CS 224w: Problem Set 1

Tony Hyun Kim

October 8, 2013

1 Fighting Reticulovirus avarum

1.1 Set of nodes that will be infected

We are assuming that once R. avarum infects a host, it always infects all of the host's contacts. Given an initially infected node v, it follows that the set $\mathrm{Out}(v)$ (i.e. nodes that can be reached from v) will be infected.

1.2 Bow-tie structure of the email network

Basic measurements on the email network:

- Total number of nodes: $85,591$;
- Largest SCC: $22,868$ nodes (26.7% of total nodes);
- In-component of the largest SCC: $8,579$ (10.0%);
- Out-component of the largest SCC: $12,319$ (14.4%);
- Disconnected components (all nodes not part of the above "bow-tie"): $41,825$ (48.9%).

1.3 Probability that a randomly chosen infected node leads to a large-scale epidemic (at least 30% of the graph)

Per the instructions, we focus on the "bow-tie" structure (the SCC core, the in-component and the-outcomponent). There are three possibilities:

- **A node in the SCC core is initially infected.** In this case, all nodes in the SCC will be infected, as well as the out-component. Hence the infection is large-scale.
- **A node in the in-component is initially infected.** In this case, at least one node of the in-component is infected, as well as the entirety of the SCC and the out-component. The infection is large-scale.

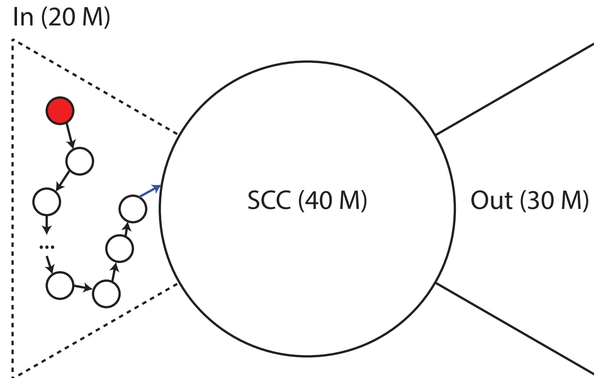

Figure 1: [Left] A "chain" structure for the in-component of the Twitter network. We assume that the in-component consists of a single chain of length 20M that leads into the SCC. The initial node (red) represents the ideal initial infection site for maximum havoc. In this structure, the in-component is "weakly" connected to the SCC and can be disconnected by, for instance, deleting the edge to SCC (blue arrow).

- **A node in the out-component is initially infected.** In this case, at most the entire out-component is infected, whose size is below the "large-scale" criterion (30%). The infection is *not* large-scale.

We ignore potential large-scale outbreaks in the non-bowtie components (tendrils, tubes, disconnected components).

Hence, the probability that a randomly chosen infected node leads to a large-scale epidemic is:

$$(8,579 + 22,868)\,/85,591 = 36.7\%, \tag{1}$$

i.e. the probability that the randomly chosen node lies in the core or the in-component of the largest SCC.

1.4 Twitter bow-tie

1.4.1 Infection in SCC

If a node in the SCC is infected, then the entirety of the SCC and the out-component is infected, leading to $40\text{M} + 30\text{M} = 70\text{M}$ infected nodes (70% of the entire graph).

1.4.2 Worst-case outbreak size

With no assumptions about the network's structure, the worst site for the initial infection is the in-component, since the infection will capture the entirety of the SCC and the out-component.

Now, the fraction of the in-component that will become infected will depend on the structure of the network. The worst case scenario is that a single infection site in the in-component can reach all nodes in the in-component.

A simple network structure (though highly unlikely) that achieves this worst-case infection scenario is if the in-component is a single directed chain (of 20M nodes) that leads into the SCC, as shown in Fig. 1. In this case, the evil TA should target the leading node in the in-component (marked red) for maximum havoc, i.e. infecting the entire bow-tie of 90M nodes.

