__main__":
    seed="imichaeldotorg"
    seed_url="http://"+seed+".livejournal.com"
    # 3.1 Scrape, parse and build seed's ego net
    sg=get_sg(seed_url)
    net,newnodes=create_egonet(sg)
    nx.write_pajek(net,"../../data/"+seed+"_ego.net")
    nx.info(net)
```

```
def get_sg(seed_url):
  sgapi_url="http://socialgraph.apis.google.com/lookup?q="+seed_url+"&edo=1&edi=1&fme=1&pretty=0"
  try:
      furl=urlopen(sgapi_url)
      fr=furl.read()
      furl.close()
      return fr
  except IOError:
      print "Could not connect to website"
      print sgapi_url
      return
```

The code, part 1

Loading the libraries and scraping egonet

```
from cjson import *
from urllib import *
from time import *
from scipy import array,unique
...
if __name__ == "__main__":
    seed="imichaeldotorg"
    seed_url="http://"+seed+".livejournal.com"
    # 3.1 Scrape, parse and build seed's ego net
    sg=get_sg(seed_url)
    net,newnodes=create_egonet(sg)
    nx.write_pajek(net,"../../data/"+seed+"_ego.net")
    nx.info(net)
```

```
def get_sg(seed_url):
  sgapi_url="http://socialgraph.apis.google.com/lookup?q="+seed_url+"&edo=1&edi=1&fme=1&pretty=0"
  try:
      furl=urlopen(sgapi_url)
      fr=furl.read()
      furl.close()
      return fr
  except IOError:
      print "Could not connect to website"
      print sgapi_url
      return
```

The code, part 1

Loading the libraries and scraping egonet

```
from cjson import *
from urllib import *
from time import *
from scipy import array,unique
...
if __name__ == "__main__":
    seed="imichaeldotorg"
    seed_url="http://"+seed+".livejournal.com"
    # 3.1 Scrape, parse and build seed's ego net
    sg=get_sg(seed_url)
    net,newnodes=create_egonet(sg)
    nx.write_pajek(net,"../../data/"+seed+"_ego.net")
    nx.info(net)
```

```
Name:                  ['http://imichaeldotorg.livejournal.com/']
Type:                  DiGraph
Number of nodes:       5
Number of edges:       5
Average in degree:     1.0
Average out degree:    1.0
```

```
def get_sg(seed_url):
  sgapi_url="http://socialgraph.apis.google.com/lookup?q="+seed_url+"&edo=1&edi=1&fme=1&pretty=0"
  try:
      furl=urlopen(sgapi_url)
      fr=furl.read()
      furl.close()
      return fr
  except IOError:
      print "Could not connect to website"
      print sgapi_url
      return
```

Build egonet and snowball

Creating the egonet

```
def create_egonet(s):
    try:
        raw=decode(s)
        G=nx.DiGraph()
        pendants=[]
        n=raw['nodes']
        nk=n.keys()
        G.name=str(nk)
        pendants=[]
        for a in range(0,len(nk)):
            for b in range(0,len(nk)):
                if a!=b:
                    G.add_edge(nk[a],nk[b])
        for k in nk:
            ego=n[k]
            ego_out=ego['nodes_referenced']
            for o in ego_out:
                G.add_edge(k,o)
                pendants.append(o)
            ego_in=ego['nodes_referenced_by']
            for i in ego_in:
                G.add_edge(i,k)
                pendants.append(i)
        pendants=array(pendants,dtype=str)
        pendants.flatten()
        pendants=unique(pendants)
        return G,pendants
    except DecodeError:
    ...
    except KeyError:
```

Rolling the snowball

```
def snowball_round(G,seeds,myspace=False):
    t0=time()
    if myspace:
        seeds=get_myspace_url(seeds)
    sb_data=[]
    for s in range(0,len(seeds)):
        s_sg=get_sg(seeds[s])
        new_ego,pen=create_egonet(s_sg)
        for p in pen:
                sb_data.append(p)
        if s<1:
            sb_net=nx.compose(G,new_ego)
        else:
            sb_net=nx.compose(new_ego,sb_net)
        del new_ego
        if s==round(len(seeds)*0.2):
            sb_net.name='20% complete'
            nx.info(sb_net)
            print 'AT: '+strftime('%m/%d/%Y, %H:%M:%S', gmtime())
            print ''
    ...
    # More time keeping, probably a MUCH better way to do this
    sb_data=array(sb_data)
    sb_data.flatten()
    sb_data=unique(sb_data)
    nx.info(sb_net)
    return sb_net,sb_data
```

Build egonet and snowball

Creating the egonet

```
def create_egonet(s):
    try:
        raw=decode(s)
        G=nx.DiGraph()
        pendants=[]
        n=raw['nodes']
        nk=n.keys()
        G.name=str(nk)
        pendants=[]
        for a in range(0,len(nk)):
            for b in range(0,len(nk)):
                if a!=b:
                    G.add_edge(nk[a],nk[b])
        for k in nk:
            ego=n[k]
            ego_out=ego['nodes_referenced']
            for o in ego_out:
                G.add_edge(k,o)
                pendants.append(o)
            ego_in=ego['nodes_referenced_by']
            for i in ego_in:
                G.add_edge(i,k)
                pendants.append(i)
        pendants=array(pendants,dtype=str)
        pendants.flatten()
        pendants=unique(pendants)
        return G,pendants
    except DecodeError:
    ...
    except KeyError:
```

Rolling the snowball

```
def snowball_round(G,seeds,myspace=False):
    t0=time()
    if myspace:
        seeds=get_myspace_url(seeds)
    sb_data=[]
    for s in range(0,len(seeds)):
        s_sg=get_sg(seeds[s])
        new_ego,pen=create_egonet(s_sg)
        for p in pen:
                sb_data.append(p)
        if s<1:
            sb_net=nx.compose(G,new_ego)
        else:
            sb_net=nx.compose(new_ego,sb_net)
        del new_ego
        if s==round(len(seeds)*0.2):
            sb_net.name='20% complete'
            nx.info(sb_net)
            print 'AT: '+strftime('%m/%d/%Y, %H:%M:%S', gmtime())
            print ''
    ...
    # More time keeping, probably a MUCH better way to do this
    sb_data=array(sb_data)
    sb_data.flatten()
    sb_data=unique(sb_data)
    nx.info(sb_net)
    return sb_net,sb_data
```

Build egonet and snowball

Creating the egonet

```
def create_egonet(s):
    try:
        raw=decode(s)
        G=nx.DiGraph()
        pendants=[]
        n=raw['nodes']
        nk=n.keys()
        G.name=str(nk)
        pendants=[]
        for a in range(0,len(nk)):
            for b in range(0,len(nk)):
                if a!=b:
                    G.add_edge(nk[a],nk[b])
        for k in nk:
            ego=n[k]
            ego_out=ego['nodes_referenced']
            for o in ego_out:
                G.add_edge(k,o)
                pendants.append(o)
            ego_in=ego['nodes_referenced_by']
            for i in ego_in:
                G.add_edge(i,k)
                pendants.append(i)
        pendants=array(pendants,dtype=str)
        pendants.flatten()
        pendants=unique(pendants)
        return G,pendants
    except DecodeError:
    ...
    except KeyError:
```

Rolling the snowball

```
def snowball_round(G,seeds,myspace=False):
    t0=time()
    if myspace:
        seeds=get_myspace_url(seeds)
    sb_data=[]
    for s in range(0,len(seeds)):
        s_sg=get_sg(seeds[s])
        new_ego,pen=create_egonet(s_sg)
        for p in pen:
                sb_data.append(p)
        if s<1:
            sb_net=nx.compose(G,new_ego)
        else:
            sb_net=nx.compose(new_ego,sb_net)
        del new_ego
        if s==round(len(seeds)*0.2):
            sb_net.name='20% complete'
            nx.info(sb_net)
            print 'AT: '+strftime('%m/%d/%Y, %H:%M:%S', gmtime())
            print ''
    ...
    # More time keeping, probably a MUCH better way to do this
    sb_data=array(sb_data)
    sb_data.flatten()
    sb_data=unique(sb_data)
    nx.info(sb_net)
    return sb_net,sb_data
```

Build egonet and snowball

Creating the egonet

```
def create_egonet(s):
    try:
        raw=decode(s)
        G=nx.DiGraph()
        pendants=[]
        n=raw['nodes']
        nk=n.keys()
        G.name=str(nk)
        pendants=[]
        for a in range(0,len(nk)):
            for b in range(0,len(nk)):
                if a!=b:
                    G.add_edge(nk[a],nk[b])
        for k in nk:
            ego=n[k]
            ego_out=ego['nodes_referenced']
            for o in ego_out:
                G.add_edge(k,o)
                pendants.append(o)
            ego_in=ego['nodes_referenced_by']
            for i in ego_in:
                G.add_edge(i,k)
                pendants.append(i)
        pendants=array(pendants,dtype=str)
        pendants.flatten()
        pendants=unique(pendants)
        return G,pendants
    except DecodeError:
    ...
    except KeyError:
```

Rolling the snowball

```
def snowball_round(G,seeds,myspace=False):
    t0=time()
    if myspace:
        seeds=get_myspace_url(seeds)
    sb_data=[]
    for s in range(0,len(seeds)):
        s_sg=get_sg(seeds[s])
        new_ego,pen=create_egonet(s_sg)
        for p in pen:
                sb_data.append(p)
        if s<1:
            sb_net=nx.compose(G,new_ego)
        else:
            sb_net=nx.compose(new_ego,sb_net)
        del new_ego
        if s==round(len(seeds)*0.2):
            sb_net.name='20% complete'
            nx.info(sb_net)
            print 'AT: '+strftime('%m/%d/%Y, %H:%M:%S', gmtime())
            print ''
    ...
    # More time keeping, probably a MUCH better way to do this
    sb_data=array(sb_data)
    sb_data.flatten()
    sb_data=unique(sb_data)
    nx.info(sb_net)
    return sb_net,sb_data
```

Build egonet and snowball

Creating the egonet

```
def create_egonet(s):
    try:
        raw=decode(s)
        G=nx.DiGraph()
        pendants=[]
        n=raw['nodes']
        nk=n.keys()
        G.name=str(nk)
        pendants=[]
        for a in range(0,len(nk)):
            for b in range(0,len(nk)):
                if a!=b:
                    G.add_edge(nk[a],nk[b])
        for k in nk:
            ego=n[k]
            ego_out=ego['nodes_referenced']
            for o in ego_out:
                G.add_edge(k,o)
                pendants.append(o)
            ego_in=ego['nodes_referenced_by']
            for i in ego_in:
                G.add_edge(i,k)
                pendants.append(i)
        pendants=array(pendants,dtype=str)
        pendants.flatten()
        pendants=unique(pendants)
        return G,pendants
    except DecodeError:
    ...
    except KeyError:
```

Rolling the snowball

```
def snowball_round(G,seeds,myspace=False):
    t0=time()
    if myspace:
        seeds=get_myspace_url(seeds)
    sb_data=[]
    for s in range(0,len(seeds)):
        s_sg=get_sg(seeds[s])
        new_ego,pen=create_egonet(s_sg)
        for p in pen:
                sb_data.append(p)
        if s<1:
            sb_net=nx.compose(G,new_ego)
        else:
            sb_net=nx.compose(new_ego,sb_net)
        del new_ego
        if s==round(len(seeds)*0.2):
            sb_net.name='20% complete'
            nx.info(sb_net)
            print 'AT: '+strftime('%m/%d/%Y, %H:%M:%S', gmtime())
            print ''
    ...
    # More time keeping, probably a MUCH better way to do this
    sb_data=array(sb_data)
    sb_data.flatten()
    sb_data=unique(sb_data)
    nx.info(sb_net)
    return sb_net,sb_data
```

Build egonet and snowball

Creating the egonet

