CS 224w: Problem Set 4

Tony Hyun Kim

December 2, 2013

1 Variations on a theme of PageRank

1.1 Personalized PageRanks for E, F, G

The basic intuition is that personalized PageRank should follow superposition in the teleport set S. For instance, the personalized PageRank vector for $S = \{1, 2\}$ should be a superposition (up to normalization) of the PageRank vectors of $\{1\}$ and $\{2\}$ separately. The superposition property makes particular sense given the "random surfer with restarts" interpretation of personalized PageRank, where each instance of the path made by the random surfer is independent of others.

1.1.1 Eloise

Yes. Eloise's PageRank vector is given by:

$$r_E = 3r_A - (3r_B - (3r_C - r_D)) - r_D = 3r_A - 3r_B + 3r_C - 2r_D. \tag{1}$$

1.1.2 Felicity

No. Given the known PageRank vectors, it is not possible to form a linear combination of the corresponding teleport sets to produce the desired set $\{5\}$. Note that the teleport sets that include 5 are Bertha's and Clementine's. However both also have 4 in their teleport sets, which is not possible to eliminate without also eliminating 5.

1.1.3 Glynnis

Yes, Glynnis' PageRank vector is given by:

$$\begin{aligned} r_G &= \frac{1}{10}\left(2 \cdot 3r_A + 1 \cdot 3r_B + 1 \cdot 3r_C - 2 \cdot 1r_D\right), & (2) \\ &= \frac{1}{10}\left(6r_A + 3r_B + 3r_C - 2r_D\right). & (3) \end{aligned}$$

In Eq. 2, I factored each coefficient as $a \cdot b$ where a is the combination coefficient required to construct Glynnis' teleport set from the corresponding known teleport sets, and b is a normalization factor.

1.2 Set of possible personalized PageRank vectors

Given V, we can compute PageRank vectors that correspond to teleport sets that can be formed as a linear combination of the teleport sets associated with V.

1.3 Isolated spam farm

We use the fact that by symmetry of the boosting pages, $p_1 = p_2 = \cdots = p_k$. The PageRank equation for the target page can be written as follows:

$$p_0 = \beta \cdot \sum_{i \to 0} \frac{r_i}{d_i} + (1-\beta) \cdot \frac{1}{N}, \tag{4}$$

$$= \beta \cdot (\lambda + k p_1) + (1-\beta) \cdot \frac{1}{N}, \tag{5}$$

where, in the second line, we have separately accounted for the PageRank flow coming from the "rest of the network" and the boosting pages.

The PageRank equation for the boosting pages can be written as:

$$p_1 = \beta \cdot \frac{p_0}{k} + (1-\beta) \cdot \frac{1}{N}. \tag{6}$$

We then eliminate p_1 from Eq. 5 using Eq. 6:

$$p_0 = \beta \cdot \left[\frac{\lambda + (1-\beta) \cdot \frac{k}{N}}{1-\beta^2} \right] + (1-\beta) \cdot \left[\frac{1}{(1-\beta^2) \cdot N} \right]. \tag{7}$$

1.4 Linked spam farm: Configuration 1

In the new configuration, $p_1 = p_2 = \cdots = p_k = q_1 = q_2 = \cdots = q_m$. In addition, $\bar{p}_0 = \bar{q}_0$.

Applying the PageRank equation, we find:

$$\bar{p}_0 = \bar{q}_0 = \beta \cdot \left[k \cdot \frac{p_1}{2} + m \cdot \frac{q_1}{2} \right] + (1-\beta) \cdot \frac{1}{N}, \tag{8}$$

$$= \beta \cdot \left[(k+m) \cdot \frac{p_1}{2} \right] + (1-\beta) \cdot \frac{1}{N}, \tag{9}$$

$$p_1 = q_1 = \beta \cdot \left[\frac{\bar{p}_0}{k+m} + \frac{\bar{q}_0}{k+m} \right] + (1-\beta) \cdot \frac{1}{N}, \tag{10}$$

$$= \beta \cdot \left[\frac{2 \cdot \bar{p}_0}{k+m} \right] + (1-\beta) \cdot \frac{1}{N}. \tag{11}$$

