Analysis of Large-Scale Networks

Analysis of Empirical Networks

Jukka-Pekka Onnela

Department of Biostatistics
Harvard School of Public Health

June 20, 2012

About the Exercises

- We will now practice Python and NetworkX by writing a number of functions
- It makes sense to develop the functions in the interactive environment
- Once a function is completed, we will include it in a module to facilitate code reuse
- This will eventually result in a small customized network library
- This module, call it `mynetlib.py`, should have the following type of structure:

```python
# mynetlib.py
import module1
import module2
...
def myfunc1():
    ...
def myfunc2():
    ...
```

- It is useful to have the import statements at the beginning for better readability

About the Exercises

- Make sure to add the following imports to `mynetlib.py`:

```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
import random
```

- As usual, the module can be imported in two different ways:

```
import mynetlib
...
mynetlib.myfunc1()
```

```
from mynetlib import *
...
myfunc1()
```

- For brevity, we have followed the latter convention

The Enron Case Study

- The Enron email communication network consists of some 500,000 emails exchanged by Enron employees
- These data were originally made public and posted on the web by the Federal Energy Regulatory Commission during its investigation
- Nodes of the network are email addresses of the employees
- If person i sent at least one email to person j, the two nodes are connected by an undirected edge
- Only emails sent within Enron are visible in the data set
- The network consists of 36,692 nodes and 367,662 (double-counted) edges
- The data can be downloaded from `http://snap.stanford.edu/data/email-Enron.html`
- We assume in the following that a copy of the data are available in a local file named `email-Enron.txt`

Exercise 1

Exercise 1: Reading Input

Write your own function that reads in the network data as an edge list. Allow for arbitrary field separators (e.g., `\t` or `,`) and for arbitrary comment line characters (e.g., # or $).

Exercise 1: Reading Input

Write your own function that reads in the network data as an edge list. Allow for arbitrary field separators (e.g., `\t` or `,`) and for arbitrary comment line characters (e.g., # or $).

```
# Routine for reading in a network as an edge list.
def readnet(input_file, sep_char, comment_char):
    G = nx.Graph()
    for line in open(input_file):
        if line[0] != comment_char:
            line = line.rstrip().split(sep_char)
            G.add_edge(int(line[0]), int(line[1]))
    return G
```

Exercise 1: Reading Input

Write your own function that reads in the network data as an edge list. Allow for arbitrary field separators (e.g., `\t` or `,`) and for arbitrary comment line characters (e.g., # or $).

```python
# Routine for reading in a network as an edge list.
def readnet(input_file, sep_char, comment_char):
    num_lines = 0
    G = nx.Graph()
    for line in open(input_file):
        num_lines += 1
        if line[0] != comment_char:
            line = line.rstrip().split(sep_char)
            if len(line) == 2:
                G.add_edge(int(line[0]), int(line[1]))
    print 'Read ' + str(num_lines) + ' lines.'
    print 'Network has %d nodes and %d edges.' % (G.number_of_nodes(), \
                                                   G.number_of_edges())
    return G
```

Exercise 1

Exercise 1: Reading Input

Write your own function that reads in the network data as an edge list. Allow for arbitrary field separators (e.g., `\t` or `,`) and for arbitrary comment line characters (e.g., # or $).

```
# Routine for reading in a network as an edge list.
def readnet(input_file, sep_char, comment_char):
    num_lines = 0
    G = nx.Graph()
    for line in open(input_file):
        num_lines += 1
        if line[0] != comment_char:
            line = line.rstrip().split(sep_char)
            if len(line) == 2:
                G.add_edge(int(line[0]), int(line[1]))
                if num_lines < 10:
                    print line[0] + " " + line[1]
                if num_lines == 10:
                    print "..."
    print 'Read ' + str(num_lines) + ' lines.'
    print 'Network has %d nodes and %d edges.' % (G.number_of_nodes(), \
                                                  G.number_of_edges())
    return G
```

Exercise 2

Exercise 2: Writing Output

Write your own function that writes network data to a file as an edge list. Allow for arbitrary field separators (e.g., `\t` or `,`).