```
def create_egonet(s):
    try:
        raw=decode(s)
        G=nx.DiGraph()
        pendants=[]
        n=raw['nodes']
        nk=n.keys()
        G.name=str(nk)
        pendants=[]
        for a in range(0,len(nk)):
            for b in range(0,len(nk)):
                if a!=b:
                    G.add_edge(nk[a],nk[b])
        for k in nk:
            ego=n[k]
            ego_out=ego['nodes_referenced']
            for o in ego_out:
                G.add_edge(k,o)
                pendants.append(o)
            ego_in=ego['nodes_referenced_by']
            for i in ego_in:
                G.add_edge(i,k)
                pendants.append(i)
        pendants=array(pendants,dtype=str)
        pendants.flatten()
        pendants=unique(pendants)
        return G,pendants
    except DecodeError:
    ...
    except KeyError:
```

Rolling the snowball

```
def snowball_round(G,seeds,myspace=False):
    t0=time()
    if myspace:
        seeds=get_myspace_url(seeds)
    sb_data=[]
    for s in range(0,len(seeds)):
        s_sg=get_sg(seeds[s])
        new_ego,pen=create_egonet(s_sg)
        for p in pen:
                sb_data.append(p)
        if s<1:
            sb_net=nx.compose(G,new_ego)
        else:
            sb_net=nx.compose(new_ego,sb_net)
        del new_ego
        if s==round(len(seeds)*0.2):
            sb_net.name='20% complete'
            nx.info(sb_net)
            print 'AT: '+strftime('%m/%d/%Y, %H:%M:%S', gmtime())
            print ''
    ...
    # More time keeping, probably a MUCH better way to do this
    sb_data=array(sb_data)
    sb_data.flatten()
    sb_data=unique(sb_data)
    nx.info(sb_net)
    return sb_net,sb_data
```

Build egonet and snowball

Creating the egonet

```
def create_egonet(s):
    try:
        raw=decode(s)
        G=nx.DiGraph()
        pendants=[]
        n=raw['nodes']
        nk=n.keys()
        G.name=str(nk)
        pendants=[]
        for a in range(0,len(nk)):
            for b in range(0,len(nk)):
                if a!=b:
                    G.add_edge(nk[a],nk[b])
        for k in nk:
            ego=n[k]
            ego_out=ego['nodes_referenced']
            for o in ego_out:
                G.add_edge(k,o)
                pendants.append(o)
            ego_in=ego['nodes_referenced_by']
            for i in ego_in:
                G.add_edge(i,k)
                pendants.append(i)
        pendants=array(pendants,dtype=str)
        pendants.flatten()
        pendants=unique(pendants)
        return G,pendants
    except DecodeError:
    ...
    except KeyError:
```

Rolling the snowball

```
def snowball_round(G,seeds,myspace=False):
    t0=time()
    if myspace:
        seeds=get_myspace_url(seeds)
    sb_data=[]
    for s in range(0,len(seeds)):
        s_sg=get_sg(seeds[s])
        new_ego,pen=create_egonet(s_sg)
        for p in pen:
                sb_data.append(p)
        if s<1:
            sb_net=nx.compose(G,new_ego)
        else:
            sb_net=nx.compose(new_ego,sb_net)
        del new_ego
        if s==round(len(seeds)*0.2):
            sb_net.name='20% complete'
            nx.info(sb_net)
            print 'AT: '+strftime('%m/%d/%Y, %H:%M:%S', gmtime())
            print ''
    ...
    # More time keeping, probably a MUCH better way to do this
    sb_data=array(sb_data)
    sb_data.flatten()
    sb_data=unique(sb_data)
    nx.info(sb_net)
    return sb_net,sb_data
```

Build egonet and snowball

Creating the egonet

```
def create_egonet(s):
    try:
        raw=decode(s)
        G=nx.DiGraph()
        pendants=[]
        n=raw['nodes']
        nk=n.keys()
        G.name=str(nk)
        pendants=[]
        for a in range(0,len(nk)):
            for b in range(0,len(nk)):
                if a!=b:
                    G.add_edge(nk[a],nk[b])
        for k in nk:
            ego=n[k]
            ego_out=ego['nodes_referenced']
            for o in ego_out:
                G.add_edge(k,o)
                pendants.append(o)
            ego_in=ego['nodes_referenced_by']
            for i in ego_in:
                G.add_edge(i,k)
                pendants.append(i)
        pendants=array(pendants,dtype=str)
        pendants.flatten()
        pendants=unique(pendants)
        return G,pendants
    except DecodeError:
    ...
    except KeyError:
```

Rolling the snowball

```
def snowball_round(G,seeds,myspace=False):
    t0=time()
    if myspace:
        seeds=get_myspace_url(seeds)
    sb_data=[]
    for s in range(0,len(seeds)):
        s_sg=get_sg(seeds[s])
        new_ego,pen=create_egonet(s_sg)
        for p in pen:
                sb_data.append(p)
        if s<1:
            sb_net=nx.compose(G,new_ego)
        else:
            sb_net=nx.compose(new_ego,sb_net)
        del new_ego
        if s==round(len(seeds)*0.2):
            sb_net.name='20% complete'
            nx.info(sb_net)
            print 'AT: '+strftime('%m/%d/%Y, %H:%M:%S', gmtime())
            print ''
    ...
    # More time keeping, probably a MUCH better way to do this
    sb_data=array(sb_data)
    sb_data.flatten()
    sb_data=unique(sb_data)
    nx.info(sb_net)
    return sb_net,sb_data
```

Build egonet and snowball

Creating the egonet

```
def create_egonet(s):
    try:
        raw=decode(s)
        G=nx.DiGraph()
        pendants=[]
        n=raw['nodes']
        nk=n.keys()
        G.name=str(nk)
        pendants=[]
        for a in range(0,len(nk)):
            for b in range(0,len(nk)):
                if a!=b:
                    G.add_edge(nk[a],nk[b])
        for k in nk:
            ego=n[k]
            ego_out=ego['nodes_referenced']
            for o in ego_out:
                G.add_edge(k,o)
                pendants.append(o)
            ego_in=ego['nodes_referenced_by']
            for i in ego_in:
                G.add_edge(i,k)
                pendants.append(i)
        pendants=array(pendants,dtype=str)
        pendants.flatten()
        pendants=unique(pendants)
        return G,pendants
    except DecodeError:
    ...
    except KeyError:
```

Rolling the snowball

```
def snowball_round(G,seeds,myspace=False):
    t0=time()
    if myspace:
        seeds=get_myspace_url(seeds)
    sb_data=[]
    for s in range(0,len(seeds)):
        s_sg=get_sg(seeds[s])
        new_ego,pen=create_egonet(s_sg)
        for p in pen:
                sb_data.append(p)
        if s<1:
            sb_net=nx.compose(G,new_ego)
        else:
            sb_net=nx.compose(new_ego,sb_net)
        del new_ego
        if s==round(len(seeds)*0.2):
            sb_net.name='20% complete'
            nx.info(sb_net)
            print 'AT: '+strftime('%m/%d/%Y, %H:%M:%S', gmtime())
            print ''
    ...
    # More time keeping, probably a MUCH better way to do this
    sb_data=array(sb_data)
    sb_data.flatten()
    sb_data=unique(sb_data)
    nx.info(sb_net)
    return sb_net,sb_data
```

Build egonet and snowball

Creating the egonet

```
def create_egonet(s):
    try:
        raw=decode(s)
        G=nx.DiGraph()
        pendants=[]
        n=raw['nodes']
        nk=n.keys()
        G.name=str(nk)
        pendants=[]
        for a in range(0,len(nk)):
            for b in range(0,len(nk)):
                if a!=b:
                    G.add_edge(nk[a],nk[b])
        for k in nk:
            ego=n[k]
            ego_out=ego['nodes_referenced']
            for o in ego_out:
                G.add_edge(k,o)
                pendants.append(o)
            ego_in=ego['nodes_referenced_by']
            for i in ego_in:
                G.add_edge(i,k)
                pendants.append(i)
        pendants=array(pendants,dtype=str)
        pendants.flatten()
        pendants=unique(pendants)
        return G,pendants
    except DecodeError:
    ...
    except KeyError:
```

Rolling the snowball

```
def snowball_round(G,seeds,myspace=False):
    t0=time()
    if myspace:
        seeds=get_myspace_url(seeds)
    sb_data=[]
    for s in range(0,len(seeds)):
        s_sg=get_sg(seeds[s])
        new_ego,pen=create_egonet(s_sg)
        for p in pen:
                sb_data.append(p)
        if s<1:
            sb_net=nx.compose(G,new_ego)
        else:
            sb_net=nx.compose(new_ego,sb_net)
        del new_ego
        if s==round(len(seeds)*0.2):
            sb_net.name='20% complete'
            nx.info(sb_net)
            print 'AT: '+strftime('%m/%d/%Y, %H:%M:%S', gmtime())
            print ''
    ...
    # More time keeping, probably a MUCH better way to do this
    sb_data=array(sb_data)
    sb_data.flatten()
    sb_data=unique(sb_data)
    nx.info(sb_net)
    return sb_net,sb_data
```

Build egonet and snowball

Creating the egonet

```
def create_egonet(s):
    try:
        raw=decode(s)
        G=nx.DiGraph()
        pendants=[]
        n=raw['nodes']
        nk=n.keys()
        G.name=str(nk)
        pendants=[]
        for a in range(0,len(nk)):
            for b in range(0,len(nk)):
                if a!=b:
                    G.add_edge(nk[a],nk[b])
        for k in nk:
            ego=n[k]
            ego_out=ego['nodes_referenced']
            for o in ego_out:
                G.add_edge(k,o)
                pendants.append(o)
            ego_in=ego['nodes_referenced_by']
            for i in ego_in:
                G.add_edge(i,k)
                pendants.append(i)
        pendants=array(pendants,dtype=str)
        pendants.flatten()
        pendants=unique(pendants)
        return G,pendants
    except DecodeError:
    ...
    except KeyError:
```

Rolling the snowball

```
def snowball_round(G,seeds,myspace=False):
    t0=time()
    if myspace:
        seeds=get_myspace_url(seeds)
    sb_data=[]
    for s in range(0,len(seeds)):
        s_sg=get_sg(seeds[s])
        new_ego,pen=create_egonet(s_sg)
        for p in pen:
                sb_data.append(p)
        if s<1:
            sb_net=nx.compose(G,new_ego)
        else:
            sb_net=nx.compose(new_ego,sb_net)
        del new_ego
        if s==round(len(seeds)*0.2):
            sb_net.name='20% complete'
            nx.info(sb_net)
            print 'AT: '+strftime('%m/%d/%Y, %H:%M:%S', gmtime())
            print ''
    ...
    # More time keeping, probably a MUCH better way to do this
    sb_data=array(sb_data)
    sb_data.flatten()
    sb_data=unique(sb_data)
    nx.info(sb_net)
    return sb_net,sb_data
```

Build egonet and snowball

Creating the egonet

```
def create_egonet(s):
    try:
        raw=decode(s)
        G=nx.DiGraph()
        pendants=[]
        n=raw['nodes']
        nk=n.keys()
        G.name=str(nk)
        pendants=[]
        for a in range(0,len(nk)):
            for b in range(0,len(nk)):
                if a!=b:
                    G.add_edge(nk[a],nk[b])
        for k in nk:
            ego=n[k]
            ego_out=ego['nodes_referenced']
            for o in ego_out:
                G.add_edge(k,o)
                pendants.append(o)
            ego_in=ego['nodes_referenced_by']
            for i in ego_in:
                G.add_edge(i,k)
                pendants.append(i)
        pendants=array(pendants,dtype=str)
        pendants.flatten()
        pendants=unique(pendants)
        return G,pendants
    except DecodeError:
    ...
    except KeyError:
```

Rolling the snowball