1.4.3 Reduce the worst-case outbreak size

Given the simple "chain" assumption for the in-component (Fig. 1), it is trivial to reduce the size of the worst-case outbreak. I would remove the single edge (marked in blue) that connects the in-component to the SCC. By removal of this bridge edge, the worst case infection is reduced from 90M to 70M.

```
# Tony Hyun Kim
# 2013 10 03
# CS 224w, PS 1, Problem 1

import snap

source = "email_network.txt"
G = snap.LoadEdgeList(snap.PNGraph, source, 0, 1)
snap.PrintInfo(G)

# Part b: Determine the basic bow-tie structure of the email graph
sccCntV = snap.TIntPrV()
snap.GetSccSzCnt(G, sccCntV)

print "SCC-size count"
for scc in sccCntV:
    print "{} {}".format(scc.GetVal1(), scc.GetVal2()) # (Num nodes in SCC, Num such
 components)

largest_scc = snap.GetMxScc(G)
sz_largest_scc = largest_scc.GetNodes()

random_nid_in_scc = largest_scc.GetRndNId()
print "Random NId in MxSCC: {}".format(random_nid_in_scc)

# Find the out- and in-components
outcomp = snap.GetBfsTree(G, random_nid_in_scc, True,  False)
incomp  = snap.GetBfsTree(G, random_nid_in_scc, False, True)

sz_outcomp = outcomp.GetNodes()
sz_incomp  = incomp.GetNodes()
print "Size of out-component: {}".format(sz_outcomp - sz_largest_scc)
print "Size of  in-component: {}".format(sz_incomp  - sz_largest_scc)
```

2 Network characteristics

2.1 Degree distribution

The (unnormalized) degree distribution is plotted in the left panel of Fig. 2. Some comments about the individual curves:

- $G(n, m)$ random network: the degree distribution is expected to be a binomial distribution with success probability $p \approx 2 \cdot m^2/n$, with mean $\bar{k} = (n-1) \cdot p = 5.5$. In the right panel of Fig. 2, the measurements on the simulated graph is compared against theory (binomial distribution), which shows excellent agreement.
- Small-world network: The salient feature of this distribution is that the distribution is non-zero for $k \geq 4$. Obviously, this follows from the way in which the network is constructed: we began with an ordered network where each node has exactly 4 neighbors, and random long-distance edges were added.
- Real-world collaboration network: Interestingly, the real world graph has a longer tail compared to the previous two examples. It appears that certain authors are extremely prolific and influential (many co-authors).

2.2 Excess degree distribution

2.2.1 Plot of the excess degree distributions

The (unnormalized) excess degree distribution is plotted in the left panel of Fig. 3. While the distributions are not normalized in the figure, it can be seen that the tail of the distribution (high degrees) is more significant in the excess degree distribution relative to the degree distribution. This is so, because the excess degree distribution weights the degrees of nodes that have a large number of neighbors (shown explicitly in Section 2.2.2).

2.2.2 Closed-form formula

We wish to express the excess degree distribution $\{q_k\}$ in terms of the degree distribution $\{p_k\}$.

Consider the unnormalized excess degree distribution q'_k, given by

$$q'_k = \sum_{i \in V} \sum_{(i,j) \in E} I_{[k_j = k+1]} = \sum_{i \in V} \sum_{(i,j) \in E} I_{[k_i = k+1]}. \quad (2)$$

Both double sum expressions in Eq. 2 enumerate the $2 \cdot m$ terms that represent the two terminating nodes of every edge in the graph. It then follows:

$$\begin{aligned} q'_k &= \sum_{i \in V} \sum_{(i,j) \in E} I_{[k_i = k+1]}, & (3) \\ &= \sum_{i \in V} \deg(i) \cdot I_{[k_i = k+1]} = \sum_{i \in V} (k+1) \cdot I_{[k_i = k+1]}, & (4) \\ &= (k+1) \cdot \sum_{i \in V} I_{[k_i = k+1]} = (k+1) \cdot p'_k, & (5) \end{aligned}$$