Exercise 2: Writing Output

Write your own function that writes network data to a file as an edge list. Allow for arbitrary field separators (e.g., `\t` or `,`).

```
# Routine for writing a network as an edge list.
def writenet(G, output_file, sep_char):
    num_lines = 0
    F = open(output_file, 'w')
    for edge in G.edges():
        F.write(str(edge[0]) + sep_char + str(edge[1]) + '\n')
        num_lines += 1
    F.close()
    print 'Wrote ' + str(num_lines) + ' lines.'
```

Exercise 3a

Exercise 3a: Making Plots

Explore `plt.plot()`, `plt.loglog()`, and `np.histogram()` functions.

Exercise 3a: Making Plots

Explore `plt.plot()`, `plt.loglog()`, and `np.histogram()` functions.

```
import matplotlib.pyplot as plt
import numpy as np
import random

xs = range(21)
ys = [x**2 for x in xs]
plt.plot(xs, ys)
plt.show()

plt.plot(xs, ys, marker = "o")
plt.xlabel("x-variable")
plt.ylabel("y-variable")
plt.show()

plt.loglog(xs, ys, marker = "o")
plt.show()

data = [random.normalvariate(0,1) for i in xrange(100)]
(bin_counts, bin_edges) = np.histogram(data)
bin_centers = [(bin_edges[i]+bin_edges[i+1])/2 for i in range(len(bin_edges)-1)]
plt.plot(bin_centers, bin_counts, marker = "o")
plt.show()
```

Exercise 3b

Exercise 3b: Degree Distribution

Plot vertex degree distribution for the network. Compute average and median degree.

Exercise 3b: Degree Distribution

Plot vertex degree distribution for the network. Compute average and median degree.

```
def degree_distribution(G, number_of_bins):
    degree_sequence = G.degree().values()
    min_degree = np.min(degree_sequence)
    max_degree = np.max(degree_sequence)
    mean_degree = np.mean(degree_sequence)
    median_degree = np.median(degree_sequence)
    # Print some basics.
    print "Minimum degree is %.3f." % min_degree
    print "Maximum degree is %.3f." % max_degree
    print "Mean degree is %.3f." % mean_degree
    print "Median degree is %.3f." % median_degree
    # Make the plot.
    fig = plt.figure(figsize=(10, 10))
    y, bin_edges = np.histogram(G.degree().values(), density=True, bins=number_of_bins)
    x = [(bin_edges[i] + bin_edges[i+1]) / 2 for i in range(len(bin_edges) - 1)]
    plt.loglog(x, y, marker="o", markersize=10, basex=10, basey=10)
    plt.xlabel("log k", fontsize=15); plt.ylabel("log P(k)", fontsize=15)
    plt.ylim((10**-7, 1)); plt.xlim((1, 10**4))
    #plt.show()
    fig.savefig("../figs/enron_degree_distribution.pdf")
```

Exercise 3b: Degree Distribution

Plot vertex degree distribution for the network. Compute average and median degree.

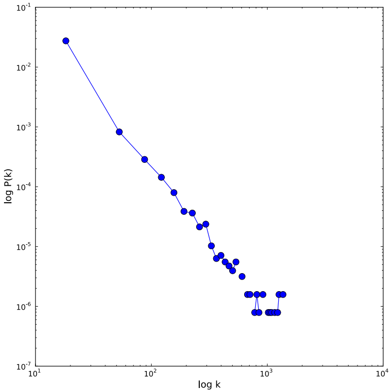

Figure: Degree distribution for the Enron email network. The minimum and maximum degrees are 1 and 1383, respectively. Median degree is 3, and average degree $\langle k \rangle = 10.02$.

Exercise 4: Sociocentric Sample

Write a function that returns a sociocentric sample on a given list of nodes.

Exercise 4: Sociocentric Sample

Write a function that returns a sociocentric sample on a given list of nodes.

```
def sociocentric_sample(G, nodes):
    g = nx.subgraph(G, nodes)
    return g
```

Exercise 5

Exercise 5: Simple Random Sample

Write a function that carries out a simple random sample (i.e., nodes chosen uniformly at random from the network).