```
def snowball_round(G,seeds,myspace=False):
    t0=time()
    if myspace:
        seeds=get_myspace_url(seeds)
    sb_data=[]
    for s in range(0,len(seeds)):
        s_sg=get_sg(seeds[s])
        new_ego,pen=create_egonet(s_sg)
        for p in pen:
                sb_data.append(p)
        if s<1:
            sb_net=nx.compose(G,new_ego)
        else:
            sb_net=nx.compose(new_ego,sb_net)
        del new_ego
        if s==round(len(seeds)*0.2):
            sb_net.name='20% complete'
            nx.info(sb_net)
            print 'AT: '+strftime('%m/%d/%Y, %H:%M:%S', gmtime())
            print ''
    ...
    # More time keeping, probably a MUCH better way to do this
    sb_data=array(sb_data)
    sb_data.flatten()
    sb_data=unique(sb_data)
    nx.info(sb_net)
    return sb_net,sb_data
```

Build the whole network

Step	Nodes	Edges	Mean Degree	Density
Seed	5	5	2.0	0.25
$k = 2$	75	115	3.0	0.02
$k = 3$	4,938	8,659	3.5	$3.6(10^{-4})$

Build the whole network

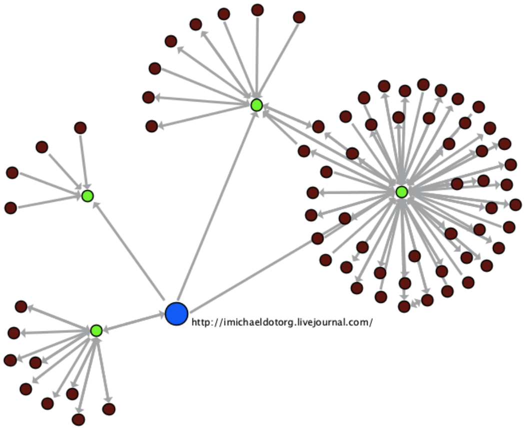
Step	Nodes	Edges	Mean Degree	Density
Seed	5	5	2.0	0.25
$k = 2$	75	115	3.0	0.02
$k = 3$	4,938	8,659	3.5	$3.6(10^{-4})$

- Our seed is abnormally isolated, with only four neighbors

Build the whole network

Step	Nodes	Edges	Mean Degree	Density
Seed	5	5	2.0	0.25
$k = 2$	75	115	3.0	0.02
$k = 3$	4,938	8,659	3.5	$3.6(10^{-4})$

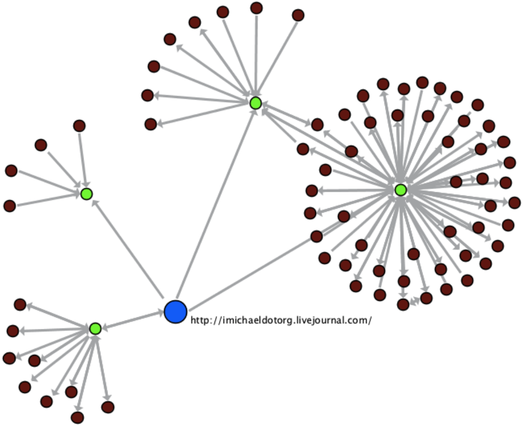
- Our seed is abnormally isolated, with only four neighbors
- Large jump after first snowball



Build the whole network

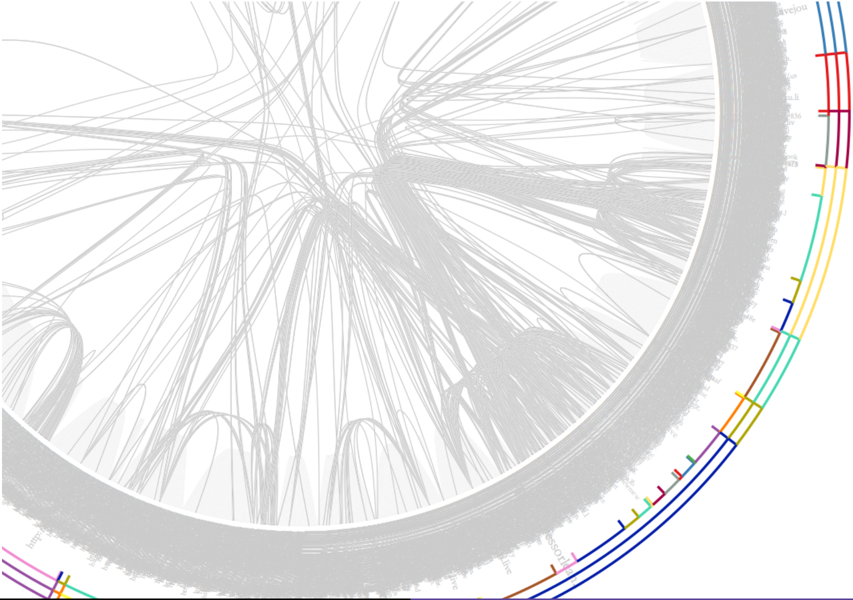
Step	Nodes	Edges	Mean Degree	Density
Seed	5	5	2.0	0.25
$k = 2$	75	115	3.0	0.02
$k = 3$	4,938	8,659	3.5	$3.6(10^{-4})$

- Our seed is abnormally isolated, with only four neighbors
- Large jump after first snowball
- Massive structural leap at $k = 3$



The full network

To get a feeling for the size of the full network...



Python Dictionaries

The `dict` type is a data structure that represents a key$\rightarrow$value mapping

Python Dictionaries

The `dict` type is a data structure that represents a key→value mapping

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }
```

Python Dictionaries

The dict type is a data structure that represents a key→value mapping

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }

# Can retrieve the keys and values as Python lists (vector)
>>> fruit_dict.keys()
["orange","apple","banana"]
```

The dict type is a data structure that represents a key→value mapping

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }

# Can retrieve the keys and values as Python lists (vector)
>>> fruit_dict.keys()
["orange","apple","banana"]

# Or create a (key,value) tuple
>>> fruit_dict.items()
[("orange",[0.23,0.11]),("apple",1),("Banana",True)]
```

The dict type is a data structure that represents a key→value mapping

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }

# Can retrieve the keys and values as Python lists (vector)
>>> fruit_dict.keys()
["orange","apple","banana"]

# Or create a (key,value) tuple
>>> fruit_dict.items()
[("orange",[0.23,0.11]),("apple",1),("Banana",True)]
# This becomes especially useful when you master Python ''list comprehension''
```

Python Dictionaries

The dict type is a data structure that represents a key→value mapping

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }

# Can retrieve the keys and values as Python lists (vector)
>>> fruit_dict.keys()
["orange","apple","banana"]

# Or create a (key,value) tuple
>>> fruit_dict.items()
[("orange",[0.23,0.11]),("apple",1),("Banana",True)]
# This becomes especially useful when you master Python ``list comprehension''
```

The Python dictionary is an extremely flexible and useful data structure, making it one of the primary advantages of Python over other languages

- This is particularly useful when performing analysis on networks, where node labels are natural keys

The `dict` type is a data structure that represents a key→value mapping

Working with the dict type

```
# Keys and values can be of any data type
>>> fruit_dict={"apple":1,"orange":[0.23,0.11],"banana":True }

# Can retrieve the keys and values as Python lists (vector)
>>> fruit_dict.keys()
["orange","apple","banana"]

# Or create a (key,value) tuple
>>> fruit_dict.items()
[("orange",[0.23,0.11]),("apple",1),("Banana",True)]
# This becomes especially useful when you master Python ``list comprehension''
```

The Python dictionary is an extremely flexible and useful data structure, making it one of the primary advantages of Python over other languages

- This is particularly useful when performing analysis on networks, where node labels are natural keys

Now, try creating a `dict` of your own

Using dictionaries for network analysis

From the documentation...

networkx.closeness_centrality

closeness_centrality(*G*, *v=None*, *weighted_edges=False*)

Compute closeness centrality for nodes.

Closeness centrality at a node is 1/average distance to all other nodes.

Parameters: **G** : graph

A networkx graph

v : node, optional

Return only the value for node v.

weighted_edges : bool, optional

Consider the edge weights in determining the shortest paths. If False, all edge weights are considered equal.

Returns: **nodes** : dictionary

Dictionary of nodes with closeness centrality as the value.

`NetworkX`'s metric's make extensive use of the dict type

- In this case the key→value mapping is of the form: {`node_label: metric`}

Let's look at an example:

```
>>> in_cen=nx.in_degree_centrality(hartford)
>>> in_cen
{1: 0.014218009478672987, 2: 0.018957345971563982,...
...
90: 0.0047393364928909956, 293: 0.0}
```

We can see that node #90 has in-degree centrality 0.0047

- But we can do so much more!

Running multiple measures

For our first analysis in NetworkX, we will do some basic network manipulation, then run multiple measures to find highest centrality nodes

- First, we will need to convert to an undirected network, and extract the main component

```
# Many of the centrality metrics require undirected graphs, so we will symmetrize
>>> hartford_ud=hartford.to_undirected()
# The network also has many small components, but for
# this analysis we are interested in the largest
>>> hartford_mc=hartford_main=nx.connected_component_subgraphs(hartford_ud)[0]
```

Next, we will calculate multiple measures

```
# Betweenness centrality
>>> bet_cen=nx.betweenness_centrality(hartford_mc)
# Closeness centrality
>>> clo_cen=nx.closeness_centrality(hartford_mc)
# Eigenvector centrality
>>> eig_cen=nx.eigenvector_centrality(hartford_mc)
```

Running multiple measures

For our first analysis in NetworkX, we will do some basic network manipulation, then run multiple measures to find highest centrality nodes

- First, we will need to convert to an undirected network, and extract the main component

```
# Many of the centrality metrics require undirected graphs, so we will symmetrize
>>> hartford_ud=hartford.to_undirected()
# The network also has many small components, but for
# this analysis we are interested in the largest
>>> hartford_mc=hartford_main=nx.connected_component_subgraphs(hartford_ud)[0]
```

Next, we will calculate multiple measures

```
# Betweenness centrality
>>> bet_cen=nx.betweenness_centrality(hartford_mc)
# Closeness centrality
>>> clo_cen=nx.closeness_centrality(hartford_mc)
# Eigenvector centrality
>>> eig_cen=nx.eigenvector_centrality(hartford_mc)
```

Running multiple measures

For our first analysis in NetworkX, we will do some basic network manipulation, then run multiple measures to find highest centrality nodes

- First, we will need to convert to an undirected network, and extract the main component

```
# Many of the centrality metrics require undirected graphs, so we will symmetrize
>>> hartford_ud=hartford.to_undirected()
# The network also has many small components, but for
# this analysis we are interested in the largest
>>> hartford_mc=hartford_main=nx.connected_component_subgraphs(hartford_ud)[0]
```

Next, we will calculate multiple measures

```
# Betweenness centrality
>>> bet_cen=nx.betweenness_centrality(hartford_mc)
# Closeness centrality
>>> clo_cen=nx.closeness_centrality(hartford_mc)
# Eigenvector centrality
>>> eig_cen=nx.eigenvector_centrality(hartford_mc)
```

Running multiple measures

For our first analysis in `NetworkX`, we will do some basic network manipulation, then run multiple measures to find highest centrality nodes

- First, we will need to convert to an undirected network, and extract the main component

```
# Many of the centrality metrics require undirected graphs, so we will symmetrize
>>> hartford_ud=hartford.to_undirected()
# The network also has many small components, but for
# this analysis we are interested in the largest
>>> hartford_mc=hartford_main=nx.connected_component_subgraphs(hartford_ud)[0]
```

Next, we will calculate multiple measures

```
# Betweenness centrality
>>> bet_cen=nx.betweenness_centrality(hartford_mc)
# Closeness centrality
>>> clo_cen=nx.closeness_centrality(hartford_mc)
# Eigenvector centrality
>>> eig_cen=nx.eigenvector_centrality(hartford_mc)
```

