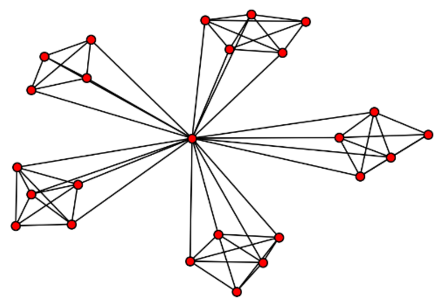
Charles Valentine

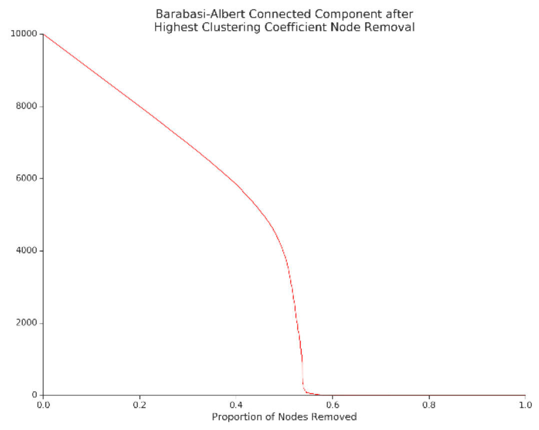
Homework 3

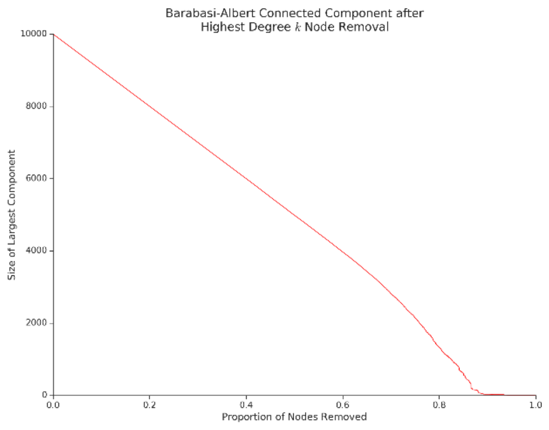
11/28/2016 at 5:30 PM

Problem 1
Code is at end of document

My computer struggled to perform this exercise and I could not think of a faster way around the computation. I have a cheap CPU and have already logged 7 hours of continuous computation on this problem. I cannot seem to finish the hierarchical model simulation due to an errant crash which does not seem to be caused by anything except the Python kernel itself failing. To prove my implementation of the hierarchical model I have provided this network graphic (1 iteration) showing that I can, at least, generate these networks.




Barabasi-Albert Connected Component after
Highest Clustering Coefficient Node Removal
10000
8000
6000
4000
2000
0
0.0
0.2
0.4
0.6
0.8
1.0
Proportion of Nodes Removed



For the Barabási-Albert model of a network the most successful attack strategy to fragment the network into the smallest connected components is to target, explicitly, high clustering coefficient nodes for removal. This makes intuitive sense since nodes with high clustering (and not necessarily high *k*) have a role in joining together clumps of the network. Nodes that are responsible for clustering the graph should be protected more so than nodes with a high degree *k*.

A simulation for random attacks which makes the assumption the attacker does not have any prior information about the role of a target node could be used to asses if all individual's information should be kept secret. It is likely that, for the Barabási-Albert model the network would be very resilient to random attacks as there are many hubs that provide redundant clustering roles and many, many low degree and low clustering nodes that would absorb many attacks with little effect on the network's structure.

Problem 2

a) The rate equation for the time evolution of the degree k_i of node i is:

$$\frac{dk_i}{dt} = m\prod_i = m\frac{\eta_i}{\sum_j \eta_j} = m\frac{\eta_i}{t\langle\eta\rangle} \quad \text{where} \quad m = 1$$

$$\frac{dk_i}{dt} = m\frac{\eta_i}{t\langle\eta\rangle} \quad \text{where} \quad m = 1$$

b) Integrating both sides with respect to t between t_i and t allows us to obtain the degree k_i for node i as a function of time:

$$\int_{t_i}^{t} \frac{dk_i}{dt} dt = \int_{t_i}^{t} t\frac{\eta_i}{\langle\eta\rangle} dt$$

$$k_i\,(t) - k_i(t_i) = \frac{\eta_i}{\langle\eta\rangle} \cdot (\ln(t) - ln(t_i)) \quad \text{where} \quad k_i(t_i) = 1$$

$$k_i(t) - k_i(t_i) = \frac{\ln(t)}{\ln(t_i)} \cdot \frac{\eta_i}{\langle\eta\rangle} \quad \text{where} \quad k_i(t_i) = 1 \quad \text{and} \quad t = N$$

$$k_i(t) = \ln\left(\frac{N}{t_i}\right) \cdot \frac{\eta_i}{\langle\eta\rangle} + 1$$

c) We will use the following inequality to assess the number of nodes with fitness η having a degree less than or equal to a given value, k.