Exercise 5: Simple Random Sample

Write a function that carries out a simple random sample (i.e., nodes chosen uniformly at random from the network).

```
def simple_random_sample(G, num_sample_nodes):
    if num_sample_nodes > G.number_of_nodes():
        print 'Sample size exceeds population size.'
        return None
    else:
        sample_nodes = set()
        while len(sample_nodes) < num_sample_nodes:
            sample_nodes.add(random.choice(G.nodes()))
        return sample_nodes
```

Exercise 5

Exercise 5: Simple Random Sample

Write a function that carries out a simple random sample (i.e., nodes chosen uniformly at random from the network).

```
def simple_random_sample(G, num_sample_nodes):
    if num_sample_nodes > G.number_of_nodes():
        print 'Sample size exceeds population size.'
        return None
    else:
        sample_nodes = set()
        while len(sample_nodes) < num_sample_nodes:
            sample_nodes.add(random.choice(G.nodes()))
        return sample_nodes
```

- You could return a list of nodes instead of a set of nodes by using `return list(sample_nodes)`
- A shorter approach to returning a sample is to use the `random.sample()` method

```
random.sample(G.nodes(), num_sample_nodes)
```

Exercise 6

Exercise 6: Snowball Sample

Write a function that returns a snowball sample of the nodes of a network (i.e., no edges, just the nodes). The algorithm should start from a given node, and it should stop when it reaches a pre-specified layer, i.e., when it reaches nodes at a given distance from the source node.

Exercise 6

Exercise 6: Snowball Sample

Write a function that returns a snowball sample of the nodes of a network (i.e., no edges, just the nodes). The algorithm should start from a given node, and it should stop when it reaches a pre-specified layer, i.e., when it reaches nodes at a given distance from the source node.

```
def snowball_sample(G, node, curr_layer, max_layer):
    global sample_nodes
    sample_nodes.add(node)
    if curr_layer < max_layer:
        for neighbor in G.neighbors(node):
            snowball_sample(G, neighbor, curr_layer + 1, max_layer)
```

Exercise 6

Exercise 6: Snowball Sample

Write a function that returns a snowball sample of the nodes of a network (i.e., no edges, just the nodes). The algorithm should start from a given node, and it should stop when it reaches a pre-specified layer, i.e., when it reaches nodes at a given distance from the source node.

```python
def snowball_sample_opt(G, node, parent, curr_layer, max_layer):
    global sample_nodes
    sample_nodes.add(node)
    new_neighbors = G.neighbors(node)
    if parent in new_neighbors:
        new_neighbors.remove(parent)
    if curr_layer < max_layer:
        for neighbor in new_neighbors:
            snowball_sample_opt(G, neighbor, node, curr_layer + 1, max_layer)
```

Exercise 8

Exercise 8: Random Failure of Nodes

See what happens to the network, quantified using the degree distribution, as you delete some fraction of the nodes. The deleted nodes should be chosen uniformly at random, i.e., each node should have the same probability to be chosen for deletion. This process is sometimes referred to as ”random failure” of nodes.

Exercise 8: Random Failure of Nodes

See what happens to the network, quantified using the degree distribution, as you delete some fraction of the nodes. The deleted nodes should be chosen uniformly at random, i.e., each node should have the same probability to be chosen for deletion. This process is sometimes referred to as ”random failure” of nodes.

```
def simple_random_sample(G, num_sample_nodes):
    if num_sample_nodes > G.number_of_nodes():
        print 'Sample size exceeds population size.'
        return None
    else:
        sample_nodes = set()
        while len(sample_nodes) < num_sample_nodes:
            sample_nodes.add(random.choice(G.nodes()))
        return sample_nodes
```

```
def random_node_deletion(G, fraction_to_delete):
    if fraction_to_delete > 1:
        print 'Trying to delete more nodes than exist in the network.'
        return None
    number_to_delete = int(round(fraction_to_delete * G.number_of_nodes()))
    nodes_to_delete = simple_random_sample(G, number_to_delete)
    G.remove_nodes_from(nodes_to_delete)
    return None
```

```
G = readnet("./../data/email-Enron.txt", "\t", "#")
degree_distribution(G, 30)
random_node_deletion(G, 0.3333)
degree_distribution(G, 30)
```