Finding most central actors

To find the most central actors we will use Python's list comprehension technique to do basic data manipulation on our centrality dictionaries

```
def highest_centrality(cent_dict):
    """Returns node key with largest value from
    NX centrality dict"""
    # Create ordered tuple of centrality data
    cent_items=cent_dict.items()
    # List comprehension!
    cent_items=[(b,a) for (a,b) in cent_items]
    # Sort in descending order
    cent_items.sort()
    cent_items.reverse()
    return cent_items[0][1]
```

Now, just ask for the answer

Finding Most central actors

```
>>>  print("Actor "+str(highest_centrality(bet_cen))+" has the highest Betweenness centrality")
Actor 82 has the highest Betweenness centrality
```

Finding most central actors

To find the most central actors we will use Python's list comprehension technique to do basic data manipulation on our centrality dictionaries

```
def highest_centrality(cent_dict):
    """Returns node key with largest value from
    NX centrality dict"""
    # Create ordered tuple of centrality data
    cent_items=cent_dict.items()
    # List comprehension!
    cent_items=[(b,a) for (a,b) in cent_items]
    # Sort in descending order
    cent_items.sort()
    cent_items.reverse()
    return cent_items[0][1]
```

Now, just ask for the answer

Finding Most central actors

```
>>> print("Actor "+str(highest_centrality(bet_cen))+" has the highest Betweenness centrality")
Actor 82 has the highest Betweenness centrality
```

Finding most central actors

To find the most central actors we will use Python's list comprehension technique to do basic data manipulation on our centrality dictionaries

```
def highest_centrality(cent_dict):
    """Returns node key with largest value from
    NX centrality dict"""
    # Create ordered tuple of centrality data
    cent_items=cent_dict.items()
    # List comprehension!
    cent_items=[(b,a) for (a,b) in cent_items]
    # Sort in descending order
    cent_items.sort()
    cent_items.reverse()
    return cent_items[0][1]
```

List comprehension

- Given a dict: `d={1: 0.15, 2: 0.67}`
- `d.items()` → `[(1,0.15),(2,0.67)]`
- `d=[(b,a) for (a,b in d)]` → `[(0.15,1),(0.67,2)]`

Now, just ask for the answer

Finding Most central actors

```
>>>  print("Actor "+str(highest_centrality(bet_cen))+" has the highest Betweenness centrality")
Actor 82 has the highest Betweenness centrality
```

Finding most central actors

To find the most central actors we will use Python's list comprehension technique to do basic data manipulation on our centrality dictionaries

```
def highest_centrality(cent_dict):
    """Returns node key with largest value from
    NX centrality dict"""
    # Create ordered tuple of centrality data
    cent_items=cent_dict.items()
    # List comprehension!
    cent_items=[(b,a) for (a,b) in cent_items]
    # Sort in descending order
    cent_items.sort()
    cent_items.reverse()
    return cent_items[0][1]
```

List comprehension

- Given a dict: `d={1: 0.15, 2: 0.67}`
- `d.items()` → `[(1,0.15),(2,0.67)]`
- `d=[(b,a) for (a,b in d)]` → `[(0.15,1),(0.67,2)]`

Here, we use list comprehension in order to use Python's built-in sort and reverse list functions

Now, just ask for the answer

Finding Most central actors

```
>>>  print("Actor "+str(highest_centrality(bet_cen))+" has the highest Betweenness centrality")
Actor 82 has the highest Betweenness centrality
```

Finding most central actors

To find the most central actors we will use Python's list comprehension technique to do basic data manipulation on our centrality dictionaries

```
def highest_centrality(cent_dict):
    """Returns node key with largest value from
    NX centrality dict"""
    # Create ordered tuple of centrality data
    cent_items=cent_dict.items()
    # List comprehension!
    cent_items=[(b,a) for (a,b) in cent_items]
    # Sort in descending order
    cent_items.sort()
    cent_items.reverse()
    return cent_items[0][1]
```

List comprehension

- Given a dict: `d={1: 0.15, 2: 0.67}`
- `d.items()` → `[(1,0.15),(2,0.67)]`
- `d=[(b,a) for (a,b in d)]` → `[(0.15,1),(0.67,2)]`

Here, we use list comprehension in order to use Python's built-in `sort` and `reverse` list functions

Now, just ask for the answer

Finding Most central actors

```
>>>  print("Actor "+str(highest_centrality(bet_cen))+" has the highest Betweenness centrality")
Actor 82 has the highest Betweenness centrality
```

Calculating degree distribution

One of the most popular network level statistical description of a network is its degree distribution

- In NetworkX this is a simply one-line operation

Get list of degree rank frequency

```
# Create a Barabasi-Albert network
>>> ba_net=barabasi_albert_graph(1000,2)
# 6.1 Built-in function for degree distribution
>>> dh=degree_histogram(ba_net)
```

Calculating degree distribution

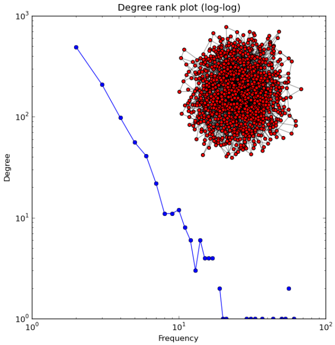
One of the most popular network level statistical description of a network is its degree distribution

- In NetworkX this is a simply one-line operation

Get list of degree rank frequency

```
# Create a Barabasi-Albert network
>>> ba_net=barabasi_albert_graph(1000,2)
# 6.1 Built-in function for degree distribution
>>> dh=degree_histogram(ba_net)
```

- As we will see next, we can use matplotlib to take this data and create publication ready plots
- Ex. from http://networkx.lanl.gov/examples/drawing/degree_histogram.html



Calculating basic community structure

Often in network analysis we are interested in estimating the cohesiveness of a network, or the communities that exists within the structure

Calculating basic community structure

Often in network analysis we are interested in estimating the cohesiveness of a network, or the communities that exists within the structure

Cliques

- Maximal cliques are the largest complete subgraph containing a given point. There are several algorithms for finding cliques, including Bron Kerbosch (1973), Tomita, Tanaka and Takahashi (2006), Cazals and Karande (2008)

Often in network analysis we are interested in estimating the cohesiveness of a network, or the communities that exists within the structure

Cliques

- Maximal cliques are the largest complete subgraph containing a given point. There are several algorithms for finding cliques, including Bron Kerbosch (1973), Tomita, Tanaka and Takahashi (2006), Cazals and Karande (2008)

Clustering

- For each node find the fraction of possible triangles that exist, $c_v = \frac{2T(v)}{deg(v)(deg(v)-1)}$, where $T(v)$ is the number of triangles through node v.

Calculating basic community structure

Often in network analysis we are interested in estimating the cohesiveness of a network, or the communities that exists within the structure

Cliques

- Maximal cliques are the largest complete subgraph containing a given point. There are several algorithms for finding cliques, including Bron Kerbosch (1973), Tomita, Tanaka and Takahashi (2006), Cazals and Karande (2008)

Clustering

- For each node find the fraction of possible triangles that exist,$c_v = \frac{2T(v)}{deg(v)(deg(v)-1)}$, where $T(v)$ is the number of triangles through node v.

Transitivity

- The fraction of all possible triangles which are in fact triangles. Or, $Trans = 3\left(\frac{T}{t}\right)$, where $T = \#$ of possible triangles and $t = \#$ of actual triads

Calculating basic community structure

Often in network analysis we are interested in estimating the cohesiveness of a network, or the communities that exists within the structure

Cliques

- Maximal cliques are the largest complete subgraph containing a given point. There are several algorithms for finding cliques, including Bron Kerbosch (1973), Tomita, Tanaka and Takahashi (2006), Cazals and Karande (2008)

Clustering

- For each node find the fraction of possible triangles that exist, $c_v = \frac{2T(v)}{deg(v)(deg(v)-1)}$, where $T(v)$ is the number of triangles through node v.

Transitivity

- The fraction of all possible triangles which are in fact triangles. Or, $Trans = 3\left(\frac{T}{t}\right)$, where $T = \#$ of possible triangles and $t = \#$ of actual triads

We will use clustering coefficients to identify community structure in the Hartford drug network

Toy community detection example (not a good one)

Calculating clustering coefficients

```
# Calculate clustering coefficients of each node (return as dict)
clus=clustering(hartford_mc,with_labels=True)
# Get counts of nodes membership for each clustering coefficient, and clean up
unique_clus=list(unique(clus.values()))
clus_counts=zip(map(lambda c: clus.values().count(c),unique_clus),unique_clus)
clus_counts.sort()
clus_counts.reverse()
# Create a subgraph from nodes with most frequent clustering coefficient
mode_clus_sg=subgraph(hartford_mc,[(a) for (a,b) in clus.items() if b==clus_counts[0][1]])
```

Toy community detection example (not a good one)

Calculating clustering coefficients

```
# Calculate clustering coefficients of each node (return as dict)
clus=clustering(hartford_mc,with_labels=True)
# Get counts of nodes membership for each clustering coefficient, and clean up
unique_clus=list(unique(clus.values()))
clus_counts=zip(map(lambda c: clus.values().count(c),unique_clus),unique_clus)
clus_counts.sort()
clus_counts.reverse()
# Create a subgraph from nodes with most frequent clustering coefficient
mode_clus_sg=subgraph(hartford_mc,[(a) for (a,b) in clus.items() if b==clus_counts[0][1]])
```

- Use the `with_labels` to return a `dict` keyed by node label

Toy community detection example (not a good one)

Calculating clustering coefficients

```
# Calculate clustering coefficients of each node (return as dict)
clus=clustering(hartford_mc,with_labels=True)
# Get counts of nodes membership for each clustering coefficient, and clean up
unique_clus=list(unique(clus.values()))
clus_counts=zip(map(lambda c: clus.values().count(c),unique_clus),unique_clus)
clus_counts.sort()
clus_counts.reverse()
# Create a subgraph from nodes with most frequent clustering coefficient
mode_clus_sg=subgraph(hartford_mc,[(a) for (a,b) in clus.items() if b==clus_counts[0][1]])
```

- Use the `with_labels` to return a `dict` keyed by node label
- The `zip` function takes two `lists` and returns a `tuple`

Toy community detection example (not a good one)

Calculating clustering coefficients

```
# Calculate clustering coefficients of each node (return as dict)
clus=clustering(hartford_mc,with_labels=True)
# Get counts of nodes membership for each clustering coefficient, and clean up
unique_clus=list(unique(clus.values()))
clus_counts=zip(map(lambda c: clus.values().count(c),unique_clus),unique_clus)
clus_counts.sort()
clus_counts.reverse()
# Create a subgraph from nodes with most frequent clustering coefficient
mode_clus_sg=subgraph(hartford_mc,[(a) for (a,b) in clus.items() if b==clus_counts[0][1]])
```

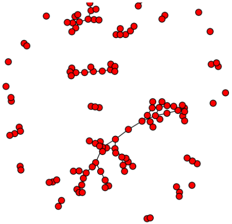
- Use the `with_labels` to return a `dict` keyed by node label
- The `zip` function takes two `lists` and returns a `tuple`
- More complex list comprehension with logic operator

Toy community detection example (not a good one)

Calculating clustering coefficients

```
# Calculate clustering coefficients of each node (return as dict)
clus=clustering(hartford_mc,with_labels=True)
# Get counts of nodes membership for each clustering coefficient, and clean up
unique_clus=list(unique(clus.values()))
clus_counts=zip(map(lambda c: clus.values().count(c),unique_clus),unique_clus)
clus_counts.sort()
clus_counts.reverse()
# Create a subgraph from nodes with most frequent clustering coefficient
mode_clus_sg=subgraph(hartford_mc,[(a) for (a,b) in clus.items() if b==clus_counts[0][1]])
```

- Use the `with_labels` to return a `dict` keyed by node label
- The `zip` function takes two `lists` and returns a `tuple`
- More complex list comprehension with logic operator

