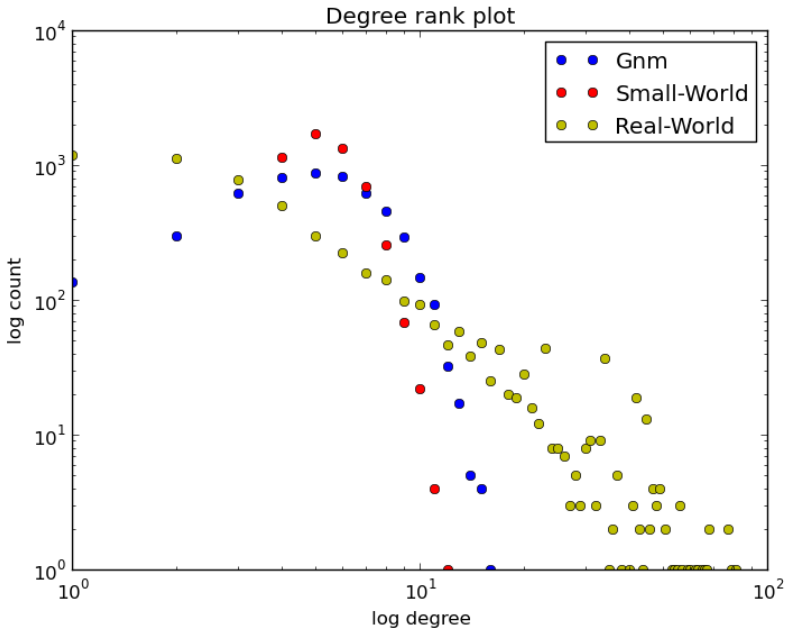
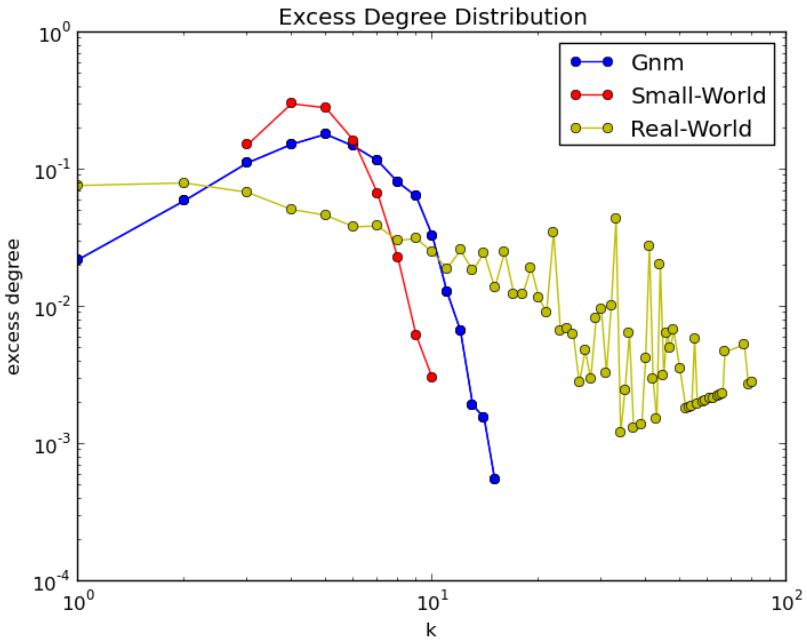
224w Hwk1
Ryan Foley

1a)





Expected Excess Degree for 3 graphs:

Gnm: 5.55822295806
SW: 4.80449779249
RW: 15.8713940649

Expected Degree for the 3 graphs:

Gnm: 5.53071346814
SW: 5.53071346814
RW:5.53071346814

Real World Collaboration network has more counts of higher degrees given by looking at the degree histograms for the 3 graphs. This shows that the real world graph has a higher level of connectedness - in class we discussed that real world graphs tend to have higher clustering and this is evidence of that.