$$k_i(t) \le k$$

$$\ln\left(\frac{N}{t_i}\right) \cdot \frac{\eta_i}{\langle\eta\rangle} + 1 \le k$$

$$\frac{N}{t_i} \le \mathrm{e}^{\frac{\langle\eta\rangle(k-1)}{\eta_i}}$$

$$\frac{N}{\mathrm{e}^{\frac{\langle\eta\rangle(k-1)}{\eta_i}}} \le t_i$$

The number of nodes with fitness η having a degree less than or equal to a given value, k, is:

$$\frac{N}{\mathrm{e}^{\frac{\langle\eta\rangle(k-1)}{\eta_i}}}$$

d) We obtain the cumulative distribution function of degree for the subset of nodes with fitness η by dividing by the total number of nodes of fitness η, N_η or $Np(\eta)$.

$$P(k) = \frac{1}{N} \cdot \frac{N}{\mathrm{e}^{\frac{\langle\eta\rangle(k-1)}{\eta_i}}} = \mathrm{e}^{\frac{\langle\eta\rangle(k-1)}{\eta_i}}$$

We can then take the derivative with respect to k to obtain the associated probability density function.

$$p_k = \frac{dP(k)}{dk} = \frac{\langle \eta \rangle e^{\frac{\langle \eta \rangle (k-1)}{\eta_i}}}{\eta_i}$$

e) If we assume that the fitness distribution $p(\eta)$ is half nodes with $\eta = 1$ and half nodes $\eta = 2$ the overall degree distribution of the network is:

$$\langle \eta \rangle = 1.5$$

$$p_k = \frac{1.5 \cdot e^{1.5 \cdot \frac{(k-1)}{\eta_i}}}{\eta_i}$$

The average degree of a good looking is the expected value $\mathrm{E}(\mathrm{X})$ for an exponential distribution. A good-looking node will have $\eta_i = 2$ so:

$$p_k = 0.75e^{0.75(k-1)} \quad \text{which is of the form} \quad \lambda e^{-\lambda x} \quad \text{where} \quad \mathrm{E}(X) = \frac{1}{\lambda}$$

Therefore the expected degree of a good looking node is estimated to be $\frac{4}{3}$. Using the same method we can estimate the expected value k for a less good looking node by setting $\eta_i = 1$.

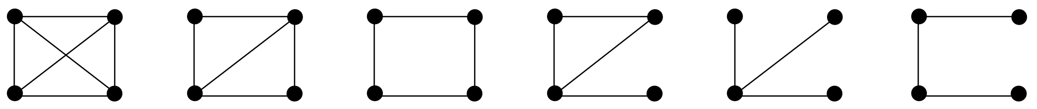
$$p_k = 1.5e^{1.5(k-1)} \quad \text{which is of the form} \quad \lambda e^{-\lambda x} \quad \text{where} \quad \mathrm{E}(X) = \frac{1}{\lambda}$$

The expected degree of a good looking node is estimated to be $\frac{2}{3}$.

Problem 3

a) For the following we discount shuffling of nodes as valid non-degenerate motifs. All six degenerate undirected 4 node motifs (with no unconnected nodes) are summarized in the following table. The number of directed motifs is calculated with only two link types incoming and outgoing directions. However, many of these non-degenerate motifs can be represented by a canonical set. To represent this set we need to remove the motifs that can be represented by a symmetrical mirroring. We can subtract this fraction of patterns by dividing by the number of nodes that lie on a potential symmetry line. There are 38 degenerate directed motifs.

$$M = \frac{2^L}{N \not\subset N_S}$$

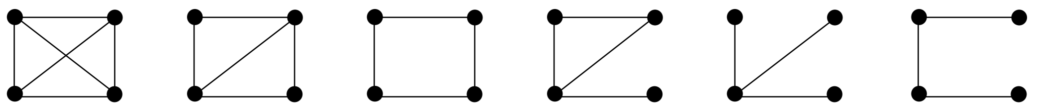


Nodes (N)	4	4	4	4	4	4
Edges (L)	6	5	4	4	3	3
Symmetries (S)	∞	2	∞	1	1	1
$N \not\subset N_S$	4	2	4	2	2	4
Non-degenerate Directed Motifs	64	32	16	16	8	8
Degenerate Directed Motifs	16	8	4	4	4	2

b) Estimating how many nodes in an Erdős–Rényi (ER) model graph can be approached by estimating the probability of multiple dependent links between nodes. For an ER graph with $N = 10{,}000$ nodes and the probability of any two nodes sharing an edge $p = 0.004$. We first start by estimating how many links exist in the ER graph as L_{total} and the avg. degree $\langle k \rangle$.

$$L_{total} = \frac{N^2 p}{2} = 20{,}000 \quad \text{and} \quad \langle k \rangle = Np = 40$$

Since every node, on average, has 40 links (subtract one from our source node) extending from it we have $L_{total} \cdot (\langle k \rangle - 1)$ motifs of three nodes in a row. For extending this model to three nodes we also need to subtract the proportion of links that return from our third node to our source node to keep our model as beads on a string. This subtraction term is represented as Np which is also the average degree of the node. Using this alternation of expressions depending on the motif structure we were able to estimate the following frequencies.