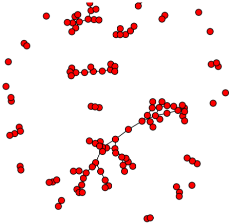


Toy community detection example (not a good one)

Calculating clustering coefficients

```
# Calculate clustering coefficients of each node (return as dict)
clus=clustering(hartford_mc,with_labels=True)
# Get counts of nodes membership for each clustering coefficient, and clean up
unique_clus=list(unique(clus.values()))
clus_counts=zip(map(lambda c: clus.values().count(c),unique_clus),unique_clus)
clus_counts.sort()
clus_counts.reverse()
# Create a subgraph from nodes with most frequent clustering coefficient
mode_clus_sg=subgraph(hartford_mc,[(a) for (a,b) in clus.items() if b==clus_counts[0][1]])
```

- Use the `with_labels` to return a `dict` keyed by node label
- The `zip` function takes two `lists` and returns a `tuple`
- More complex list comprehension with logic operator



Later, we'll learn how to create a network visualization like the one above

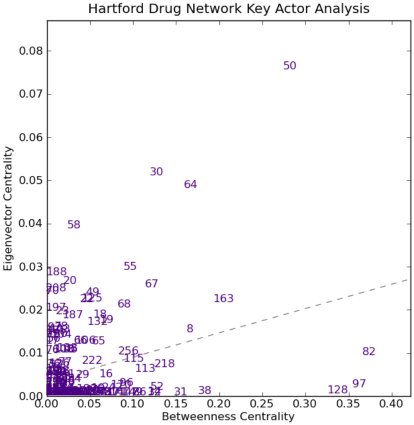
Introduction to `matplotlib`

Recall Python's scientific computing trinity: `NumPy`, `SciPy` and `matplotlib`

- While `NumPy` and `SciPy` do most of the behind the scenes work, you will interact with `matplotlib` frequently for when doing network analysis

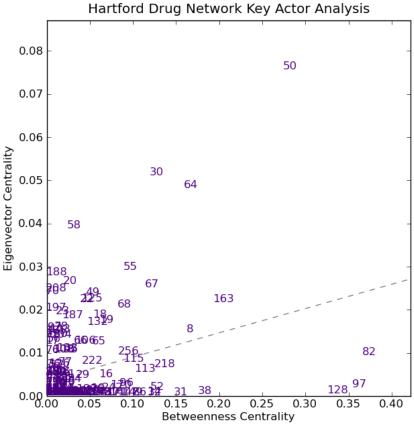
Recall Python's scientific computing trinity: `NumPy`, `SciPy` and `matplotlib`

- While `NumPy` and `SciPy` do most of the behind the scenes work, you will interact with `matplotlib` frequently for when doing network analysis



Recall Python's scientific computing trinity: `NumPy`, `SciPy` and `matplotlib`

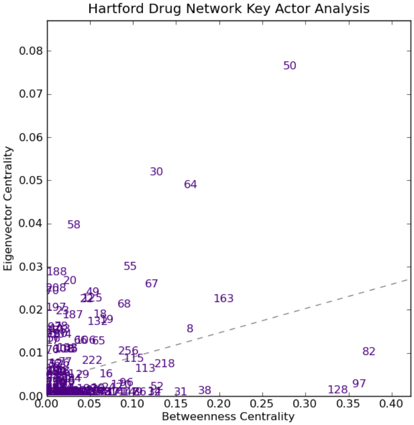
- While `NumPy` and `SciPy` do most of the behind the scenes work, you will interact with `matplotlib` frequently for when doing network analysis



We will need to create a function that takes two centrality `dict` and generates this plot

Recall Python's scientific computing trinity: `NumPy`, `SciPy` and `matplotlib`

- While `NumPy` and `SciPy` do most of the behind the scenes work, you will interact with `matplotlib` frequently for when doing network analysis

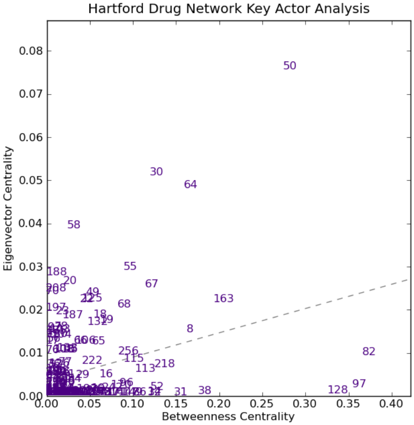


We will need to create a function that takes two centrality `dict` and generates this plot

1. Create a `matplotlib` figure

Recall Python's scientific computing trinity: `NumPy`, `SciPy` and `matplotlib`

- While `NumPy` and `SciPy` do most of the behind the scenes work, you will interact with `matplotlib` frequently for when doing network analysis

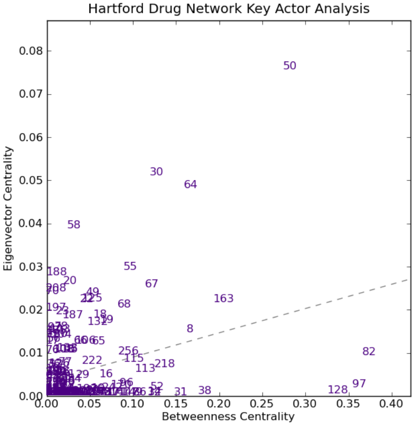


We will need to create a function that takes two centrality `dict` and generates this plot

1. Create a `matplotlib` figure
2. Plot each node label as a point

Recall Python's scientific computing trinity: `NumPy`, `SciPy` and `matplotlib`

- While `NumPy` and `SciPy` do most of the behind the scenes work, you will interact with `matplotlib` frequently for when doing network analysis

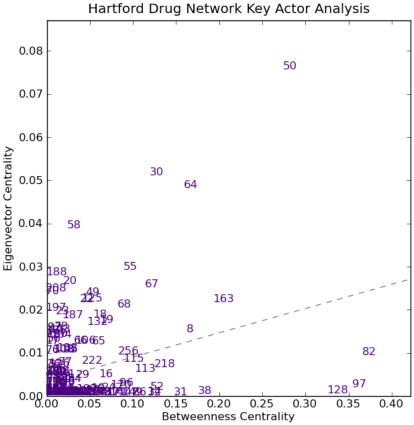


We will need to create a function that takes two centrality `dict` and generates this plot

1. Create a `matplotlib` figure
2. Plot each node label as a point
3. Add a "best fit" line

Recall Python's scientific computing trinity: NumPy, SciPy and matplotlib

- While NumPy and SciPy do most of the behind the scenes work, you will interact with matplotlib frequently for when doing network analysis

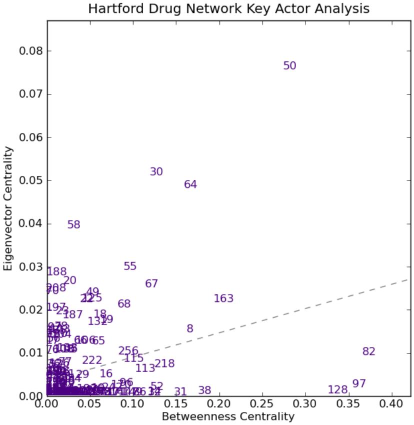


We will need to create a function that takes two centrality dict and generates this plot

1. Create a matplotlib figure
2. Plot each node label as a point
3. Add a "best fit" line
4. Add axis and title labels

Recall Python's scientific computing trinity: `NumPy`, `SciPy` and `matplotlib`

- While `NumPy` and `SciPy` do most of the behind the scenes work, you will interact with `matplotlib` frequently for when doing network analysis



We will need to create a function that takes two centrality `dict` and generates this plot

1. Create a `matplotlib` figure
2. Plot each node label as a point
3. Add a "best fit" line
4. Add axis and title labels
5. Save figure as a PNG file

Creating a key actor plot in matplotlib

The centrality_scatter function, part one

```
def centrality_scatter(met_dict1,met_dict2,path="",ylab="",xlab="",title="",reg=False):
    # Create figure and drawing axis
    fig=P.figure(figsize=(7,7))
    ax1=fig.add_subplot(111)
    # Create items so actors can be sorted properly
    met_items1=met_dict1.items()
    met_items2=met_dict2.items()
    met_items1.sort()
    met_items2.sort()
    # Grab data
    xdata=[(b) for (a,b) in met_items1]
    ydata=[(b) for (a,b) in met_items2]
    # Add each actor to the plot by ID
    for p in xrange(len(met_items1)):
        ax1.text(x=xdata[p],y=ydata[p],s=str(met_items1[p][0]),color="indigo")
```

Creating a key actor plot in `matplotlib`

The `centrality_scatter` function, part one

```
def centrality_scatter(met_dict1,met_dict2,path="",ylab="",xlab="",title="",reg=False):
    # Create figure and drawing axis
    fig=P.figure(figsize=(7,7))
    ax1=fig.add_subplot(111)
    # Create items so actors can be sorted properly
    met_items1=met_dict1.items()
    met_items2=met_dict2.items()
    met_items1.sort()
    met_items2.sort()
    # Grab data
    xdata=[(b) for (a,b) in met_items1]
    ydata=[(b) for (a,b) in met_items2]
    # Add each actor to the plot by ID
    for p in xrange(len(met_items1)):
        ax1.text(x=xdata[p],y=ydata[p],s=str(met_items1[p][0]),color="indigo")
```

- Create a canvas to draw on

Creating a key actor plot in matplotlib

The centrality_scatter function, part one

```
def centrality_scatter(met_dict1,met_dict2,path="",ylab="",xlab="",title="",reg=False):
    # Create figure and drawing axis
    fig=P.figure(figsize=(7,7))
    ax1=fig.add_subplot(111)
    # Create items so actors can be sorted properly
    met_items1=met_dict1.items()
    met_items2=met_dict2.items()
    met_items1.sort()
    met_items2.sort()
    # Grab data
    xdata=[(b) for (a,b) in met_items1]
    ydata=[(b) for (a,b) in met_items2]
    # Add each actor to the plot by ID
    for p in xrange(len(met_items1)):
        ax1.text(x=xdata[p],y=ydata[p],s=str(met_items1[p][0]),color="indigo")
```

- Create a canvas to draw on
- manipulate and store centrality data

Creating a key actor plot in matplotlib

The centrality_scatter function, part one

```
def centrality_scatter(met_dict1,met_dict2,path="",ylab="",xlab="",title="",reg=False):
    # Create figure and drawing axis
    fig=P.figure(figsize=(7,7))
    ax1=fig.add_subplot(111)
    # Create items so actors can be sorted properly
    met_items1=met_dict1.items()
    met_items2=met_dict2.items()
    met_items1.sort()
    met_items2.sort()
    # Grab data
    xdata=[(b) for (a,b) in met_items1]
    ydata=[(b) for (a,b) in met_items2]
    # Add each actor to the plot by ID
    for p in xrange(len(met_items1)):
        ax1.text(x=xdata[p],y=ydata[p],s=str(met_items1[p][0]),color="indigo")
```

- Create a canvas to draw on
- manipulate and store centrality data
- Add points to plot as node labels

Creating a key actor plot in matplotlib

The centrality_scatter function, part one

```
def centrality_scatter(met_dict1,met_dict2,path="",ylab="",xlab="",title="",reg=False):
...
    # If adding a best fit line, we will use NumPy to calculate the points.
    if reg:
        # Function returns y-intercept and slope.  So, we create a function to
        # draw LOBF from this data
        slope,yint=polyfit(xdata,ydata,1)
        xline=P.xticks()[0]
        yline=map(lambda x: slope*x+yint,xline)
        # Add line
        ax1.plot(xline,yline,ls='--',color='grey')
    # Set new x- and y-axis limits to data
    P.xlim((0.0,max(xdata)+(.15*max(xdata))))   # Give a little buffer
    P.ylim((0.0,max(ydata)+(.15*max(ydata))))
    # Add labels
    ax1.set_title(title)
    ax1.set_xlabel(xlab)
    ax1.set_ylabel(ylab)
    # Save figure
    P.savefig(path,dpi=100)
```

