Lab9 - Random Graphs

In this lab, we explore random graphs, introduced by Erdos and Renyi. You will need to install networkx in order to complete this lab (http://networkx.github.io/documentation/latest/install.html, http://stackoverflow.com/questions/9836909/easy-install-networkx). Many of you may already have networkx because it comes default with the Anaconda installation of iPython.

You will need the following basic imports as well as a function written to draw graphs for you. The structure of a graph object is a collection of edges, in (node1, node2) form. You should know how to use `draw_graph`, but you don't really need to know how it works. Play around with it to see what it does. Wow! Look at those pretty graphs **:**)

In [ ]:

```
%matplotlib inline
from pylab import *
import random as rnd
import networkx as nx
from __future__ import division

rcParams['figure.figsize'] = 12, 12  # that's default image size for this inte
ractive session

def draw_graph(graph, labels=None, graph_layout='shell',
               node_size=1600, node_color='blue', node_alpha=0.3,
               node_text_size=12,
               edge_color='blue', edge_alpha=0.3, edge_tickness=1,
               edge_text_pos=0.3,
               text_font='sans-serif'):
    """
    Based on: https://www.udacity.com/wiki/creating-network-graphs-with-python
    We describe a graph as a list enumerating all edges.
    Ex: graph = [(1,2), (2,3)] represents a graph with 2 edges - (node1 - node
2) and (node2 - node3)
    """

    # create networkx graph
    G=nx.Graph()

    # add edges
    for edge in graph:
        G.add_edge(edge[0], edge[1])

    # these are different layouts for the network you may try
    # shell seems to work best
    if graph_layout == 'spring':
        graph_pos=nx.spring_layout(G)
    elif graph_layout == 'spectral':
        graph_pos=nx.spectral_layout(G)
    elif graph_layout == 'random':
        graph_pos=nx.random_layout(G)
    else:
        graph_pos=nx.shell_layout(G)

    # draw graph
    nx.draw_networkx_nodes(G,graph_pos,node_size=node_size,
                           alpha=node_alpha, node_color=node_color)
    nx.draw_networkx_edges(G,graph_pos,width=edge_tickness,
                           alpha=edge_alpha,edge_color=edge_color)
    nx.draw_networkx_labels(G, graph_pos,font_size=node_text_size,
                            font_family=text_font)
    # show graph
    plt.show()
```

In [ ]:
```
graph = [(1,2),(2,3),(1,3)]
draw_graph(graph)
```

In [ ]:
```
graph = [(1,1),(2,2)]
draw_graph(graph) # no self-loops, so put a self-loop if you want a disconnect
ed node
```

Lets create a function that returns all the nodes that can be reached from a certain starting point given the representation of a graph above.

Q1. Fill out the following method, `find_connected_component`, that takes graph and starting_node as input arguments. It must return a set of nodes that are connected to the `starting_node`, including the `starting_node` itself.

In [ ]:
```
def find_connected_component(graph, starting_node):
    """
    >>> graph = [(1,2),(2,3),(1,3)]
    >>> find_connected_component(graph,1)
    {1, 2, 3}
    >>> graph = [(1,1),(2,3),(2,4),(3,5),(3,6),(4,6),(1,7),(7,8),(1,8)]
    >>> find_connected_component(graph,1)
    {1, 7, 8}
    >>> find_connected_component(graph,2)
    {2, 3, 4, 5, 6}
    """
    connected_nodes = set()
    connected_nodes.add( starting_node )
    #Your code here

    return connected_nodes
```

Q2. Fill out the following method, `connected_components`, that takes graph and returns all the connected components of the graph. You may want to use the function you wrote above.

In [ ]:
```
def connected_components(graph):
    """
    >>> graph = [(1,1),(2,3),(2,4),(3,5),(3,6),(4,6),(1,7),(7,8),(1,8)]
    >>> connected_components(graph)
    [{1, 7, 8}, {2, 3, 4, 5, 6}]
    >>> largest_component_size(graph)
    5
    """
    nodes = set()
    components = []

    # Your code here

    return components

# These guys should work after you've implemented connected_components
component_sizes = lambda graph: [len(component) for component in (connected_co
mponents(graph))]
largest_component_size = lambda graph: max(component_sizes(graph))
```

Next, we want to create a function that, given the number of nodes in a graph, will randomly generate edges between nodes. That is, we want to construct a random graph folowing the Erdos-Renyi model.