Code for making hist of degrees:

```
import matplotlib.pyplot as plt
import matplotlib
import networkx as nx
from random import *
import excess_degree as ed

degree_sequence1=sorted(nx.degree(gnm).values(),reverse=True)
degree_sequence2=sorted(nx.degree(sw).values(),reverse=True)
degree_sequence3=sorted(nx.degree(rw).values(),reverse=True)
counts1 = {}
counts2 = {}
counts3 = {}
for degree in degree_sequence1:
  if(degree in counts1):
    counts1[degree] += 1
  else: counts1[degree] = 1

for degree in degree_sequence2:
  if(degree in counts2):
    counts2[degree] += 1
  else: counts2[degree] = 1

for degree in degree_sequence3:
  if(degree in counts3):
    counts3[degree] += 1
  else: counts3[degree] = 1

x = []
y = []
for degree in counts1:
  x.append(degree)
  y.append(counts1[degree])

p1, = plt.loglog(x, y,'bo')


x = []
y = []
for degree in counts2:
  x.append(degree)
```

```
  y.append(counts2[degree])

p2, = plt.loglog(x, y,'ro')

x = []
y = []
for degree in counts3:
  x.append(degree)
  y.append(counts3[degree])

p3, = plt.loglog(x, y,'yo')

plt.title("Degree rank plot")
plt.ylabel("log count")
plt.xlabel("log degree")
plt.legend([p1, p2, p3], ["Gnm", "Small-World", "Real-World"])
plt.savefig("test3.png")
plt.close()
```

Code for finding Expected Degree and Expected Excess Degree:

```
import networkx as nx
import matplotlib.pyplot as plt
import matplotlib

def get_q(G):
  V = G.nodes()
  E = G.edges()
  degree_sequence=sorted(nx.degree(G).values(),reverse=True)
  dmax = max(degree_sequence)
  q_prime = {}
  for k in range(dmax + 1):
    for node in V:
      neighbors = G.neighbors(node)
      for neighbor in neighbors:
        if(G.degree(neighbor) == k + 1 ):
          if(k in q_prime):
            q_prime[k] += 1
          else: q_prime[k] = 1

  q = {}
  for k in q_prime:
    q[k] = (q_prime[k] / float(sum(q_prime.values())))
  return q

def get_p(G):
  V = G.nodes()
  E = G.edges()
  degree_sequence=nx.degree(G).values()
  degree_counts = {}
  for degree in degree_sequence:
    if(degree in degree_counts):
      degree_counts[degree] += 1
    else:
      degree_counts[degree] = 1

  p = {}
  for degree in degree_counts:
```

```
    p[degree] = degree_counts[degree]/float(sum(degree_counts.values()))

  return p

def exp_excess_degree(q):
  exp_deg = 0
  for k in q:
    exp_deg += k * q[k]
  return exp_deg

def exp_degree(p):
  exp_deg = 0
  for k in p:
    exp_deg += k*p[k]
  return exp_deg


#Expected Excess Degree for 3 graphs:
print "Expected Excess Degree for 3 graphs:"
q1 = get_q(gnm)
print exp_excess_degree(q1)
q2 = get_q(sw)
print exp_excess_degree(q2)
q3 = get_q(rw)
print exp_excess_degree(q3)

#Expected Degree for the 3 graphs:
print "Expected Degree for the 3 graphs:"
p = get_p(gnm)
print exp_degree(p)
p = get_p(sw)
print exp_degree(p)
p = get_p(rw)
print exp_degree(p)
```

Plotting Excess Degree Distn:

```
x = []
y = []
for i in q1:
  x.append(i)
  y.append(q1[i])
p1, = plt.loglog(x,y,'b',marker='o')

x = []
y = []
for i in q2:
    x.append(i)
    y.append(q2[i])
p2, = plt.loglog(x,y,'r',marker='o')

x = []
y = []
for i in q3:
    x.append(i)
    y.append(q3[i])
p3, = plt.loglog(x,y,'y',marker='o')
```

```
plt.title("Excess Degree Distribution")
plt.ylabel("excess degree")
plt.xlabel("k")
plt.legend([p1, p2, p3], ["Gnm", "Small-World", "Real-World"])
plt.savefig("excess_degree_histogram3.png")
plt.close()
```