Creating a key actor plot in matplotlib

The centrality_scatter function, part one

```
def centrality_scatter(met_dict1,met_dict2,path="",ylab="",xlab="",title="",reg=False):
...
    # If adding a best fit line, we will use NumPy to calculate the points.
    if reg:
        # Function returns y-intercept and slope.  So, we create a function to
        # draw LOBF from this data
        slope,yint=polyfit(xdata,ydata,1)
        xline=P.xticks()[0]
        yline=map(lambda x: slope*x+yint,xline)
        # Add line
        ax1.plot(xline,yline,ls='--',color='grey')
    # Set new x- and y-axis limits to data
    P.xlim((0.0,max(xdata)+(.15*max(xdata))))   # Give a little buffer
    P.ylim((0.0,max(ydata)+(.15*max(ydata))))
    # Add labels
    ax1.set_title(title)
    ax1.set_xlabel(xlab)
    ax1.set_ylabel(ylab)
    # Save figure
    P.savefig(path,dpi=100)
```

- Add a best fit line

Creating a key actor plot in matplotlib

The centrality_scatter function, part one

```
def centrality_scatter(met_dict1,met_dict2,path="",ylab="",xlab="",title="",reg=False):
...
    # If adding a best fit line, we will use NumPy to calculate the points.
    if reg:
        # Function returns y-intercept and slope.  So, we create a function to
        # draw LOBF from this data
        slope,yint=polyfit(xdata,ydata,1)
        xline=P.xticks()[0]
        yline=map(lambda x: slope*x+yint,xline)
        # Add line
        ax1.plot(xline,yline,ls='--',color='grey')
    # Set new x- and y-axis limits to data
    P.xlim((0.0,max(xdata)+(.15*max(xdata))))   # Give a little buffer
    P.ylim((0.0,max(ydata)+(.15*max(ydata))))
    # Add labels
    ax1.set_title(title)
    ax1.set_xlabel(xlab)
    ax1.set_ylabel(ylab)
    # Save figure
    P.savefig(path,dpi=100)
```

- Add a best fit line
- Resize figure to fit data

Creating a key actor plot in matplotlib

The centrality_scatter function, part one

```
def centrality_scatter(met_dict1,met_dict2,path="",ylab="",xlab="",title="",reg=False):
...
    # If adding a best fit line, we will use NumPy to calculate the points.
    if reg:
        # Function returns y-intercept and slope.  So, we create a function to
        # draw LOBF from this data
        slope,yint=polyfit(xdata,ydata,1)
        xline=P.xticks()[0]
        yline=map(lambda x: slope*x+yint,xline)
        # Add line
        ax1.plot(xline,yline,ls='--',color='grey')
    # Set new x- and y-axis limits to data
    P.xlim((0.0,max(xdata)+(.15*max(xdata))))   # Give a little buffer
    P.ylim((0.0,max(ydata)+(.15*max(ydata))))
    # Add labels
    ax1.set_title(title)
    ax1.set_xlabel(xlab)
    ax1.set_ylabel(ylab)
    # Save figure
    P.savefig(path,dpi=100)
```

- Add a best fit line
- Resize figure to fit data
- Add labels, and save the figure as a PNG file

As powerful as `NetworkX` and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

Exporting network data and analytics

As powerful as `NetworkX` and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

- Suite of tools lacks your specific need

Exporting network data and analytics

As powerful as `NetworkX` and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

- Suite of tools lacks your specific need
- Require alternate visualization

Exporting network data and analytics

As powerful as `NetworkX` and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

- Suite of tools lacks your specific need
- Require alternate visualization
- Storage for later analysis

Exporting network data and analytics

As powerful as `NetworkX` and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

- Suite of tools lacks your specific need
- Require alternate visualization
- Storage for later analysis

In most cases this will entail either exporting the raw network data, or metrics from some network analysis

Exporting network data and analytics

As powerful as `NetworkX` and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

- Suite of tools lacks your specific need
- Require alternate visualization
- Storage for later analysis

In most cases this will entail either exporting the raw network data, or metrics from some network analysis

1. `NetworkX` can write out network data in as many formats as it can read them, and the process is equally straightforward

Exporting network data and analytics

As powerful as `NetworkX` and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

- Suite of tools lacks your specific need
- Require alternate visualization
- Storage for later analysis

In most cases this will entail either exporting the raw network data, or metrics from some network analysis

1. `NetworkX` can write out network data in as many formats as it can read them, and the process is equally straightforward
2. When you want to export metrics we can use Python's built-in `XML` and `CSV` libraries

Exporting network data and analytics

As powerful as `NetworkX` and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

- Suite of tools lacks your specific need
- Require alternate visualization
- Storage for later analysis

In most cases this will entail either exporting the raw network data, or metrics from some network analysis

1. `NetworkX` can write out network data in as many formats as it can read them, and the process is equally straightforward
2. When you want to export metrics we can use Python's built-in `XML` and `CSV` libraries
3. Depending on your needs you may prefer one, the other or both

Exporting network data and analytics

As powerful as `NetworkX` and the complementing scientific computing packages in Python are, it may often be useful or necessary to output your data for additional analysis

- Suite of tools lacks your specific need
- Require alternate visualization
- Storage for later analysis

In most cases this will entail either exporting the raw network data, or metrics from some network analysis

1. `NetworkX` can write out network data in as many formats as it can read them, and the process is equally straightforward
2. When you want to export metrics we can use Python's built-in `XML` and `CSV` libraries
3. Depending on your needs you may prefer one, the other or both

Next, we will review how to save data in different formats and export metrics to a `CSV` file using the Hartford drug net data

Saving network data in different formats

The syntax for exporting network data follows exactly the syntax for loading it

NX syntax for writing a network file

>>>	nx.write_format(G,	"path/to/file.txt",	*...options...*)
	↑	↑	↑
	NX function, net variable	File to be written	Nodes/edge data, etc.

Saving network data in different formats

The syntax for exporting network data follows exactly the syntax for loading it

NX syntax for writing a network file

>>>	nx.write_format(G,	"path/to/file.txt",	*...options...*)
	↑	↑	↑
	NX function, net variable	File to be written	Nodes/edge data, etc.

Saving network data in different formats

The syntax for exporting network data follows exactly the syntax for loading it

NX syntax for writing a network file

>>>	nx.write_format(G,	"path/to/file.txt",	*...options...*)
	↑	↑	↑
	NX function, net variable	File to be written	Nodes/edge data, etc.

Saving network data in different formats

The syntax for exporting network data follows exactly the syntax for loading it

NX syntax for writing a network file

>>> nx.write_format(G, "path/to/file.txt", *...options...*)

$\uparrow$ NX function, net variable

$\uparrow$ File to be written

$\uparrow$ Nodes/edge data, etc.

Saving network data in different formats

The syntax for exporting network data follows exactly the syntax for loading it

NX syntax for writing a network file

$>>>$ nx.write_format(G, "path/to/file.txt", *...options...*)

- nx.write_format(G, ↑ NX function, net variable
- "path/to/file.txt", ↑ File to be written
- *...options...*) ↑ Nodes/edge data, etc.

Let's try!

- Output the Hartford drug net data as an adjacency list
- Add metric data to each node of the network
- Output new network in Pajek format with node attributes

Saving network data and adding node attributes

As shown, this is a simple one line operation

Output Hartford drug net data as an adjacency list

```
nx.write_adjlist(hartford_mc,"../../data/hartford_mc_adj.txt")
```

Next, we will add the Eigenvector centrality of each node to the graph object

Adding node attributes

```
def add_metric(G,met_dict):
    """Adds metric data to G from a dictionary keyed by node labels"""
    if(G.nodes().sort()==met_dict.keys().sort()):
        for i in met_dict.keys():
            G.add_node(i,metric=met_dict[i])
        return G
    else:
        raise ValueError("Node labels do not match")
```

Saving network data and adding node attributes

As shown, this is a simple one line operation

Output Hartford drug net data as an adjacency list

```
nx.write_adjlist(hartford_mc,"../../data/hartford_mc_adj.txt")
```

Next, we will add the Eigenvector centrality of each node to the graph object

Adding node attributes

```
def add_metric(G,met_dict):
    """Adds metric data to G from a dictionary keyed by node labels"""
    if(G.nodes().sort()==met_dict.keys().sort()):
        for i in met_dict.keys():
            G.add_node(i,metric=met_dict[i])
        return G
    else:
        raise ValueError("Node labels do not match")
```

- Quick error checking

Saving network data and adding node attributes

As shown, this is a simple one line operation

Output Hartford drug net data as an adjacency list

```
nx.write_adjlist(hartford_mc,"../../data/hartford_mc_adj.txt")
```

Next, we will add the Eigenvector centrality of each node to the graph object

Adding node attributes

```
def add_metric(G,met_dict):
    """Adds metric data to G from a dictionary keyed by node labels"""
    if(G.nodes().sort()==met_dict.keys().sort()):
        for i in met_dict.keys():
            G.add_node(i,metric=met_dict[i])
        return G
    else:
        raise ValueError("Node labels do not match")
```

- Quick error checking
- Add node attribute as “metric”

Using the Python CSV library

Python has powerful built-in tools for reading and writing standard data formats

- One of the most useful, and frequently used, is the CSV library and the DictWriter

```
import csv
...
def csv_exporter(data_dict,path):
    """Takes a dict of centralities keyed by column headers and exports
    data as a CSV file"""
    # Create column header list
    col_headers=["Actor"]
    col_headers.extend(data_dict.keys())
    # Create CSV writer and write column headers
    writer=csv.DictWriter(open(path,"w"),fieldnames=col_headers)
    writer.writerow(dict((h,h) for h in col_headers))
    # Write each row of data
    for j in data_dict[col_headers[1]].keys():
        # Create a new dict for each row
        row=dict.fromkeys(col_headers)
        row["Actor"]=j
        for k in data_dict.keys():
            row[k]=data_dict[k][j]
        writer.writerow(row)
```

Python has powerful built-in tools for reading and writing standard data formats

- One of the most useful, and frequently used, is the CSV library and the `DictWriter`

```
import csv
...
def csv_exporter(data_dict,path):
    """Takes a dict of centralities keyed by column headers and exports
    data as a CSV file"""
    # Create column header list
    col_headers=["Actor"]
    col_headers.extend(data_dict.keys())
    # Create CSV writer and write column headers
    writer=csv.DictWriter(open(path,"w"),fieldnames=col_headers)
    writer.writerow(dict((h,h) for h in col_headers))
    # Write each row of data
    for j in data_dict[col_headers[1]].keys():
        # Create a new dict for each row
        row=dict.fromkeys(col_headers)
        row["Actor"]=j
        for k in data_dict.keys():
            row[k]=data_dict[k][j]
        writer.writerow(row)
```

Python has powerful built-in tools for reading and writing standard data formats

- One of the most useful, and frequently used, is the CSV library and the `DictWriter`

```
import csv
...
def csv_exporter(data_dict,path):
    """Takes a dict of centralities keyed by column headers and exports
    data as a CSV file"""
    # Create column header list
    col_headers=["Actor"]
    col_headers.extend(data_dict.keys())
    # Create CSV writer and write column headers
    writer=csv.DictWriter(open(path,"w"),fieldnames=col_headers)
    writer.writerow(dict((h,h) for h in col_headers))
    # Write each row of data
    for j in data_dict[col_headers[1]].keys():
        # Create a new dict for each row
        row=dict.fromkeys(col_headers)
        row["Actor"]=j
        for k in data_dict.keys():
            row[k]=data_dict[k][j]
        writer.writerow(row)
```