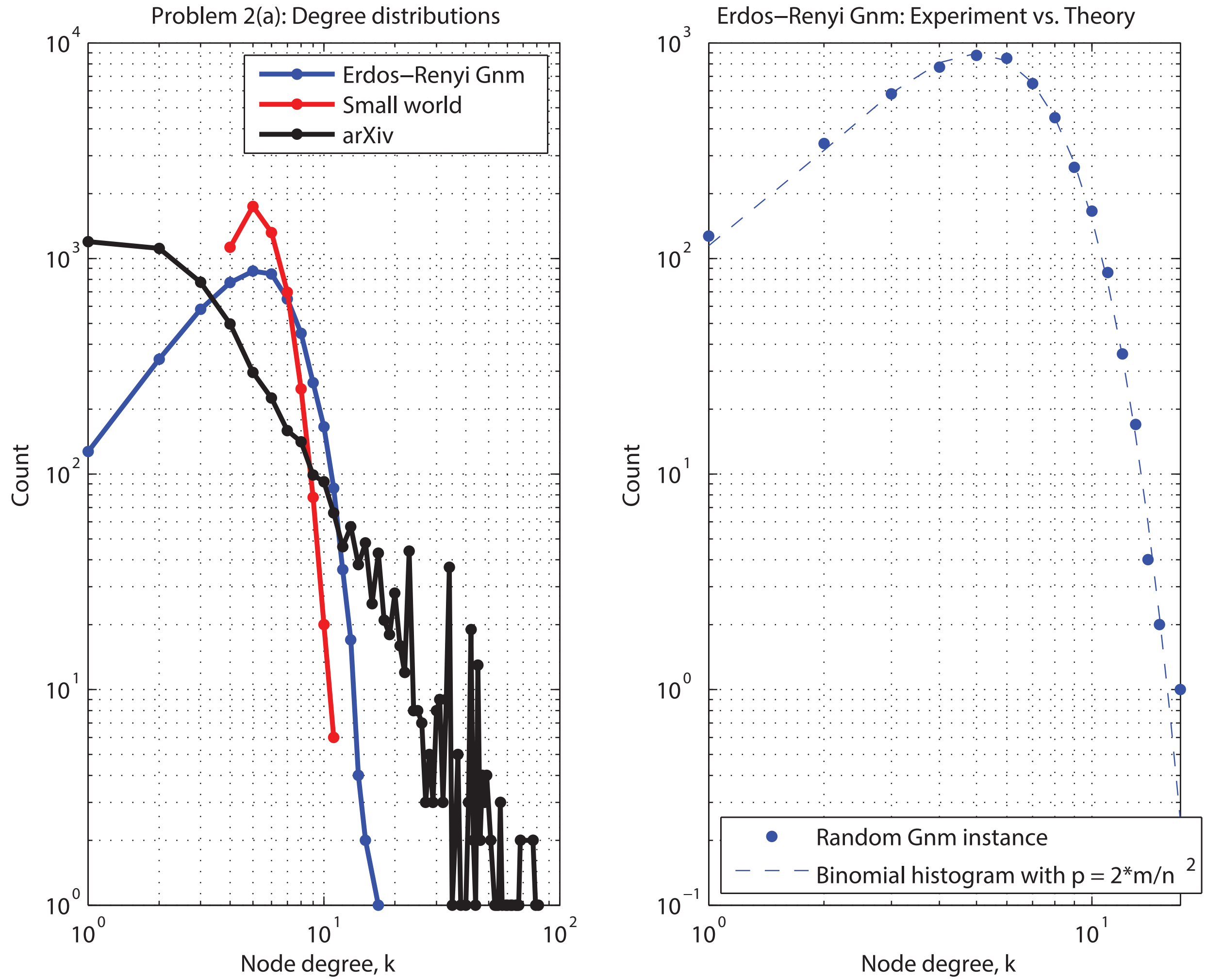

Figure 2: [Left] The degree distributions of $G(n = 5242, m = 14496)$ (blue), small world graph (red) and a real-world collaboration graph (black) on a log-log plot. [Right] Comparison of the observed degree distribution of $G(n, m)$ (dots) to the theoretically expected binomial distribution (dashed line).