Estimated Abundance	233,99,760	231,99,760	231,99,920	311,999,920	234,00,000	304,199,920

Code

```
%load_ext autoreload
%autoreload 2
%matplotlib inline
import itertools
import numpy as np
import networkx as nx
import matplotlib as mpl
import matplotlib.pyplot as plt
from operator import itemgetter

mpl.rc('figure', facecolor='white')
mpl.rc('xtick', labelsize=14, color="#222222", direction='out')
mpl.rc('ytick', labelsize=14, color="#222222", direction='out')
mpl.rc('font', size=16)
mpl.rc('xtick.major', size=6, width=1)
mpl.rc('xtick.minor', size=3, width=1)
mpl.rc('ytick.major', size=6, width=1)
mpl.rc('ytick.minor', size=3, width=1)
mpl.rc('axes', linewidth=1, edgecolor="#222222", labelcolor="#222222")
mpl.rc('text', usetex=False, color="#222222")

def cleanup_chart_junk(ax):
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.yaxis.set_ticks_position('left')
    ax.xaxis.set_ticks_position('bottom')
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()
    return ax
```

```
def hierarchical_graph(i=1):
    # Cliques are defined as complete graphs, N=5, with all but one peripheral node.
    clique = nx.complete_graph(5)
    for node in clique.nodes():
        clique.node[node]['peripheral'] = False if node == 0 else True

    # The seed of our graph is a clique.
    G = clique.copy()

    # Every iteration
    for _ in range(i):
        G = nx.disjoint_union_all([G, *itertools.repeat(clique, 4)])
        G.add_edges_from([(0, node) for node in G.nodes() if G.node[node]['peripheral'] == True
])
        clique = G.copy()
    return G

nx.draw(hierarchical_graph(1), pos=nx.spring_layout(hierarchical_graph(1)), node_size=42)
```

```
np.random.seed(1)
N_BA = 1e4
i = 5

BA_C = nx.barabasi_albert_graph(N_BA, m=10)
BA_k = BA_C.copy()
HA_C = hierarchical_graph(i)
HA_k = HA_C.copy()
```

```
N_BA_C = []
N_BA_k = []

for x in range(1, len(BA_C)):
    # Remove node with highest clustering coefficient
    node = sorted(BA_C.degree().items(), key=itemgetter(1), reverse=True)[0][0]
    BA_C.remove_node(node)

    # Remove node with highest degree k
    node = sorted(nx.clustering(BA_k).items(), key=itemgetter(1), reverse=True)[0][0]
    BA_k.remove_node(node)

    # Record size of largest connected component
    N_BA_C.append(len(sorted(nx.connected_components(BA_C), key=len, reverse=True)[0]))
    N_BA_k.append(len(sorted(nx.connected_components(BA_k), key=len, reverse=True)[0]))
```

```
N_HA_C = []
N_HA_k = []

for x in range(1, len(HA_k)):
    # Remove node with highest clustering coefficient
    node = sorted(HA_C.degree().items(), key=itemgetter(1), reverse=True)[0][0]
    HA_C.remove_node(node)

    # Remove node with highest degree k
    node = sorted(nx.clustering(HA_k).items(), key=itemgetter(1), reverse=True)[0][0]
    HA_k.remove_node(node)

    # Record size of largest connected component
    N_HA_C.append(len(sorted(nx.connected_components(HA_C), key=len, reverse=True)[0]))
    N_HA_k.append(len(sorted(nx.connected_components(HA_k), key=len, reverse=True)[0]))
```

```
fig, axes = plt.subplots(2, 2, figsize=(15, 15))

data = [N_BA_C, N_BA_k, N_HA_C, N_HA_k,]
labels = ['Barabasi-Albert', 'Barabasi-Albert', 'Hierarchichal', 'Hierarchichal']
styles = itertools.cycle(['Highest Clustering Coefficient', 'Highest Degree $k$'])
sizes = itertools.cycle([N, N_HA])

for ax, y, label, style, size in zip(axes.flatten(), data, labels, styles, sizes):
    ax = cleanup_chart_junk(ax)
    ax.set_ylabel('Size of Largest Component')
    ax.set_title(label + ' Connected Component after\n' + style + ' Node Removal')
    ax.plot([x / size for x in range(0, len(y))], y, color='red')

for j in [0, 1]:
    axes[1, j].set_xlabel('Proportion of Nodes Removed')
fig.tight_layout()
```