Solving for $\bar{p}_0$ yields:

$$\bar{p}_0 = \bar{q}_0 = \beta \cdot \left[\frac{\left(\frac{k+m}{2}\right)}{(1+\beta) \cdot N} \right] + (1-\beta) \cdot \left[\frac{1}{(1-\beta^2) \cdot N} \right]. \tag{12}$$

For comparison against the isolated spam farm case, we set$\lambda = 0$ in Eq. 7:

$$p'_0 = \beta \cdot \left[\frac{k}{(1+\beta)\cdot N}\right] + (1-\beta)\cdot\left[\frac{1}{(1-\beta^2)\cdot N}\right], \quad (13)$$

$$q'_0 = \beta \cdot \left[\frac{m}{(1+\beta)\cdot N}\right] + (1-\beta)\cdot\left[\frac{1}{(1-\beta^2)\cdot N}\right]. \quad (14)$$

In then follows that:

$$\bar{p}_0 - p'_0 = \beta \cdot \left[\frac{\left(\frac{-k+m}{2}\right)}{(1+\beta)\cdot N}\right] + (1-\beta)\cdot\left[\frac{1}{(1-\beta^2)\cdot N}\right], \quad (15)$$

$$\bar{q}_0 - q'_0 = \beta \cdot \left[\frac{\left(\frac{k-m}{2}\right)}{(1+\beta)\cdot N}\right] + (1-\beta)\cdot\left[\frac{1}{(1-\beta^2)\cdot N}\right]. \quad (16)$$

Overall, this spam farm configuration simply "averages" the effort of the two independent two farms and does not offer any nonlinear "compounding" gains in the resources used (*i.e.* $\bar{p}_0 + \bar{q}_0 = p'_0 + q'_0$).

1.5 Linked spam farm: Configuration 2

We now consider the case where only the target pages are linked. In this configuration, the boosting pages do not have any in-links, so their PageRank values are $p_1 = \cdots = p_k = q_1 = \cdots = q_m = (1-\beta)\cdot\frac{1}{N}$. The PageRank equations for $\bar{\bar{p}}_0$ and $\bar{\bar{q}}_0$ are:

$$\bar{\bar{p}}_0 = \beta\cdot(k\cdot p_1 + \bar{\bar{q}}_0) + (1-\beta)\cdot\frac{1}{N} = \beta\cdot\left((1-\beta)\cdot\frac{k}{N} + \bar{\bar{q}}_0\right) + (1-\beta)\cdot\frac{1}{N}, \quad (17)$$

$$\bar{\bar{q}}_0 = \beta\cdot(m\cdot q_1 + \bar{\bar{p}}_0) + (1-\beta)\cdot\frac{1}{N} = \beta\cdot\left((1-\beta)\cdot\frac{m}{N} + \bar{\bar{p}}_0\right) + (1-\beta)\cdot\frac{1}{N}. \quad (18)$$

Solving the set of equations for $\bar{\bar{p}}_0$ yields:

$$\bar{\bar{p}}_0 = \beta\cdot\left[\frac{k+\beta m+1}{(1+\beta)\cdot N}\right] + (1-\beta)\cdot\left[\frac{1}{(1-\beta^2)\cdot N}\right]. \quad (19)$$

We can also obtain $\bar{\bar{q}}_0$ by swapping k and m:

$$\bar{\bar{p}}_0 = \beta\cdot\left[\frac{m+\beta k+1}{(1+\beta)\cdot N}\right] + (1-\beta)\cdot\left[\frac{1}{(1-\beta^2)\cdot N}\right] \quad (20)$$

So, it follows that:

$$\bar{\bar{p}}_0 - p'_0 = \beta\cdot\left[\frac{\beta m+1}{(1+\beta)\cdot N}\right], \quad (21)$$

$$\bar{\bar{q}}_0 - q'_0 = \beta\cdot\left[\frac{\beta k+1}{(1+\beta)\cdot N}\right]. \quad (22)$$

In other words, both $\bar{\bar{p}}_0$ and $\bar{\bar{q}}_0$ gain by linking their spam farms in this manner (*i.e.* $\bar{\bar{p}}_0 + \bar{\bar{q}}_0 > p'_0 + q'_0$).