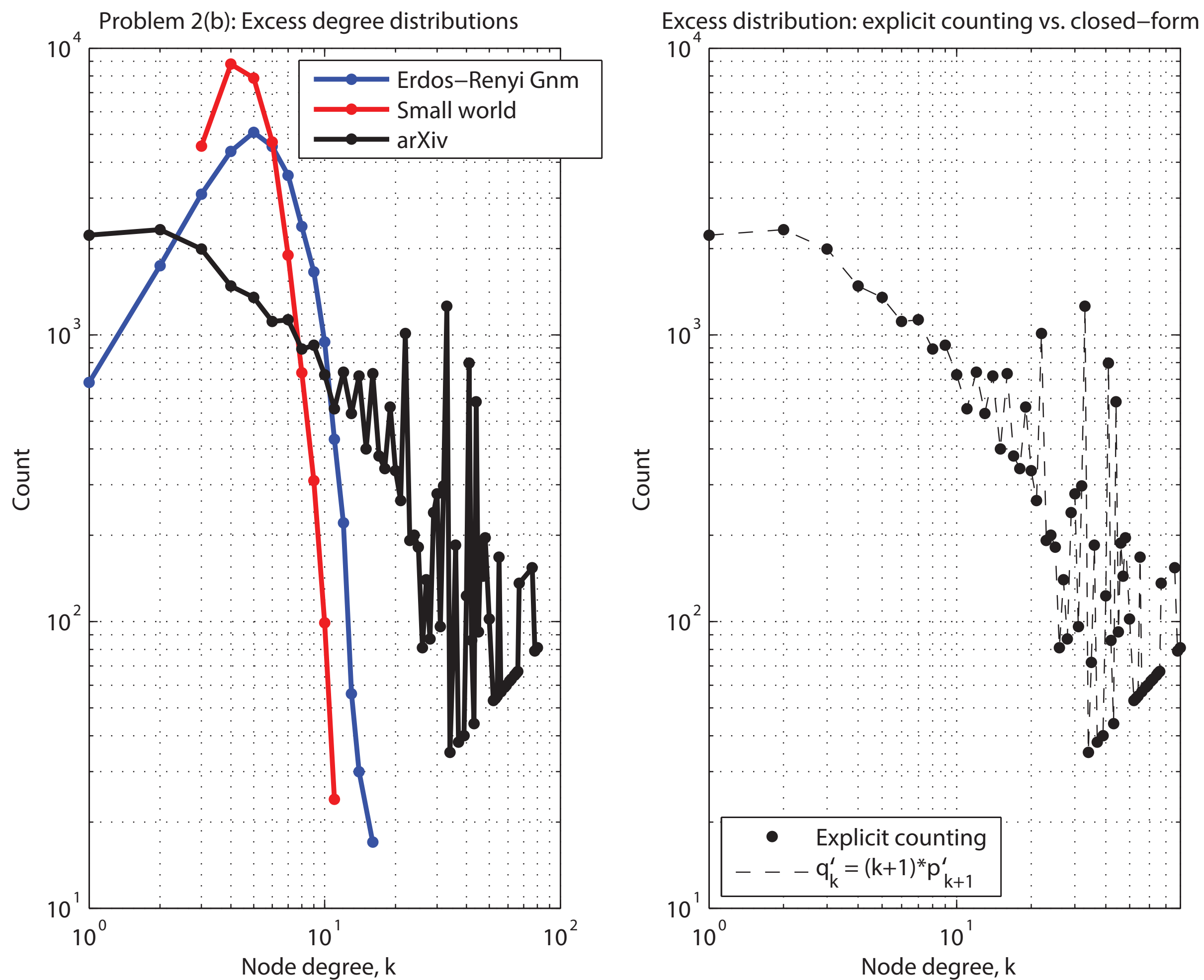

Figure 3: [Left] Excess degree distributions of $G(n = 5242, m = 14496)$ (blue), small world graph (red) and a real-world collaboration graph (black) on a log-log plot. [Right] Comparison of the explicitly calculated excess degree distribution of the arXiv network (dots) to the closed-form expression based on the degree distribution (dashed line).

where in Eq. 4 we have used the property that the indicator function is nonzero only when $\deg(i) = k + 1$. The right panel of Fig. 3 shows a comparison between the explicitly counted excess degree distribution (of the arXiv network) and the closed-form formula of Eq. 5. Using the normalization relations $p_k = 1/n \cdot p'_k$ and $q_k = 1/(2m) \cdot q'_k$, we conclude:

$$q_k = \frac{n}{2m} \cdot (k+1) \cdot p_{k+1}. \tag{6}$$

2.3 Clustering coefficient

My approach for computing the clustering coefficient is as follows. Fix a node i, with at least 2 neighbors. Obtain the induced subgraph consisting of node i and its immediate neighbors. The number of edges in the induced graph minus $\deg(i)$ gives the number of edges e_i between the neighbors of i, and the clustering coefficient for node i can be computed as $C_i = \frac{2 \cdot e_i}{k_i \cdot (k_i - 1)}$.

Note that in computing the average clustering coefficient $\bar{C} = \frac{1}{|V|} \sum_{i \in V} C_i$, I let $|V|$ equal the number of nodes that have degree at least 2, *not* the total number of nodes in the graph.

The calculated clustering coefficients are as follows:

- $G(n,m)$ random network: $\bar{C} = 0.0017$;
- Small-world network: $\bar{C} = 0.2839$;
- Real-world collaboration network: $\bar{C} = 0.3479$.