[5 points] Can you calculate the excess degree distribution qk using the degree distribution pk? Find this formula.

$$q_k = \frac{(k+1)*N*p_{k+1}}{M} \text{ where } N \text{ is the total number of nodes and } M \text{ is the total number of edges}$$

Note: in calculating the number of edges, I am assuming that the graph is undirected, but that (A,B) and (B,A) are counted as only one edge - hence the times 2.

1c)
Average clustering coefficient for each graph using my code:

Gnm: 0.000605801845787
SW: 0.284896479374
RW: 0.531466928675

Using built in function average_clustering() to check:

```
>>> nx.average_clustering(gnm)
0.0005610280600742255
>>> nx.average_clustering(sw)
0.2834199904974384
>>> nx.average_clustering(rw)
0.5296358110521363
```

It is not surprising that we would see the highest clustering coefficient in the Real-World graph. In class, we discussed that it is more common in real world graphs to see higher clusterings. However, we see that Small-World graph gets much closer to that of the Real World than the Gnm model. This is because of the structure associated with the Small World model. Each node is connected in a ring and there are randomly selected paths across the ring making the average path between any two nodes much shorter and increasing the average clustering coefficient. The clustering coefficient for the random Gnm model is nearly non existent because very rarely are multiple nodes connected together when so few edges are assigned at random.

Given that there are n = 5242 nodes, there are $\binom{n}{2}$ = n(n-1)/2 = 13736661 total possible edges. Thus, if we are only assigning m = 14496 edges, it seems unlikely that they would form clusterings that would make for high clustering coefficients.

```
def get_c(G):
    sum = 0
    for node in G.nodes():
        k = G.degree(node)
        if k > 1:
            for neighbor in G.neighbors(node):
                for otherNeighbor in G.neighbors(node):
```

```
                    if otherNeighbor in G.neighbors(neighbor):
                        sum += 1/float(k * (k-1))
    print sum/float(len(G.nodes()))
```

2)
a) We would look for the largest connected component. It wouldn't necessarily have to be strongly connected because we wouldn't need that there be a path from A to B and B to A, but rather just the longest connected component of comprised of nodes in the paths from the first infected node. Furthermore, it couldn't be weakly connected because direction is important in the dispersion of the virus.

Essentially, the most infected nodes would be the set of nodes in every possible path from the first infected node to every other node that it can reach.

b)
The graph contains 85591 nodes in total.
i)
22868 nodes are in the largest SCC, so the SCC makes up 22868/85591 ~ 26.7 % of the number of nodes.

```
import networkx as nx
G = nx.read_adjlist('email_network.txt', create_using=nx.DiGraph())
scc = nx.strongly_connected_components(G)
print len(scc[0])
```

ii)*Size of the in-component of the largest SCC:*
There are 8579 nodes that have a path connecting them to the largest SCC, so 8579/85591 ~ 10% of the nodes in the graph are of the in component of the largest SCC.

```
import networkx as nx
from sets import Set

G = nx.read_adjlist('email_network.txt', create_using=nx.DiGraph())
scc = nx.strongly_connected_components(G)
b_scc = scc[0]

nodes_in = Set()
for node in G.nodes():
  if len(nodes_in) % 10 == 0: print len(nodes_in), "InNodes Found"
  if node not in nodes_in and node not in b_scc:
    for other in b_scc:
      try:
        path = nx.shortest_path(G,node, other)
        if len(path) > 0:
          nodes_in.add(node)
          break
      except:
        place_holder = 0

print "Nodes in =", len(nodes_in)
```