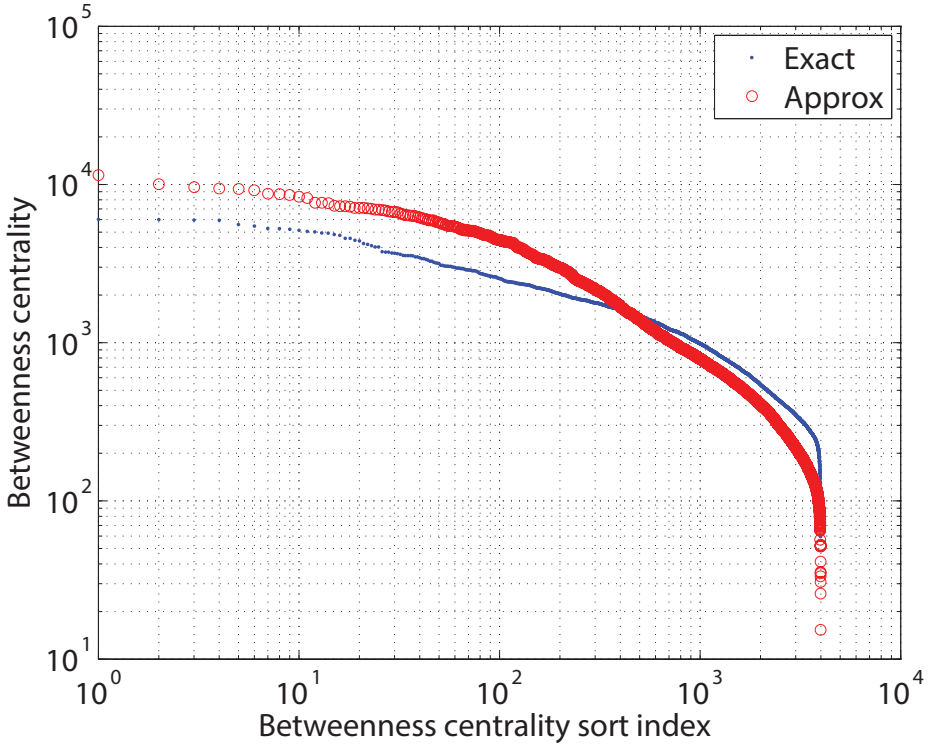

Figure 1: Comparison of the exact and approximate betweenness centrality computation results.

2 Approximate betweenness centrality

The comparison between the exact and approximate betweenness centrality computation results are shown in Fig. 1. The approximate result is plausible, given that we only used 10% of the nodes for the approximation.

On the other hand, the exact ordering of the edges according to decreasing betweenness centrality isn't too great for the approximate method (*i.e.* what is the true rank of the edge with the highest *approximate* betweenness centrality)? This makes sense, since the ordering of edges will be very sensitive to errors in the approximate betweenness centralities.