As we know from our class discussion, the purely random network $G(n,m)$ has very small clustering coefficient. The small world model has a higher clustering coefficient, which owes to the regular connectivity structure of the graph prior to the random long-distance edges. (The small world model with no random edges would have a clustering coefficient of 0.5.) The real-world collaboration network has even a higher clustering coefficient, reflecting our general observation that real-life (social) networks have significant "local structure."

```
# Tony Hyun Kim
# 2013 10 03
# CS 224w, PS 1, Problem 2

import random
import snap

numNodes = 5242
numEdges = 14496

# Purely random (Erdos-Renyi) graph
G1 = snap.GenRndGnm(snap.PUNGraph, numNodes, numEdges)

# Small-world network
G2 = snap.TUNGraph.New()

for nid in range(numNodes):
    G2.AddNode(nid)

for nid in range(numNodes):
    for x in [-2, -1, 1, 2]:
        G2.AddEdge(nid, (nid+x)%numNodes)

numRndEdges = 4012
x = 0 # Counter for number of random edges added
while (x < numRndEdges):
    n1 = random.randint(0, numNodes-1)
    n2 = random.randint(0, numNodes-1)
    if (not G2.IsEdge(n1,n2) and (n1 != n2)): # Optionally block self-loops
        G2.AddEdge(n1,n2)
        x += 1

# Real-world collaboration graph
source = 'ca-GrQc.txt'
G3 = snap.LoadEdgeList(snap.PUNGraph, source, 0, 1)

# Part a: Get the degree distribution
#   Surely, Snap.py has a mechanism for exporting data into a file.
#   In this assignment, I just piped the console output into a file.
#------------------------------------------------------------
G = G2
deg_hist = snap.TIntPrV()
snap.GetDegCnt(G, deg_hist)
'''
for item in deg_hist:
    print "{} {}".format(item.GetVal1(), item.GetVal2())
'''

# Part b: Compute the excess distribution
#   Below, I compute the excess degree distribution by the explicit
#   formula provided in the assignment, even though I was able to
#   derive a closed-form formula for qk in terms of pk
#------------------------------------------------------------
G = G2

# Roundabout way to get the max degree of a graph...
max_degree = (G.GetNI(snap.GetMxDegNId(G))).GetDeg()
excess_deg_hist = [0]*max_degree

# Literal interpretation of the provided definition
'''
for k in range(max_degree): # Fix k
    for ni in G.Nodes():
        for j in ni.GetOutEdges():
            if (G.GetNI(j).GetDeg() == k+1):
                excess_deg_hist[k] += 1
'''

# Slightly more efficient calculation
for ni in G.Nodes():
    for j in ni.GetOutEdges():
        excess_deg_hist[G.GetNI(j).GetDeg()-1] += 1
```



```
'''
for k in range(max_degree):
    if (excess_deg_hist[k] > 0):
        print "{} {}".format(k, excess_deg_hist[k])
'''

# Part c: Compute the clustering coefficient "manually"
#------------------------------------------------------------
G = G2
for ni in G.Nodes():
    ni_neighbors = snap.TIntV()
    ni_neighbors.Add(ni.GetId())
    ki = ni.GetOutDeg()
    if (ki < 2): # Ignore nodes with 0 or 1 neighbor
        break
    for nj_id in ni.GetOutEdges():
        ni_neighbors.Add(nj_id)
    G_ind = snap.GetSubGraph(G, ni_neighbors)
    ei = G_ind.GetEdges() - ki
    print "{} {} {}".format(ni.GetId(), ki, ei)
```

3 Decentralized search

3.1 Basic tree properties

3.1.1 Write $h(T)$ in terms of N

$$h(T) = \log_b(N) \tag{7}$$

3.1.2 Maximum value of $h(v, w)$

$$h(v, w) \leq h(T) = \log_b(N) \tag{8}$$

3.1.3 Number of nodes satisfying $h(v, w) = d$

Let the distance d and a particular network node v fixed. Consider the subtree T_1 whose root node r is the d-th ancestor of v. The subtree T_1 has a total of b^d leaves. Now, among the direct descendants of r, there is one subtree T_2 that contains v as one of its leaves. This subtree T_2 has a total of b^{d-1} leaves. Now, any leaves of T_2 has a tree distance to v of $d-1$ at most. It follows that there are exactly $b^d - b^{d-1}$ nodes that have a tree distance of exactly d with respect to v.