Using the Python CSV library

Python has powerful built-in tools for reading and writing standard data formats

- One of the most useful, and frequently used, is the CSV library and the `DictWriter`

```
import csv
...
def csv_exporter(data_dict,path):
    """Takes a dict of centralities keyed by column headers and exports
    data as a CSV file"""
    # Create column header list
    col_headers=["Actor"]
    col_headers.extend(data_dict.keys())
    # Create CSV writer and write column headers
    writer=csv.DictWriter(open(path,"w"),fieldnames=col_headers)
    writer.writerow(dict((h,h) for h in col_headers))
    # Write each row of data
    for j in data_dict[col_headers[1]].keys():
        # Create a new dict for each row
        row=dict.fromkeys(col_headers)
        row["Actor"]=j
        for k in data_dict.keys():
            row[k]=data_dict[k][j]
        writer.writerow(row)
```

Using the Python CSV library

Python has powerful built-in tools for reading and writing standard data formats

- One of the most useful, and frequently used, is the CSV library and the DictWriter

```
import csv
...
def csv_exporter(data_dict,path):
    """Takes a dict of centralities keyed by column headers and exports
    data as a CSV file"""
    # Create column header list
    col_headers=["Actor"]
    col_headers.extend(data_dict.keys())
    # Create CSV writer and write column headers
    writer=csv.DictWriter(open(path,"w"),fieldnames=col_headers)
    writer.writerow(dict((h,h) for h in col_headers))
    # Write each row of data
    for j in data_dict[col_headers[1]].keys():
        # Create a new dict for each row
        row=dict.fromkeys(col_headers)
        row["Actor"]=j
        for k in data_dict.keys():
            row[k]=data_dict[k][j]
        writer.writerow(row)
```

Using the Python CSV library

Python has powerful built-in tools for reading and writing standard data formats

- One of the most useful, and frequently used, is the CSV library and the `DictWriter`

```
import csv
...
def csv_exporter(data_dict,path):
    """Takes a dict of centralities keyed by column headers and exports
    data as a CSV file"""
    # Create column header list
    col_headers=["Actor"]
    col_headers.extend(data_dict.keys())
    # Create CSV writer and write column headers
    writer=csv.DictWriter(open(path,"w"),fieldnames=col_headers)
    writer.writerow(dict((h,h) for h in col_headers))
    # Write each row of data
    for j in data_dict[col_headers[1]].keys():
        # Create a new dict for each row
        row=dict.fromkeys(col_headers)
        row["Actor"]=j
        for k in data_dict.keys():
            row[k]=data_dict[k][j]
        writer.writerow(row)
```

Using the Python CSV library

Python has powerful built-in tools for reading and writing standard data formats

- One of the most useful, and frequently used, is the `CSV` library and the `DictWriter`

```
import csv
...
def csv_exporter(data_dict,path):
    """Takes a dict of centralities keyed by column headers and exports
    data as a CSV file"""
    # Create column header list
    col_headers=["Actor"]
    col_headers.extend(data_dict.keys())
    # Create CSV writer and write column headers
    writer=csv.DictWriter(open(path,"w"),fieldnames=col_headers)
    writer.writerow(dict((h,h) for h in col_headers))
    # Write each row of data
    for j in data_dict[col_headers[1]].keys():
        # Create a new dict for each row
        row=dict.fromkeys(col_headers)
        row["Actor"]=j
        for k in data_dict.keys():
            row[k]=data_dict[k][j]
        writer.writerow(row)
```

The results of CSV export

We can now open the CSV file in our favorite spreadsheet program

- Perform traditional data exploration
- Load into other analytics platforms for additional analysis (e.g., R)
- Store for latter use

	A	B	C	D
1	Actor	Closeness	Betweeness	Eigenvector
2	1	0.12467532	0.0072576	0.00025176
3	2	0.12475634	0.01767427	0.00025964
4	3	0.12565445	0.05687441	0.00023185
5	4	0.10223642	0.03108639	1.44E-05
6	5	0.1443609	0	0.00313152
7	6	0.09943035	0.01041667	1.49E-07
8	7	0.11340815	0.04362093	6.78E-05
9	8	0.20512821	0.16354003	0.01471888
10	9	0.11267606	0.00741624	0.0001101
11	10	0.13983977	0.05258239	0.00095456
12	11	0.1703638	0.01250999	0.0032333
13	13	0.13892909	0	1.79E-05
14	14	0.17219731	0.11848775	0.00029737
15	15	0.13521127	0.00079897	2.11E-05
16	16	0.15907208	0.06203647	0.00432838

Development of visualization techniques and algorithms has become somewhat of a cottage industry

What makes a good network visualization technique

Development of visualization techniques and algorithms has become somewhat of a cottage industry

- Maximize "visibility" of network
- Scale up to very large graphs
- Display nodal- (centrality) of network-level (community structure) information

What makes a good network visualization technique

Development of visualization techniques and algorithms has become somewhat of a cottage industry

- Maximize "visibility" of network
- Scale up to very large graphs
- Display nodal- (centrality) of network-level (community structure) information

`NetworkX` was designed as a data manipulation and analysis tool, and therefore is not meant as a visualization platform

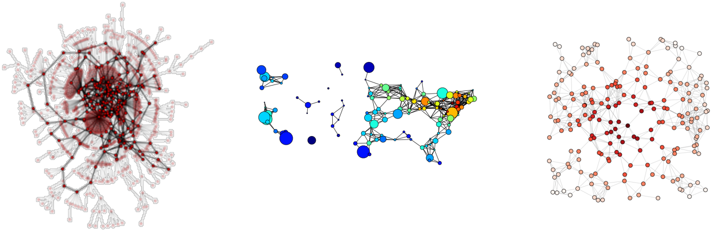
What makes a good network visualization technique

Development of visualization techniques and algorithms has become somewhat of a cottage industry

- Maximize "visibility" of network
- Scale up to very large graphs
- Display nodal- (centrality) of network-level (community structure) information

`NetworkX` was designed as a data manipulation and analysis tool, and therefore is not meant as a visualization platform

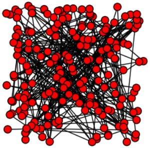
- It is, however, still capable of making very nice visualization

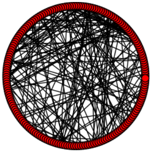


Visualization algorithms in NetworkX - Random & Circle

The most basic visualization techniques are the random and circular layouts

- The random layout places nodes in...random positions
- The circular layout places nodes in...a circle



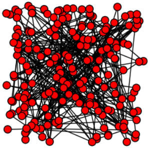


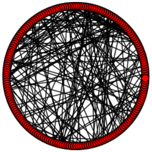
```
# Use subplots to draw random and circular layouts
# of drug net side-by-side
fig1=P.figure(figsize=(9,4))
fig1.add_subplot(121)
nx.draw_random(hartford_mc,with_labels=False,node_size=60)
fig1.add_subplot(122)
nx.draw_circular(hartford_mc,with_labels=False,node_size=60)
P.savefig("../../images/networks/rand_circ.png")
```

Visualization algorithms in NetworkX - Random & Circle

The most basic visualization techniques are the random and circular layouts

- The random layout places nodes in...random positions
- The circular layout places nodes in...a circle



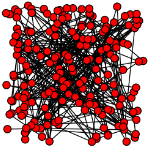


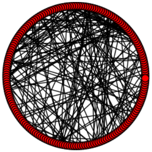
```
# Use subplots to draw random and circular layouts
# of drug net side-by-side
fig1=P.figure(figsize=(9,4))
fig1.add_subplot(121)
nx.draw_random(hartford_mc,with_labels=False,node_size=60)
fig1.add_subplot(122)
nx.draw_circular(hartford_mc,with_labels=False,node_size=60)
P.savefig("../../images/networks/rand_circ.png")
```

Visualization algorithms in NetworkX - Random & Circle

The most basic visualization techniques are the random and circular layouts

- The random layout places nodes in...random positions
- The circular layout places nodes in...a circle



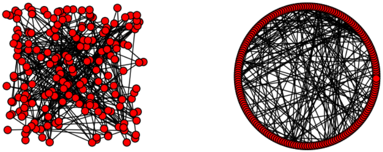


```
# Use subplots to draw random and circular layouts
# of drug net side-by-side
fig1=P.figure(figsize=(9,4))
fig1.add_subplot(121)
nx.draw_random(hartford_mc,with_labels=False,node_size=60)
fig1.add_subplot(122)
nx.draw_circular(hartford_mc,with_labels=False,node_size=60)
P.savefig("../../images/networks/rand_circ.png")
```

Visualization algorithms in NetworkX - Random & Circle

The most basic visualization techniques are the random and circular layouts

- The random layout places nodes in...random positions
- The circular layout places nodes in...a circle



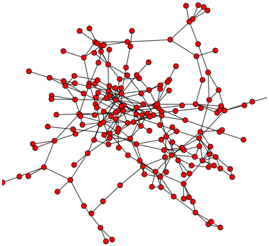
```
# Use subplots to draw random and circular layouts
# of drug net side-by-side
fig1=P.figure(figsize=(9,4))
fig1.add_subplot(121)
nx.draw_random(hartford_mc,with_labels=False,node_size=60)
fig1.add_subplot(122)
nx.draw_circular(hartford_mc,with_labels=False,node_size=60)
P.savefig("../../images/networks/rand_circ.png")
```

Visualization algorithms in NetworkX - Spring & Spectral

More commonly used visualization techniques include the spring and spectral layouts

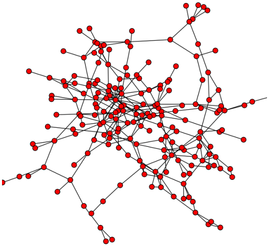
More commonly used visualization techniques include the spring and spectral layouts

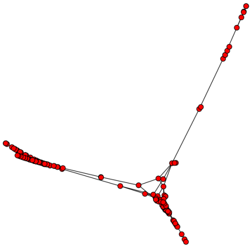
- The spring layout is a version of the Fruchterman-Reingold force-directed algorithm, which attempts to minimize overlapping edges



More commonly used visualization techniques include the spring and spectral layouts

- The spring layout is a version of the Fruchterman-Reingold force-directed algorithm, which attempts to minimize overlapping edges
- The spectral layout finds node position using the eigenvectors of the graph Laplacian, which is useful for quickly visualizing structural clustering





Visualization algorithms in NetworkX - Shell

The shell layout draws nodes as concentric circles

- Two dimensional extension of the circle layout
- We may have some reason to isolate certain nodes

25th percentile Eigenvector centrality actors

```
P.figure(figsize=(8,8))
# Find actors in 25th percentile
max_eig=max([(b) for (a,b) in eig_cen.items()])
s1=[(a) for (a,b) in eig_cen.items() if b>=.25*max_eig]
s2=hartford_mc.nodes()
# setdiff1d is a very useful NumPy function!
s2=list(setdiff1d(s2,s1))
shells=[s1,s2]
# Calculate position and draw
shell_pos=shell_layout(hartford_mc,shells)
draw_networkx(hartford_mc,shell_pos,with_labels=False,node_size=60)
P.savefig("../../images/networks/shell.png")
```

The shell layout draws nodes as concentric circles

- Two dimensional extension of the circle layout
- We may have some reason to isolate certain nodes

25th percentile Eigenvector centrality actors

```
P.figure(figsize=(8,8))
# Find actors in 25th percentile
max_eig=max([(b) for (a,b) in eig_cen.items()])
s1=[(a) for (a,b) in eig_cen.items() if b>=.25*max_eig]
s2=hartford_mc.nodes()
# setdiff1d is a very useful NumPy function!
s2=list(setdiff1d(s2,s1))
shells=[s1,s2]
# Calculate position and draw
shell_pos=shell_layout(hartford_mc,shells)
draw_networkx(hartford_mc,shell_pos,with_labels=False,node_size=60)
P.savefig("../../images/networks/shell.png")
```

The shell layout draws nodes as concentric circles

- Two dimensional extension of the circle layout
- We may have some reason to isolate certain nodes