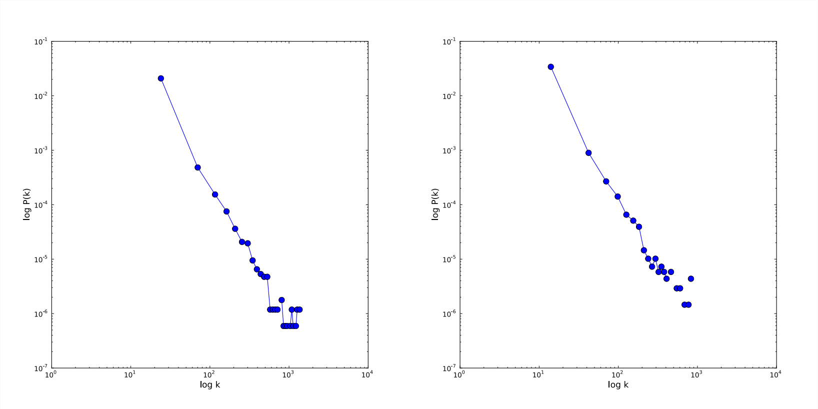

Figure: Original (left) and sampled (right) degree distribution for the Enron email network.

Exercise 8

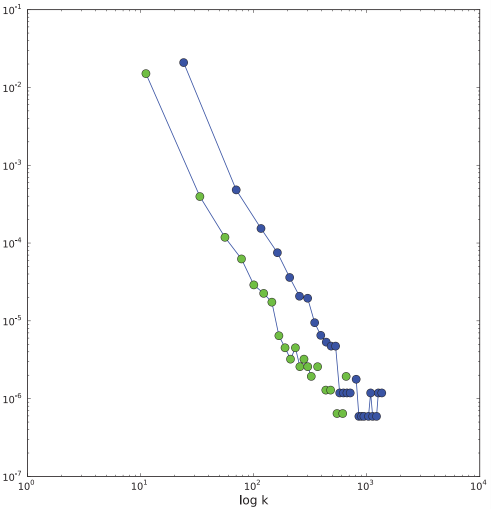

Figure: Original (blue) and sampled (green) degree distribution. The former has $\langle k \rangle = 10.020$, whereas the latter has $\langle k \rangle = 6.692$; the median degrees are 3 and 2, respectively. The sampled distribution has shifted to the left.

Exercise 9

Exercise 9: Targeted Attack of Nodes

See what happens to the network, quantified using the degree distribution, as you delete some fraction of the nodes. Instead of choosing the nodes uniformly at random, let the node selection probability depend on the degree of the node. More specifically, make the node deletion probability proportional to node degree. For example, a node of degree 6 should be twice as likely to be chosen for deletion than a node of degree 3.

Exercise 9

Exercise 9: Targeted Attack of Nodes

```
def degree_biased_sample(G, num_sample_nodes):
    if num_sample_nodes > G.number_of_nodes():
        print 'Sample size exceeds population size.'
        return None
    else:
        sample_nodes = set()
        master_list = []
        for (node_id, node_degree) in G.degree().items():
            master_list.extend([node_id for x in range(node_degree)])
        while len(sample_nodes) < num_sample_nodes:
            sample_nodes.add(random.choice(master_list))
        return sample_nodes
```

```
def degree_biased_node_deletion(G, fraction_to_delete):
    if fraction_to_delete > 1:
        print 'Trying to delete more nodes than exist in the network.'
        return None
    number_to_delete = int(round(fraction_to_delete * G.number_of_nodes()))
    nodes_to_delete = degree_biased_sample(G, number_to_delete)
    G.remove_nodes_from(nodes_to_delete)
    return None
```

Exercise 9

```
G = readnet("./../data/email-Enron.txt", "\t", "#")
degree_distribution(G, 30)
degree_biased_node_deletion(G, 0.2)
degree_distribution(G, 30)
```

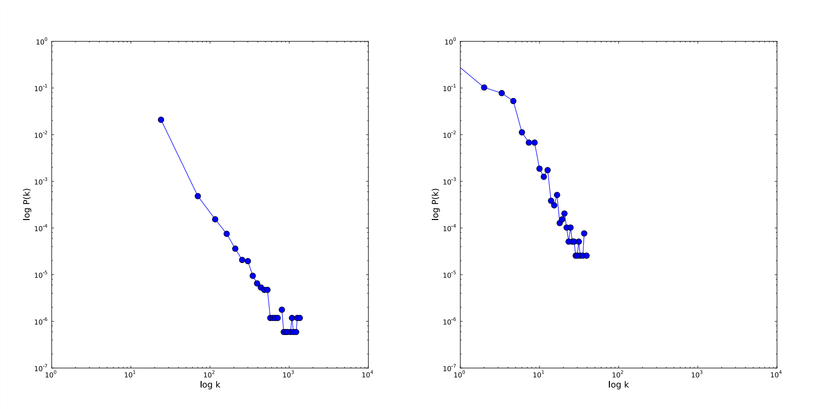

Figure: Original (left) and sampled (right) degree distribution for the Enron email network.