3.2 Network path properties

3.2.1 Show that $Z \leq \log_b(N)$

The partition function Z is defined as a sum over the nodes of the graph:

$$Z = \sum_{w \neq v} b^{-h(v,w)}. \tag{9}$$

We may express the sum in Eq. 9 as a summation over the possible distances from a node v:

$$Z = \sum_{d=1}^{h(T)} (b^d - b^{d-1}) \cdot b^{-d} = \left(1 - \frac{1}{b}\right) \cdot h(T) = \left(1 - \frac{1}{b}\right) \cdot \log_b(N) \leq \log_b(N), \tag{10}$$

where in the first equality we have used the fact that there are exactly $b^d - b^{d-1}$ nodes that have a distance d to a fixed node v.

3.2.2 Probability of edge pointing to T'

By construction, the subtree T' has $b^{h(v,t)-1}$ nodes and any leaf w in T' has distance $h(v, t)$ to the original node v. It follows that the probability of obtaining an edge from v into T' is:

$$p_{e \to T'} = b^{h(v,t)-1} \cdot \frac{1}{Z} b^{-h(v,t)} = \frac{1}{bZ} \geq \frac{1}{b \log_b(N)}. \tag{11}$$

3.2.3 Probability of no edges into T' given k out-degree

Based on the previous result, the probability that a single edge from v does *not* reach T' is:

$$p_{e \not\to T'} = 1 - p_{e \to T'} \leq 1 - \frac{1}{b \log_b(N)}. \tag{12}$$

The probability that $k = c \cdot (\log_b(N))^2$ independent edges all fail to reach T' is then bounded by:

$$p_{e \not\to T'}^k \leq \left(1 - \frac{1}{b \log_b(N)}\right)^k = \left(1 - \frac{1}{b \log_b(N)}\right)^{c \cdot (\log_b(N))^2}. \tag{13}$$

Using the substitution $x = b \log_b(N)$, Eq. 13 can be rewritten as:

$$p_{e \not\to T'}^k \leq \left[\left(1 - \frac{1}{x}\right)^x\right]^{\frac{c}{b} \cdot \log_b(N)}. \tag{14}$$

We take the limit as $N \to \infty$ (equivalently $x \to \infty$):

$$\lim_{x \to \infty} p_{e \not\to T'}^k \leq \lim_{x \to \infty} \left[\left(1 - \frac{1}{x}\right)^x\right]^{\frac{c}{b} \cdot \log_b(N)} = \lim_{N \to \infty} e^{-\frac{c}{b} \cdot \log_b(N)} = \lim_{N \to \infty} N^{-\frac{c}{b}}. \tag{15}$$

3.2.4 Show that starting from any node s, within $O(\log_b N)$ steps, we can reach any node t

Previously we showed that for any initial node v with $k = c \cdot (\log_b(N))^2$ edges, there exists an edge to a node in T' (which contains the target node t) in the limit $N \to \infty$. By taking this edge into T', we reduce the tree distance to t by 1.

For any two nodes s and t, the maximum tree distance $h(s, t)$ is $\log_b N$. At each iteration, by taking the edge that leads into T', we reduce the tree distance by 1. It follows that we can reach the target node t in $O(\log_b N)$ steps.

3.3 Simulation

3.3.1 Navigation simulation in Matlab

I didn't see any particular reason to use Snap for this simulation, so I decided to implement in Matlab. For efficiency, I precomputed the tree distance between all pairs of nodes. Furthermore, I wrote one script to generate the random graphs, and another to perform random searches over the precomputed graphs.

The main results, namely the search success probability and the average path length for successful searches as a function of α, are shown in Fig. 4. In the figure, I also show some representative adjacency matrices as a function of α. (I have ordered the node indexing such that clustering around the diagonal represents local connectivity, whereas the off-diagonal clusters represent long-range links.)