I attached the exact and approximate codes in the following page. Admittedly, my approximate code is not efficient, it will always make `max_iter` $= n/10$ BFS iterations, even if all Δ_e's are greater than cn. To implement the approximate algorithm correctly, one should properly implement the terminating condition in the loop. However, for the current case, even if I run BFS for every $v \in V$ (*i.e.* the exact algorithm), I found that very few edges attain an aggregate $\Delta_e > cn = 5000$.

```
# Tony Hyun Kim
# CS 224w, PS 4, Problem 2
#-------------------------------------------------------------
from __future__ import division
import snap

# Generate the random graph according to Barabasi-Albert PA model
#-------------------------------------------------------------
n = 1000
G = snap.GenPrefAttach(n, 4, snap.TRnd()) # Undirected graph

# I will use NG to store the betweenness centrality
# Note that the PNEANet type stores directed edges. Since the
#   actual graph of interest (G) is undirected, we will access
#   data in NG where the nodes of an edge are in increasing order
NG = snap.ConvertGraph(snap.PNEANet, G)
NG.AddFltAttrE("betweenCent", 0.0)

# Compute the exact betweenness centrality
#-------------------------------------------------------------
for s in G.Nodes():
    sid = s.GetId()
    #print "sid: {}".format(sid)

    # Obtain the BFS tree as a network, so that we can use attributes
    Ts = snap.ConvertGraph(snap.PNEANet,
                           snap.GetBfsTree(G, sid, True, False))
    Ts.AddIntAttrN("sigma", 0)   # Number of shortest paths
    Ts.AddFltAttrE("delta", 0.0) # Dependency

    # Girvan-Newman algorithm requires us to walk down BFS tree for sigma
    #   climb back up for delta
    fwd_order = [n.GetId() for n in Ts.Nodes()]
    fwd_order.pop(0) # Don't need to go over the source node
    rev_order = fwd_order[::-1]

    # Debug: Enumerate the tree structure
    '''
    for vid in fwd_order:
        v = Ts.GetNI(vid)
        Nout = [nid for nid in v.GetOutEdges()]
        Nin  = [nid for nid in v.GetInEdges()]
        print "vid: {}, {}, {}".format(vid, Nout, Nin)
    '''

    # Perform downward BFS pass on sigma
    Ts.AddIntAttrDatN(sid, 1, "sigma")
    for vid in fwd_order:
        v = Ts.GetNI(vid)
        sigma = 0
        for parent_id in v.GetInEdges():
            sigma += Ts.GetIntAttrDatN(parent_id, "sigma")
        Ts.AddIntAttrDatN(vid, sigma, "sigma")
        #print "vid={}, sig_sv={}".format(vid, sigma)

    # Perform upward BFS pass on delta
    for wid in rev_order:
        w = Ts.GetNI(wid)

        multiplier = 1
        for child_id in w.GetOutEdges():
            multiplier += Ts.GetFltAttrDatE(Ts.GetEId(wid, child_id), "delta")

        for vid in w.GetInEdges():
            delta  = Ts.GetIntAttrDatN(vid,"sigma")/Ts.GetIntAttrDatN(wid,"sigma")
            delta *= multiplier
            Ts.AddFltAttrDatE(Ts.GetEId(vid, wid), delta, "delta")

    '''
    for ei in Ts.Edges():
        print "edge ({},{}), delta={}".format(
                ei.GetSrcNId(), ei.GetDstNId(),
                Ts.GetFltAttrDatE(ei.GetId(), "delta"))
```



```
    '''

    # Accumulate deltas
    for ei in Ts.Edges():
        e = sorted([ei.GetSrcNId(), ei.GetDstNId()])
        e_bc = NG.GetEId(e[0], e[1])
        bc   = NG.GetFltAttrDatE(e_bc, "betweenCent")
        bc  += Ts.GetFltAttrDatE(ei.GetId(), "delta")
        NG.AddFltAttrDatE(e_bc, bc, "betweenCent")

# Display the final betweenness centrality
for ei in G.Edges():
    e = sorted([ei.GetSrcNId(), ei.GetDstNId()])
    e_bc = NG.GetEId(e[0], e[1])
    bc   = NG.GetFltAttrDatE(e_bc, "betweenCent")
    print "{} {} {:.4f}".format(e[0], e[1], bc)
```

```python
# Tony Hyun Kim
# CS 224w, PS 4, Problem 2
#-------------------------------------------------------------