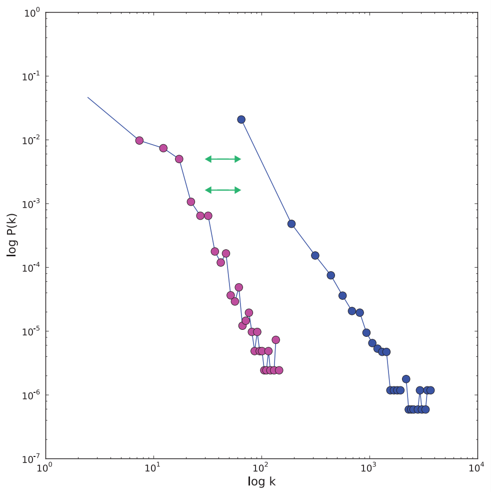

Figure: Original (blue) and sampled (green) degree distribution. The former has $\langle k \rangle = 10.020$, whereas the latter has $\langle k \rangle = 1.481$; the median degrees are 3 and 1, respectively. The tail of the sampled distribution appears bent to the left. Note that both distributions have been moved horizontally to make the effect of sampling more visible.

Exercise 10: Random Walk

Let's do some random walking on a network. Write two functions, where the first one selects a neighbor of a given node uniformly at random, and the second one uses the first function to actually perform the walk. The second function will need the starting node of the walk as a parameter, as well as the number of steps to be taken, and it should return the path of the entire walk (including the starting node) as a list.

Exercise 10: Random Walk

Let's do some random walking on a network. Write two functions, where the first one selects a neighbor of a given node uniformly at random, and the second one uses the first function to actually perform the walk. The second function will need the starting node of the walk as a parameter, as well as the number of steps to be taken, and it should return the path of the entire walk (including the starting node) as a list.

```
def select_neighbor(G, source):
    return random.choice(G[source].keys())

def perform_walk(G, source, num_steps):
    path = [source]
    for step in range(num_steps):
        source = select_neighbor(G, source)
        path.append(source)
    return path
```

Exercise 10

```
G = readnet("./../data/email-Enron.txt", "\t", "#")
select_neighbor(G, 123)
select_neighbor(G, 123)
select_neighbor(G, 123)
```

```
823
277
1028
```

Exercise 10

```
G = readnet("./../data/email-Enron.txt", "\t", "#")
select_neighbor(G, 123)
select_neighbor(G, 123)
select_neighbor(G, 123)
```

```
823
277
1028
```

```
perform_walk(G, 123, 5)
perform_walk(G, 123, 5)
perform_walk(G, 123, 5)
```

```
[123, 4398, 1953, 1202, 19366, 29385]
[123, 2773, 4230, 2736, 16533, 20634]
[123, 86, 5520, 86, 5524, 109]
```

Exercise 11: Random Walk and Node Degrees

How does the average degree of nodes encountered along the path taken by a random walk change as the walk proceeds? Generate first a Barabási-Albert network with $N = 10,000$ and $m = 3$. Perform a large number of random walks, say, 1000, each walk starting from a randomly chosen source node and consisting of 100 steps. Compute and plot the average degree of nodes encountered by the walk as a function of the number of steps taken. As a reference, include also the overall network average degree in the plot.

Exercise 11: Random Walk and Node Degrees

How does the average degree of nodes encountered along the path taken by a random walk change as the walk proceeds? Generate first a Barabási-Albert network with $N = 10,000$ and $m = 3$. Perform a large number of random walks, say, 1000, each walk starting from a randomly chosen source node and consisting of 100 steps. Compute and plot the average degree of nodes encountered by the walk as a function of the number of steps taken. As a reference, include also the overall network average degree in the plot.

```
def random_walk_degree(G, num_steps = 100, num_reps = 1000):
   degree_counts = {}
   for step in range(num_steps + 1):
      degree_counts[step] = list()

   for rep in range(num_reps):
       source = random.choice(G.nodes())
       path = perform_walk(G, source, num_steps)
       for step in range(num_steps + 1):
         degree_counts[step].append(G.degree(path[step]))

   average_degree = {}
   for step in range(num_steps + 1):
      average_degree[step] = np.mean(degree_counts[step])

   return average_degree.values()
```