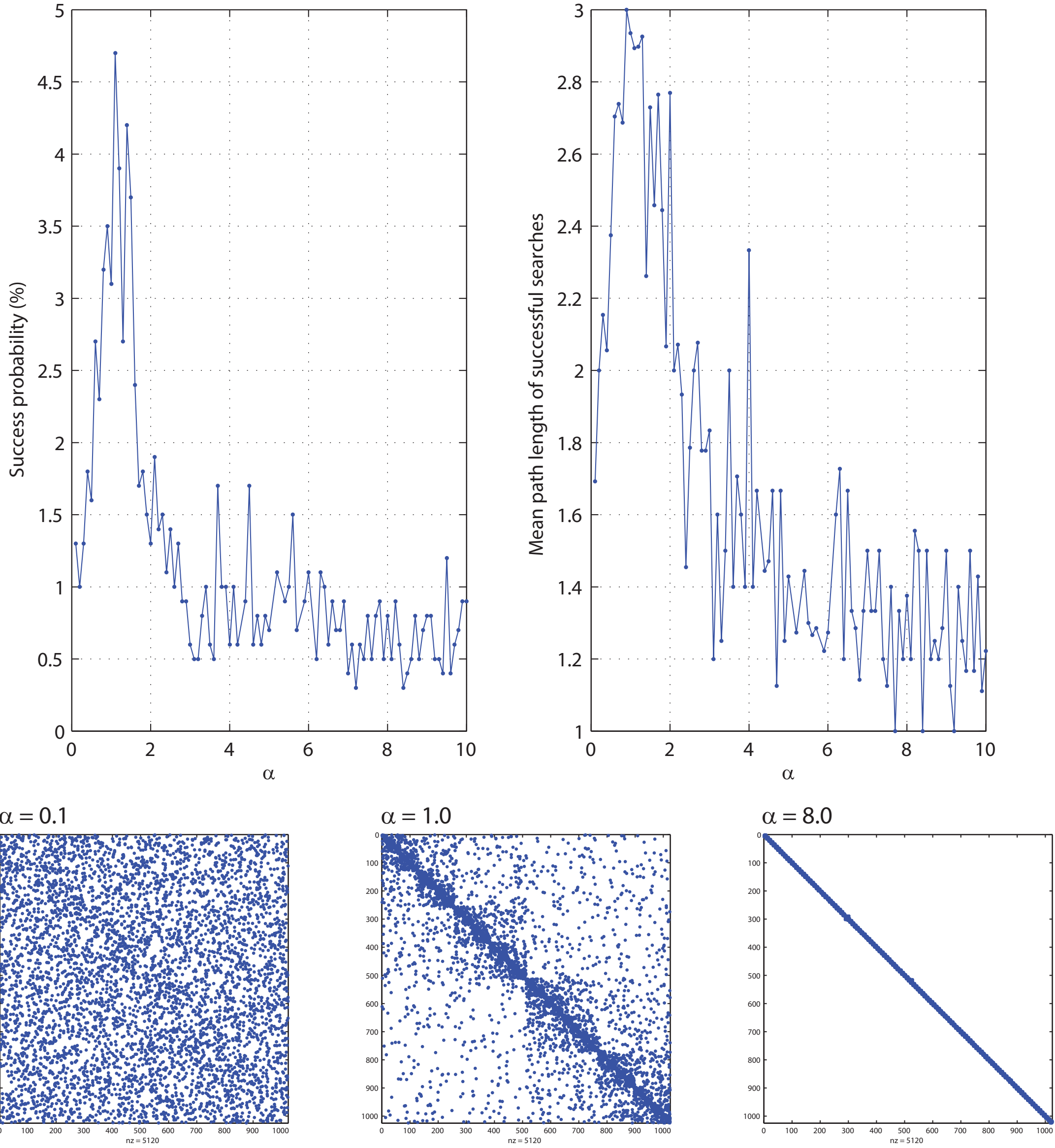

Figure 4: [Top left] The success probability of $N_{\text{search}} = 1000$ random (s, t) searches as a function of α. [Top right] The average path length of successful searches as a function of α. [Bottom] Some representative adjacency matrices. The plots correspond to graphs with $\alpha = 0.1, 1.0, 8.0$. The node indexing is such that clustering around the diagonal represents local (in the tree) connectivity, whereas the off-diagonal entries (roughly) represent long-range links.

3.3.2 Comments on results

Firstly, consider the success probability as a function of α. We find that the success probability peaks for $\alpha \approx 1$. This phenomenon can be understood as follows:

1. Case $\alpha \to 0$: In this limit, the edges in the graph are basically random and do not "respect" the tree organization that underlies the construction of the graph. The randomness is evidenced in the bottom panel of Fig. 4 for $\alpha = 0.1$ where the adjacency matrix has uniformly distributed nonzero entries. So we are trying to search in a random graph, which we showed in lecture is not efficient.

2. Case $\alpha \to \infty$: In this limit, the edges strongly prefer to stay within the local structure as defined by the tree organization, and there are few long-range edges. The strong locality makes it difficult to find paths between arbitrary leaves of the tree.

3. Case $\alpha \approx 1$: In this case, there is both local structure *and* long-range edges to other parts of the tree organization. (See the $\alpha = 1.0$ adjacency matrix in Fig. 4.) So, this case is most like the "small-world network" and is optimally (relatively speaking) searchable.