25th percentile Eigenvector centrality actors

```
P.figure(figsize=(8,8))
# Find actors in 25th percentile
max_eig=max([(b) for (a,b) in eig_cen.items()])
s1=[(a) for (a,b) in eig_cen.items() if b>=.25*max_eig]
s2=hartford_mc.nodes()
# setdiff1d is a very useful NumPy function!
s2=list(setdiff1d(s2,s1))
shells=[s1,s2]
# Calculate position and draw
shell_pos=shell_layout(hartford_mc,shells)
draw_networkx(hartford_mc,shell_pos,with_labels=False,node_size=60)
P.savefig("../../images/networks/shell.png")
```

Visualization algorithms in NetworkX - Shell

The shell layout draws nodes as concentric circles

- Two dimensional extension of the circle layout
- We may have some reason to isolate certain nodes

25th percentile Eigenvector centrality actors

```
P.figure(figsize=(8,8))
# Find actors in 25th percentile
max_eig=max([(b) for (a,b) in eig_cen.items()])
s1=[(a) for (a,b) in eig_cen.items() if b>=.25*max_eig]
s2=hartford_mc.nodes()
# setdiff1d is a very useful NumPy function!
s2=list(setdiff1d(s2,s1))
shells=[s1,s2]
# Calculate position and draw
shell_pos=shell_layout(hartford_mc,shells)
draw_networkx(hartford_mc,shell_pos,with_labels=False,node_size=60)
P.savefig("../../images/networks/shell.png")
```

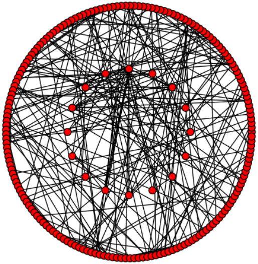
Visualization algorithms in NetworkX - Shell

The shell layout draws nodes as concentric circles

- Two dimensional extension of the circle layout
- We may have some reason to isolate certain nodes

25th percentile Eigenvector centrality actors

```
P.figure(figsize=(8,8))
# Find actors in 25th percentile
max_eig=max([(b) for (a,b) in eig_cen.items()])
s1=[(a) for (a,b) in eig_cen.items() if b>=.25*max_eig]
s2=hartford_mc.nodes()
# setdiff1d is a very useful NumPy function!
s2=list(setdiff1d(s2,s1))
shells=[s1,s2]
# Calculate position and draw
shell_pos=shell_layout(hartford_mc,shells)
draw_networkx(hartford_mc,shell_pos,with_labels=False,node_size=60)
P.savefig("../../images/networks/shell.png")
```



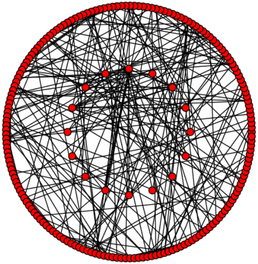
Visualization algorithms in NetworkX - Shell

The shell layout draws nodes as concentric circles

- Two dimensional extension of the circle layout
- We may have some reason to isolate certain nodes

25th percentile Eigenvector centrality actors

```
P.figure(figsize=(8,8))
# Find actors in 25th percentile
max_eig=max([(b) for (a,b) in eig_cen.items()])
s1=[(a) for (a,b) in eig_cen.items() if b>=.25*max_eig]
s2=hartford_mc.nodes()
# setdiff1d is a very useful NumPy function!
s2=list(setdiff1d(s2,s1))
shells=[s1,s2]
# Calculate position and draw
shell_pos=shell_layout(hartford_mc,shells)
draw_networkx(hartford_mc,shell_pos,with_labels=False,node_size=60)
P.savefig("../../images/networks/shell.png")
```



Beyond layout, we may also want to add analytical data to our visualization

Changing node and edge size and colors

`NetworkX` allows you to alter the size, color and shape of the nodes and edges in any visualization

Changing node and edge size and colors

`NetworkX` allows you to alter the size, color and shape of the nodes and edges in any visualization

- This can be particularly useful if we want to make some actors more prominent than others

Changing node and edge size and colors

`NetworkX` allows you to alter the size, color and shape of the nodes and edges in any visualization

- This can be particularly useful if we want to make some actors more prominent than others

In our final exercise, we will add the following analysis to the Hartford drug network

- Node size by Eigenvector centrality
- Intensity of node color by betweenness centrality
- Edge thickness by edge betweenness

The code to add analysis to visualization

More list comprehension and matplotlib colormaps

```
# Adding analysis to visualization
P.figure(figsize=(15,15))
P.subplot(111,axisbg="lightgrey")
spring_pos=nx.spring_layout(hartford_mc,iterations=1000)
# Use betweeneess centrality for node color intensity
bet_color=bet_cen.items()
bet_color.sort()
bet_color=[(b) for (a,b) in bet_color]
# Use Eigenvector centrality to set node size
eig_size=eig_cen.items()
eig_size.sort()
eig_size=[((b)*2000)+20 for (a,b) in eig_size]
# Use matplotlib's colormap for node intensity
draw_networkx(hartford_mc,spring_pos,node_color=bet_color,...
    ...cmap=P.cm.Greens,node_size=eig_size,with_labels=False)
P.savefig("../../images/networks/analysis.png")
```

The code to add analysis to visualization

More list comprehension and matplotlib colormaps

```
# Adding analysis to visualization
P.figure(figsize=(15,15))
P.subplot(111,axisbg="lightgrey")
spring_pos=nx.spring_layout(hartford_mc,iterations=1000)
# Use betweeneess centrality for node color intensity
bet_color=bet_cen.items()
bet_color.sort()
bet_color=[(b) for (a,b) in bet_color]
# Use Eigenvector centrality to set node size
eig_size=eig_cen.items()
eig_size.sort()
eig_size=[((b)*2000)+20 for (a,b) in eig_size]
# Use matplotlib's colormap for node intensity
draw_networkx(hartford_mc,spring_pos,node_color=bet_color,...
    ...cmap=P.cm.Greens,node_size=eig_size,with_labels=False)
P.savefig("../../images/networks/analysis.png")
```

The code to add analysis to visualization

More list comprehension and matplotlib colormaps

```
# Adding analysis to visualization
P.figure(figsize=(15,15))
P.subplot(111,axisbg="lightgrey")
spring_pos=nx.spring_layout(hartford_mc,iterations=1000)
# Use betweeneess centrality for node color intensity
bet_color=bet_cen.items()
bet_color.sort()
bet_color=[(b) for (a,b) in bet_color]
# Use Eigenvector centrality to set node size
eig_size=eig_cen.items()
eig_size.sort()
eig_size=[((b)*2000)+20 for (a,b) in eig_size]
# Use matplotlib's colormap for node intensity
draw_networkx(hartford_mc,spring_pos,node_color=bet_color,...
    ...cmap=P.cm.Greens,node_size=eig_size,with_labels=False)
P.savefig("../../images/networks/analysis.png")
```

The code to add analysis to visualization

More list comprehension and matplotlib colormaps

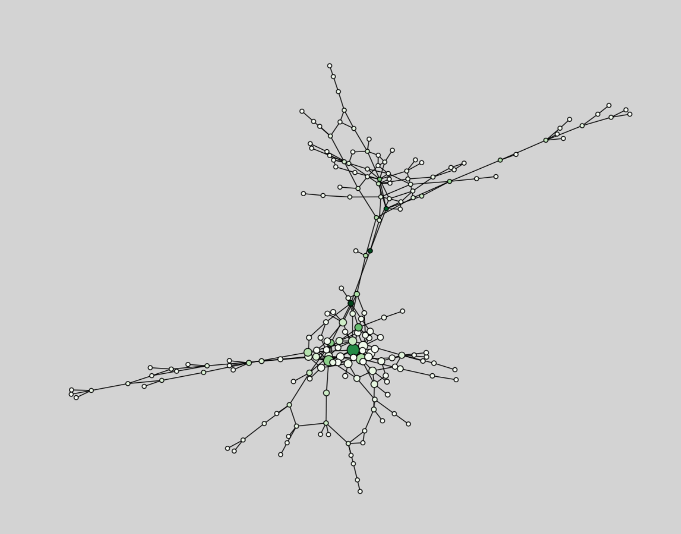
```
# Adding analysis to visualization
P.figure(figsize=(15,15))
P.subplot(111,axisbg="lightgrey")
spring_pos=nx.spring_layout(hartford_mc,iterations=1000)
# Use betweeneess centrality for node color intensity
bet_color=bet_cen.items()
bet_color.sort()
bet_color=[(b) for (a,b) in bet_color]
# Use Eigenvector centrality to set node size
eig_size=eig_cen.items()
eig_size.sort()
eig_size=[((b)*2000)+20 for (a,b) in eig_size]
# Use matplotlib's colormap for node intensity
draw_networkx(hartford_mc,spring_pos,node_color=bet_color,...
    ...cmap=P.cm.Greens,node_size=eig_size,with_labels=False)
P.savefig("../../images/networks/analysis.png")
```

The code to add analysis to visualization

More list comprehension and `matplotlib` colormaps

```
# Adding analysis to visualization
P.figure(figsize=(15,15))
P.subplot(111,axisbg="lightgrey")
spring_pos=nx.spring_layout(hartford_mc,iterations=1000)
# Use betweeneess centrality for node color intensity
bet_color=bet_cen.items()
bet_color.sort()
bet_color=[(b) for (a,b) in bet_color]
# Use Eigenvector centrality to set node size
eig_size=eig_cen.items()
eig_size.sort()
eig_size=[((b)*2000)+20 for (a,b) in eig_size]
# Use matplotlib's colormap for node intensity
draw_networkx(hartford_mc,spring_pos,node_color=bet_color,...
    ...cmap=P.cm.Greens,node_size=eig_size,with_labels=False)
P.savefig("../../images/networks/analysis.png")
```

Final visualization



Basic Analysis

Module IV Review

Basic Analysis

- How to load local data, and an example of building networks from data streamed directly from the Internet

Module IV Review

Basic Analysis

- How to load local data, and an example of building networks from data streamed directly from the Internet
- A brief review of the Python `dict` data type

Module IV Review

Basic Analysis

- How to load local data, and an example of building networks from data streamed directly from the Internet
- A brief review of the Python `dict` data type
- Calculating basic metrics, how they are stored in `NetworkX` and how to manipulate them (list comps!)

Module IV Review

Basic Analysis

- How to load local data, and an example of building networks from data streamed directly from the Internet
- A brief review of the Python `dict` data type
- Calculating basic metrics, how they are stored in `NetworkX` and how to manipulate them (list comps!)
- How to use `matplotlib` to visualize our analysis

Module IV Review

Basic Analysis

- How to load local data, and an example of building networks from data streamed directly from the Internet
- A brief review of the Python `dict` data type
- Calculating basic metrics, how they are stored in `NetworkX` and how to manipulate them (list comps!)
- How to use `matplotlib` to visualize our analysis
- Getting data out of `NetworkX` both as raw network data or analytics using the `CSV` library

Module IV Review

Basic Analysis

- How to load local data, and an example of building networks from data streamed directly from the Internet
- A brief review of the Python `dict` data type
- Calculating basic metrics, how they are stored in `NetworkX` and how to manipulate them (list comps!)
- How to use `matplotlib` to visualize our analysis
- Getting data out of `NetworkX` both as raw network data or analytics using the `CSV` library
- Network visualization techniques in `NetworkX` and how to add network analysis to a visualization

Basic Analysis

- How to load local data, and an example of building networks from data streamed directly from the Internet
- A brief review of the Python `dict` data type
- Calculating basic metrics, how they are stored in `NetworkX` and how to manipulate them (list comps!)
- How to use `matplotlib` to visualize our analysis
- Getting data out of `NetworkX` both as raw network data or analytics using the `CSV` library
- Network visualization techniques in `NetworkX` and how to add network analysis to a visualization

Questions?