iii) *Size of the out-component of the largest SCC:*
There exists a path from the largest SCC to 12319 nodes that are not in the largest SCC, so 12319/85591 ~ 14.4% of the graph is of the out component of the largest SCC.

```
import networkx as nx
from sets import Set
from Queue import *

G = nx.read_adjlist('email_network.txt', create_using=nx.DiGraph())

scc = nx.strongly_connected_components(G)

print len(scc[0])

biggest_scc = scc[0]

nodes_out = Set()
nodes_q = Queue()
#Initialize queue:
for node in biggest_scc:
  nodes_q.put(node)

#Find nodes out of queue.
nodes_out = Set()
while(nodes_q.qsize() != 0):
  print "Queue size =", nodes_q.qsize()
  node = nodes_q.get()
  neighbors = G.neighbors(node)
  for neighbor in neighbors:
    if neighbor not in biggest_scc and neighbor not in nodes_out:
      nodes_out.add(neighbor)
      nodes_q.put(neighbor)

print "Nodes out size:", len(nodes_out)
```

iv)
85591 - 22868 - 8579 - 12319 = 41825, so about **48.9%** of the graph.

2c)
If the email is dropped in any of the largest SCC (22868 nodes), then at the least the SCC and the out component are infected which is about 26.7% + 14.4% > 30%.

If the email is dropped in the in-component of the largest SCC (8579 nodes), then the minimum result is that the same number of nodes are infected as if it were in the SCC.

If dropped in the out component the number infected would be less than 30% of the graph.

So, we can assume that 22868 + 8579 = 31447 nodes would infect more than 30% of the graph if they were infected first. Thus, the probability of a large scale epidemic is 31447/85591 ~ 36.7%.

Here - per the TA's suggestion - I am assuming that there is only one, much larger, SCC and so any other SCCs would not be large enough to also infect more than 30% of the graph. Although

this is most often the case, there is a bug in the way the network was created (according to TA) and the next largest SCC is big enough to also infect more than 30% of the graph.

2d)
The worst case scenario is that the entire “bow-tie” is infected, 43766 nodes. However, this is dependent on the fact that every node in the in-component can be reached from the node first infected - a very unlikely scenario.

3a)
i) *Write h(T) in terms of N.*

$$b^{h(T)} = N \Rightarrow log_b N = h(T) \Rightarrow h(T) = log_b N$$

ii) *Given two network nodes (leaf nodes) v and w, let L(v; w) denote the subtree of T rooted at the lowest common ancestor of v and w, and h(v; w) denote its height (that is, h(L(v; w))). In Fig. 1, L(u; t) is the tree in the circle and h(u; t) = 2. For a given node v, what is the maximum possible value of h(v,w)?*

$$max\ h(v,w) = h(T)$$

iii) *Given a value d and a network node v, how many nodes satisfy h(v, w) = d?*

$$(b-1)b^{d-1}\ nodes\ satisfy\ h(v,w) = d$$

3b)
iv) *Show that* $Z \leq log_b N$

$$Z = \sum_{w \neq v} b^{-h(v,w)} = \frac{(b-1)b^0}{2} b^{-1} + \frac{(b-1)b^1}{2} b^{-2} + \ldots + \frac{(b-1)b^{h(T)-1}}{2} b^{-h(T)}$$

v)

$$Given\ h(v,u) = d\ then\ there\ are\ (b-1)b^{d-1}\ nodes\ that\ satisfy\ the\ eq.$$
$$Furthermore, we\ know\ that\ T'\ has\ height\ h(v,u) - 1 = d - 1, so\ there\ are\ (b-1)b^{d-2}\ leaf\ nod$$
$$Thus, we\ can\ formulate\ the\ probability:$$

$$P(e\ pointing\ to\ T') = P(edge\ from\ v\ to\ u \neq v\ AND\ u\ is\ a\ leaf\ in\ T') = \frac{1}{Z}\frac{(b-1)b^{d-2}}{(b-1)b^{d-1}}$$

$$= \frac{1}{Zb} \geq \frac{1}{blog_b N}$$

vi)

$$P(e\ pointing\ to\ T') \geq \frac{1}{blog_b N} \Rightarrow P(e\ not\ pointing\ to\ T') \leq 1 - \frac{1}{blog_b N}$$

$$= (\frac{1}{e})^{c\ log_b N/b} = \frac{1}{e^{c\ log_b N/b}}$$

$$let\ N^\theta = e^{c\ log_b N/b} \Rightarrow \theta logN = \frac{c\ log_b N}{b} \Rightarrow \theta = \frac{c\ log_b N}{b\ logN}$$