Secondly, we consider the average path length of successful searches.

1. Case $\alpha \to 0$: In this limit, the edges are basically randomly distributed (ignores the tree organization). In this case, a successful search consists of short paths that, by chance, connect s to t.

2. Case $\alpha \to \infty$: In this limit, successful searches are when s and t are chosen (by chance) to be very near one another in the tree organization. Given the strong local connectivity for such values of α, it becomes likely that s and t are directly connected.

3. Case $\alpha \approx 1$: In this case, the graph has both local structure and long-range links. So, we can obtain nontrivial path lengths by our heuristic, which seeks to decrease the tree distance at each iteration.

```
% Tony Hyun Kim
% 2013 10 08
% CS 224w: PS 1, Problem 3(c):
%   Generate networks to a file as a function of alpha
clear all; close all;

% Tree parameters
h = 10; % Height of tree
b = 2;  % Branching factor
k = 5;  % Degree per node

N = b^h; % Total number of nodes (leaves of tree)
M = k*N; % Total number of edges

% Load the tree distance matrix.
% Note: Once I realized how inefficient my treedist calculation was, I
%       decided to store the distance matrix for the h=10, b=2 case.
load('D_h10_b2.mat');

nodelist = 0:(N-1);    % Use 0-based index
edgelist = zeros(M,2); % Format: [source target]

% Populate the edges according to the given edge probability model
alphas = 0.1:0.1:10;
for alpha = alphas
    for i = 1:N % Select source node
        ni = nodelist(i);

        % Probability distribution to other nodes
        p = b.^(-alpha*D(i,:)); % Use stored distances
        p(i) = 0; % No self loops
        for l = 1:k % Obtain edges
            j = randsample(N, 1, true, p/sum(p));
            edgelist(k*(i-1)+l,:) = [ni nodelist(j)];

            p(j) = 0; % Do not redraw the same edge
        end
    end

    savename = sprintf('alpha%02d_%1d.mat', ...
                       floor(alpha), mod(floor(10*alpha),10));
    save(savename, 'h', 'b', 'k', 'N', 'M',...
                   'alpha', 'nodelist', 'edgelist');
end
```



```
% Tony Hyun Kim
% 2013 10 08
% CS 224w: PS 1, Problem 3(c):
%   Run search simulations on synthesized graphs
clear all; close all;

sources = dir('alpha*.mat');
Nsources = length(sources);

% Need to load the distance metric
load('D_h10_b2.mat');

for f = 1:Nsources
    load(sources(f).name);

    % Perform the random (s,t) searchs
    Nsearch = 1000;

    % Format: [ns nt pathlen]
    search = zeros(Nsearch,3);
    for i = 1:Nsearch
        r = randsample(N, 2, false); % Sample without replacement

        ns = nodelist(r(1)); % Basically conversion for Matlab 1-index
        nt = nodelist(r(2));
        search(i,:) = [ns nt inf];

        len = 0;

        % Set current position u to s
        nu = ns;
        h_ut = D(nu+1, nt+1);
        nu_neighbors = edgelist((1+k*nu):(k+k*nu),2);
        hs = D(nu_neighbors+1, nt+1);
        [min_hs, min_ind] = min(hs);

        while( min_hs < h_ut )
            len = len + 1;

            % Update current position
            nu   = nu_neighbors(min_ind);
            h_ut = min_hs;

            % Did we reach the target? Then terminate
            if (h_ut == 0)
                search(i,3) = len;
                break;
            end

            % Update neighbors
            nu_neighbors = edgelist((1+k*nu):(k+k*nu),2);
            hs = D(nu_neighbors+1, nt+1);
            [min_hs, min_ind] = min(hs);
        end
    end
```

```
    sieve = search(:,3)<inf;
    Nsuccess = sum(sieve);
    avglen = mean(search(logical(sieve),3));
    fprintf('%2.1f %.4f %2.4f\n', alpha, Nsuccess/Nsearch, avglen);
end
```



```
% Compute the tree distance between node u and list of nodes vs
% in a binary tree with height h
function ds = treedist(u, vs, h)

ds = zeros(size(vs));

for j = 1:length(vs)
    ub = dec2bin(u,h);
    vb = dec2bin(vs(j),h);

    d = 0;
    while (~isempty(ub))
        if (all(ub == vb))
            break;
        else
            d = d + 1;
            ub = ub(1:end-1);
            vb = vb(1:end-1);
        end
    end

    ds(j) = d;
end
```