from __future__ import division
import random
import snap

# Generate the random graph according to Barabasi-Albert PA model
#-------------------------------------------------------------
n = 1000
G = snap.GenPrefAttach(n, 4, snap.TRnd()) # Undirected graph

# I will use NG to store the betweenness centrality
# Note that the PNEANet type stores directed edges. Since the
#   actual graph of interest (G) is undirected, we will access
#   data in NG where the nodes of an edge are in increasing order
NG = snap.ConvertGraph(snap.PNEANet, G)
NG.AddFltAttrE("DeltaE", 0.0)
NG.AddIntAttrE("k", 0)

# Parameters for Algorithm 2
c = 5
max_iter = n//10

# Compute the approximate betweenness centrality
#-------------------------------------------------------------
sids = random.sample(xrange(n), max_iter)
for sid in sids:
    s = G.GetNI(sid)

    # Obtain the BFS tree as a network, so that we can use attributes
    Ts = snap.ConvertGraph(snap.PNEANet,
                           snap.GetBfsTree(G, sid, True, False))
    Ts.AddIntAttrN("sigma", 0)   # Number of shortest paths
    Ts.AddFltAttrE("delta", 0.0) # Dependency

    # Girvan-Newman algorithm requires us to walk down BFS tree for sigma
    #   climb back up for delta
    fwd_order = [x.GetId() for x in Ts.Nodes()]
    fwd_order.pop(0) # Don't need to go over the source node
    rev_order = fwd_order[::-1]

    # Perform downward BFS pass on sigma
    Ts.AddIntAttrDatN(sid, 1, "sigma")
    for vid in fwd_order:
        v = Ts.GetNI(vid)
        sigma = 0
        for parent_id in v.GetInEdges():
            sigma += Ts.GetIntAttrDatN(parent_id, "sigma")
        Ts.AddIntAttrDatN(vid, sigma, "sigma")

    # Perform upward BFS pass on delta
    for wid in rev_order:
        w = Ts.GetNI(wid)

        multiplier = 1
        for child_id in w.GetOutEdges():
            multiplier += Ts.GetFltAttrDatE(Ts.GetEId(wid, child_id), "delta")

        for vid in w.GetInEdges():
            delta  = Ts.GetIntAttrDatN(vid,"sigma")/Ts.GetIntAttrDatN(wid,"sigma")
            delta *= multiplier
            Ts.AddFltAttrDatE(Ts.GetEId(vid, wid), delta, "delta")

    # Accumulate deltas
    for ei in G.Edges():
        e = sorted([ei.GetSrcNId(), ei.GetDstNId()])
        e_bc = NG.GetEId(e[0], e[1])
        De   = NG.GetFltAttrDatE(e_bc, "DeltaE")
        if (De < c*n): # Accumulate only when De < c*n
            k = NG.GetIntAttrDatE(e_bc, "k")
            NG.AddIntAttrDatE(e_bc, k+1, "k")
```



```python
            e_ts = Ts.GetEId(e[0], e[1])
            if (e_ts < 0): # The tree Ts is directed
                e_ts = Ts.GetEId(e[1], e[0])
            if (e_ts < 0): # It may be the case that the edge does not occur in Ts
                continue
            De += Ts.GetFltAttrDatE(e_ts, "delta")
            NG.AddFltAttrDatE(e_bc, De, "DeltaE")


# Display the final betweenness centrality
for ei in G.Edges():
    e = sorted([ei.GetSrcNId(), ei.GetDstNId()])
    e_bc = NG.GetEId(e[0], e[1])
    De   = NG.GetFltAttrDatE(e_bc, "DeltaE")
    k    = NG.GetIntAttrDatE(e_bc, "k")
    print "{} {} {:.4f} {}".format(e[0], e[1], De, k)
```

```
% Tony Hyun Kim
% CS 224w, PS 4, Problem 2
% Display the results of betweenness centrality computation
clear all; close all;

fs = 18;

n = 1000;
source1 = 'p2_alg1.txt';
data1 = load(source1);
[betweenCent1, sind1] = sort(data1(:,3),'descend');
data1 = data1(sind1,:);
loglog(betweenCent1,'.');
hold on;

source2 = 'p2_alg2.txt';
data2 = load(source2);
betweenCent2 = n*data2(:,3)./data2(:,4);
[betweenCent2, sind2] = sort(betweenCent2,'descend');
data2 = data2(sind2,:);
loglog(betweenCent2,'ro');
xlabel('Betweenness centrality sort index','FontSize',fs);
ylabel('Betweenness centrality','FontSize',fs);
grid on;
legend('Exact','Approx',...
       'Location','NorthEast');
set(gca,'FontSize',fs);
```