$\leq N^{-\theta}$ *where* $\theta = \frac{1}{b \log N}$

The above claim indicates that for any node v, we can, with high probability, find an edge to a (leaf) node u satisfying h(u, t) < h(v, t).

vii)

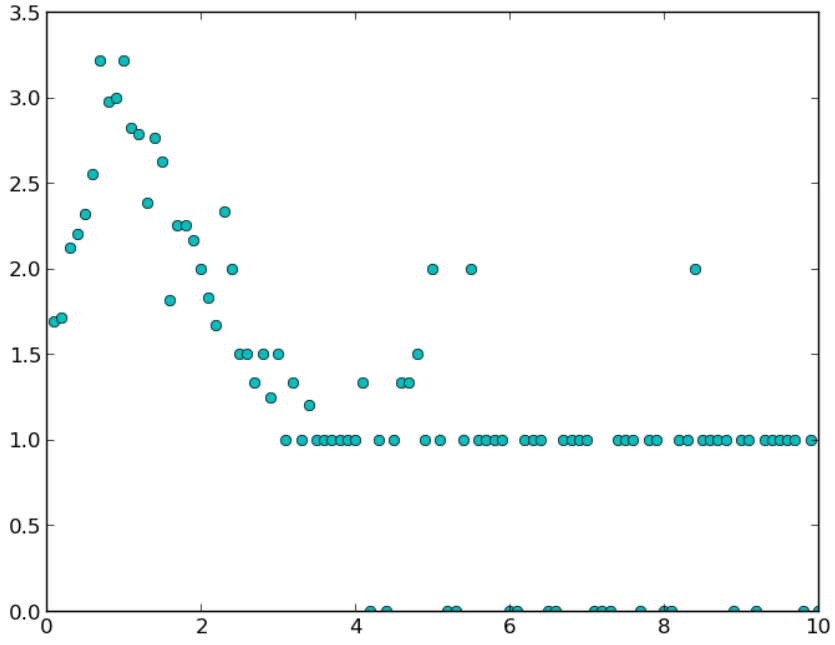
Show that starting from any (leaf) node s, within O(logb N) steps, we can reach any (leaf) node t. You do not need to prove it in a strict probabilistic argument. You can just assume that for any (leaf) node v, you can always get to a (leaf) node u satisfying h(u, t) < h(v, t) and argue why you can reach t in O(logb N) steps.

Suppose our starting node, s, is the first node on the left of the tree and our destination node t is on the right of the t. If at every step in the search we find a node u where *h(u, t) < h(s, t)* , but that *h(u, t)* is only one less than *h(s, t)* then we will have to traverse through $log_b N$ nodes to reach t. This is similar to having to traverse from the root of the tree down to the leaf, which would also take $log_b N$ steps. Thus, as N goes to infinity, the number of steps from a node to another is $log_b N$ because the behaviour of the search becomes similar to that of the example just described. Therefore, the number of steps from any node to another is $O(log_b N)$.

c)