Q3. Fill out the following function to create an Erdos-Renyi random graph \(\operatorname{G}(n,p)\). For each pair of nodes, randomly create an edge between them with probability \(p\). Return the resulting graph (same format as before).

In [ ]:
```
def G(n,p):
    graph = []
    # Recall that we describe a graph as a list enumerating all edges. Node na
mes can be numbers.

    #Your code here

    return graph
```

In [ ]:
```
# Try this!
graph = G(10,0.1)
draw_graph(graph)
```

Phase Transitions!!!

Now let's examine some of the qualatative properties of a random graph developed in the original Erdos & Renyi paper.

In [ ]:
```
epsilon = 1/100
```

Transition 1: If \(np < 1\), then a graph in \(\operatorname{G}(n, p)\) will almost surely have no connected components of size larger than \(\operatorname{O}(\log(n))\)

In [ ]:
```
largest_sizes = []
n = 50
p = 1/50 - epsilon
for i in range(1000):
    graph = G(n,p)
    largest_sizes.append(largest_component_size(graph))



print "We expect the largest component size to be on the order of: ", np.log2(
n)
print "True average size of the largest component: ", np.mean(largest_sizes)
```

Transition 2: If \(np = 1\), then a graph in \(\operatorname{G}(n, p)\) will almost surely have a largest component whose size is of order \(n^{2/3}\).

In [ ]:
```
largest_sizes = []
n = 50
p = 1/50
for i in range(1000):
    graph = G(n,p)
    largest_sizes.append(largest_component_size(graph))



print "We expect the largest component size to be on the order of: ", n**(2/3)
print "True average size of the largest component: ", np.mean(largest_sizes)
```

Transition 3: If \(np → c > 1\), where \(c\) is a constant, then a graph in \(\operatorname{G}(n,p)\) will almost surely have a unique giant component containing a positive fraction of the vertices. No other component will contain more than \(\operatorname{O}(\log(n))\) vertices.

We'll increase the number of nodes by a factor of 10 here so we can see this more clearly. Pay attention the precipitious decline from the size of the largest connected tomponent to that of all the rest.

In [ ]:
```
largest_sizes = []
epsilon = 1/10000
n = 5000
p = 1/5000 + epsilon
graph = G(n,p)
print sorted(component_sizes(graph))[::-1]
```

Transition 4: If \(p<\tfrac{(1-\epsilon)\ln n}{n}\), then a graph in \(\operatorname{G}(n,p)\) will almost surely contain isolated vertices, and thus be disconnected.

In [ ]:
```
largest_sizes = []
epsilon = 1/20000
n = 10000
p = (1-epsilon)*np.log(n) / n - epsilon
num_isolated = 0
for _ in range(10):
    graph = G(n,p)
    if 1 in component_sizes(graph):
        num_isolated += 1
print "Probability of graphs containing isolated vertices: ", num_isolated / 1
0
```

Transition 5: If \(p>\tfrac{(1-\epsilon)\ln n}{n}\), then a graph in \(\operatorname{G}(n,p)\) will almost surely be connected.

In [ ]:
```
largest_sizes = []
epsilon = 1/20000
n = 10000
p = (1-epsilon)*np.log(n) / n + epsilon
num_isolated = 0
for _ in range(10):
    graph = G(n,p)
    if 1 in component_sizes(graph):
        num_isolated += 1
print "Probability that graphs are connected: ", 1 - num_isolated / 10
```

Cool! Now we've experimentally verified the results of the Erdos-Reyni paper. Isn't it neat that you can rigorously formalize this kind of qualatative behavior of a graph? And that you can clearly see these transitions in simulation? I think it's cool.