Exercise 11

```python
def run_random_walk_degree(G, num_steps, num_reps, filename):
    ave_walk_degree = random_walk_degree(G, num_steps, num_reps)
    ave_deg = np.mean(G.degree().values())
    fig = plt.figure(figsize=(10, 10))
    plt.plot(range(101), ave_walk_degree, marker="o", markersize=5)
    plt.plot([0, 100], [ave_deg, ave_deg])
    plt.xlabel("Random walk length", fontsize=10)
    plt.ylabel("Average degree", fontsize=10)
    plt.xlim((0, 100))
    plt.ylim((0, 25))
    if filename == "": plt.show()
    else: fig.savefig(filename)
```

```python
# Let's do this for the BA network first.
G = nx.barabasi_albert_graph(10000, 3)
run_random_walk_degree(G, 100, 100000, "../figs/random_walk_degree_ba.pdf")

# But let's also do this for the ER network.
H = nx.erdos_renyi_graph(10000, 6 / float(10000-1))
components = nx.connected_components(H)
run_random_walk_degree(H.subgraph(components[0]), 100, 100000, "../figs/
    random_walk_degree_er.pdf")
```

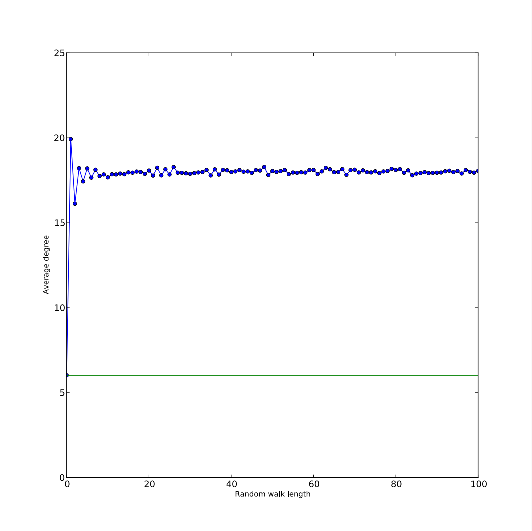

Figure: Average degree of nodes encountered in a random walk on a Barabási-Albert network with $N = 10000$ and $m = 3$.

Exercise 11

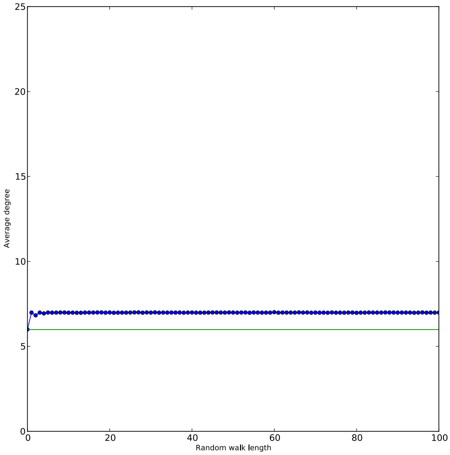

Figure: Average degree of nodes encountered in a random walk on an Erdős-Rényi network with $N = 10000$ and $p = 0.0006$.

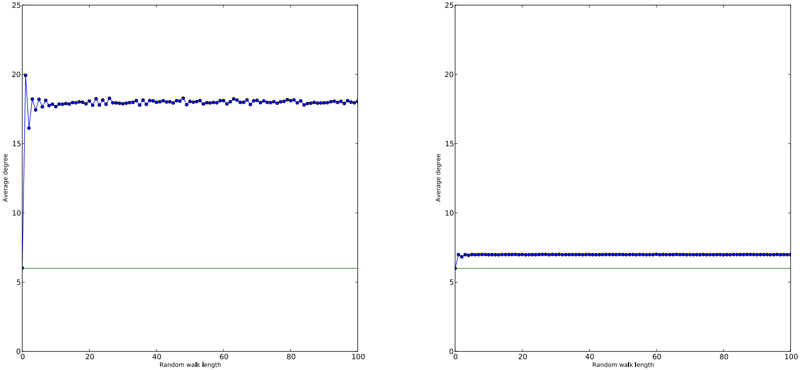

Figure: Average degree of nodes encountered in a random walk on a Barabási-Albert network (left) and Erdős-Rényi network (right).