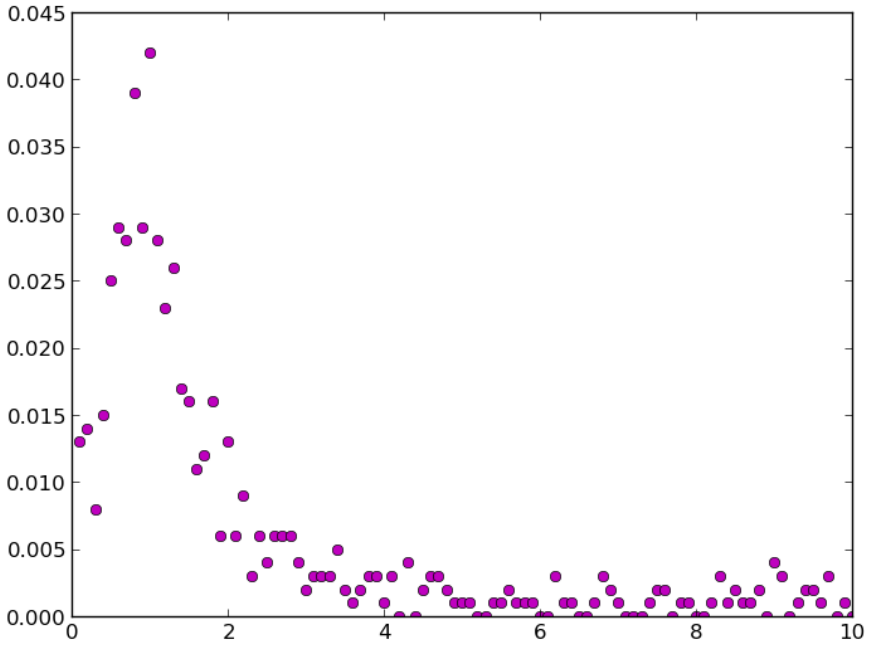
h)

Average Number of Steps from S to T Given Alpha



Note: Avg. Number of Steps on the Y-Axis and Alpha on the X-Axis.
Also note: This plot may be misleading! If average number of steps was zero, that means for 1000 iterations there was no path that succeeded. Important to compare with the next plot.

Probability of Success For Each Given Alpha



Note: Probability of Success on the Y-Axis and Alpha on the X-Axis.

i) In looking at the first plot, we see the average amount of steps is at maximum of about 3.25 average steps at alpha ~ 1 and decreases as alpha increases. At about alpha equal to 2.5, the average number of steps levels out at either 1 or 0. The zero reflects that no searches are succeeding and the 1 reflects that it may have occurred a few times where s and t had an edge connected them, so that the search only took one step. However, the second plot shows that the probability of success in the search appears to maximize around alpha = 1 and significantly decreases as alpha increases past 1. At about alpha = 2.5, the probability of success for almost all trials with larger values of alpha is below .005. This relationship with alpha is representative of the fact that as alpha increases the probability of a node having an edge to a node further away decreases, so if our nodes s and t are further away, with two high of an alpha, there is unlikely a path from s to t. Likewise, if alpha is too low, then the distance h will have very little effect on the creation of an edge and edges will be created more randomly and the likelihood of the search succeeding is also less than if alpha was one.

```
import networkx as nx
from math import *
from random import *
from numpy.random import binomial
import cPickle
import matplotlib.pyplot as plt

hT = 10
b = 2
k = 5
N = int(pow(b, hT))
alphas = []
for i in range(1,101):
  alphas.append(i/10.0)
```

```
#Create mapping of keywords to their paths up to the root.
def get_node_paths(N, hT, b):
  node_paths = {}
  for node in range(N):
    node_paths[node] = []

  for d in range(hT):
    #Calculate the number of nodes under each parent
    n = int(pow(b, d + 1))
    for node in range(N):
      parent = node / n
      node_paths[node].append(parent)
  return node_paths

def h(n1, n2, node_paths):
  path1 = node_paths[n1]
  path2 = node_paths[n2]
  for i in range(len(path1)):
    if(path1[i] == path2[i]):
      return i + 1

def get_z(n1, N, b, alpha, node_paths):
  z = 0
  for n2 in range(N):
    if(n1 != n2):
      z += pow(b, -alpha * h(n1, n2, node_paths))
  return z

def prob_of_edge(n1, n2, b, alpha, z, node_paths):
  return pow(b, -alpha * h(n1,n2, node_paths)) / z

def get_graph(hT, b, k, alpha, N, node_paths):
  G = nx.DiGraph()
  print "Making graph with", N, "nodes..."
  for node in range(N):
    z = get_z(node, N, b, alpha, node_paths) #get z for normalizing prob now
to avoid unnecessary computation
    G.add_node(node)
    print node-1, "nodes have had their edges added"
    num_edges = 0
    while(num_edges < k):
      #select random node
      otherNode = randint(0, N-1)
      if node != otherNode:
        #generate bernoulli trial  with probability p
        p = prob_of_edge(node, otherNode, b, alpha, z, node_paths)
        trial = binomial(1,p)
        if trial == 1: #success! create an edge!
          G.add_edge(node, otherNode)
          num_edges += 1
  return G

def get_graphs(alphas, hT, b, k, N):
  node_paths = get_node_paths(N, hT, b)
  graphs = []
  for alpha in alphas:
```

```
    G = get_graph(hT, b, k, alpha, N, node_paths)
    graphs.append(G)
  cPickle.dump(graphs, open("graphs.pkl", 'wb'))

#Function performs search and reports back 0 steps for failure or number of
steps to success.
def decentralized_search(s,t, G, node_paths):
  steps = 0
  while(True):
    print 'S =', s
    print 'T =', t
    s_neighbors = G.neighbors(s)
    if t in s_neighbors: #know that s != t, but if t in s neighbors, then
search succeeds in one step.
      steps +=1
      return steps

    #in nodes neighbors find next closest node.
    closest = s
    for u in s_neighbors:
      if h(u, t, node_paths) < h(closest,t, node_paths):
        #print "s=", s
        #print "closest=", closest
        #print "h(u,t) = ",  h(u, t, node_paths)
        #print "h(closest,t) = ",  h(closest, t, node_paths)
        closest = u

    if s == closest: #did not find a closer node so failed search.
      steps = 0
      return 0
    else: #found closer node so take step and continue search
      steps += 1
      s = closest

#Function designed to run our test of decentralized search across each graph
made from the different value of alpha.
def test(graphs, alphas, node_paths):
  avg_steps_by_alpha = []
  prob_success_by_alpha = []
  #for each given alpha find avg number of steps and probability of success.
  for i in range(len(graphs)):
    print 'Working on graph', i, '...'
    G = graphs[i]
    alpha = alphas[i]
    steps_list = []
    iter = 0
    while(iter < 1000):
      print 'On iteration', iter,'for graph', i, '.'
      #pick two random nodes:
      s = randint(0, len(G.nodes())-1)
      t = randint(0, len(G.nodes())-1)
      if(s != t): #check nodes different, otherwise try again.
        iter += 1
        steps = decentralized_search(s,t, G, node_paths) #get search results
from s to t
        if steps != 0: #success in search
          steps_list.append(steps)
```

```
    #get avg steps:
    if len(steps_list) != 0:
      avg_steps = sum(steps_list)/float(len(steps_list))
    else:
      avg_steps = 0
    avg_steps_by_alpha.append(avg_steps)

    #get probability of successful search
    p = len(steps_list) / float(1000)
    prob_success_by_alpha.append(p)

  cPickle.dump(avg_steps_by_alpha, open("avg_steps_by_alpha.pkl", 'wb'))
  cPickle.dump(prob_success_by_alpha, open("prob_success_by_alpha", "wb"))
  plt.plot(alphas, avg_steps_by_alpha, 'co')
  plt.savefig('avg_steps_by_alpha.png')
  plt.close()
  plt.plot(alphas, prob_success_by_alpha, 'mo')
  plt.savefig('prob_success_by_alpha.png')

print 'Getting graphs...'
node_paths = get_node_paths(N, hT, b)
graphs = cPickle.load(open('graphs.pkl', 'rb'))

print 'Testing...'
test(graphs, alphas, node_paths)
```