$$K_2 = \begin{array}{c} \begin{array}{cccc} 11 & 10 & 01 & 00 \\ \downarrow & \downarrow & \downarrow & \downarrow \end{array} \\ \left[\begin{array}{cccc} \alpha\cdot\alpha & \alpha\cdot\beta & \beta\cdot\alpha & \beta\cdot\beta \\ \alpha\cdot\beta & \alpha\cdot\gamma & \beta\cdot\beta & \beta\cdot\gamma \\ \beta\cdot\alpha & \beta\cdot\beta & \gamma\cdot\alpha & \gamma\cdot\beta \\ \beta\cdot\beta & \beta\cdot\gamma & \gamma\cdot\beta & \gamma\cdot\gamma \end{array}\right] \end{array} \begin{array}{l} \\ \longleftarrow 11 \\ \longleftarrow 10 \\ \longleftarrow 01 \\ \longleftarrow 00 \end{array}$$

Figure 2: Indexing scheme (decreasing binary counter) that allows Θ_k to be represented as a product of Θ_1, here illustrated for $k = 2$.

3 Stochastic Kronecker graphs

3.1 Indexing scheme

In order to use the desired factorization $\Theta_k(u_1u_2\cdots u_k, v_1v_2\cdots v_k) = \prod_{i=1}^{k}\Theta_1(u_i, v_i)$, the indexing should be a decrementing k-bit counter (*i.e.* decreasing index in columns from left to right, decreasing index in row from top to bottom). This indexing is explicitly illustrated for $k = 2$ in Fig. 2.

3.2 Compute $P[u, v]$

Given the results of the previous part, we are able to compute $P[u, v]$ by considering each of the k bit pairs (u_i, v_i). Since we are told

- the weight of node u is l (*i.e.* the number of 1's in the binary representation of u),
- i is the number of bits where $u_b = v_b = 1$,
- j is the number of bits where $u_b = 0$ and $v_b = 1$,

the frequency of each bit pair combination is enumerated in Table 1. Based on this table, we conclude that:

$$P[u, v] = \alpha^i \cdot \beta^{(l-i)+j} \cdot \gamma^{(k-l)-j}. \tag{23}$$

u_b	v_b	$\Theta_1(u_b, v_B)$	Frequency
1	1	α	i
1	0	β	$l - i$
0	1	β	j
0	0	γ	$(k - l) - j$

Table 1: Frequency of each bit pair combination

3.3 Expected degree of node u with weight l represented using k-bits

The expected degree of node u with weight l represented using k bits is (I think):

$$(\alpha+\beta)^l \cdot (\beta+\gamma)^{k-l}. \tag{24}$$

I observed the above pattern in the case for $k = 2$. Consider again Fig. 2. The expected degree of any particular node is the column sum of the adjacency matrix for that node, since each entry in the matrix is the probability of forming a single edge. In the case of $k = 2$, I observed that the column sums are:

- $l = 2$ (first column): The column sum is $\alpha^2 + 2\alpha\beta + \beta^2 = (\alpha+\beta)^2$,
- $l = 1$ (second and third columns): The column sum is $\alpha\beta + \alpha\gamma + \beta\beta + \beta\gamma = (\alpha+\beta)\cdot(\beta+\gamma)$,
- $l = 0$ (last column): The column sum is $\beta^2 + 2\beta\gamma + \gamma^2 = (\beta+\gamma)^2$.

I then make a leap of faith for $k \geq 3$.

3.4 Expected number of edges in the graph

We assume that the graph is undirected. In this case, we can obtain the expected number of edges $\bar{M}$ in the graph by computing the sum of the expected degrees of each node and dividing by two. (We should in principle exempt self-edges from the divide-by-two, but I'll ignore this complication as instructed.)

The sum of the expected degrees of each node is the aggregate sum of all entries in the adjacency graph Θ_k. We have previously computed the column sums in Eq. 24. Now, noting that there are k choose l nodes with weight l (the possible arrangements of l $1's$ in a k-bit binary sequence), we have:

$$\begin{aligned}
\bar{M} &= \frac{1}{2}\cdot\sum_{l=0}^{k}\binom{k}{l}\cdot(\alpha+\beta)^l\cdot(\beta+\gamma)^{k-l}, &(25)\\
&= \frac{1}{2}\cdot(\alpha+2\beta+\gamma)^k, &(26)
\end{aligned}$$

as the expected number of edges in the graph.

3.5 Expected number of self loops

The expected number of self loops is the sum over the main diagonal of the adjacency graph.

Consider a node with weight l. The probability it forms a self-edge is then given by $\alpha^l \cdot \gamma^{k-l}$. Again, taking the combinatorial multiplicity of the nodes with weight l into account, we have:

$$\begin{aligned}
\bar{M}_{\text{self}} &= \sum_{l=0}^{k}\binom{k}{l}\cdot\alpha^l\cdot\gamma^{k-l}, &(27)\\
&= (\alpha+\gamma)^k &(28)
\end{aligned}$$

as the expected number of self-loops.

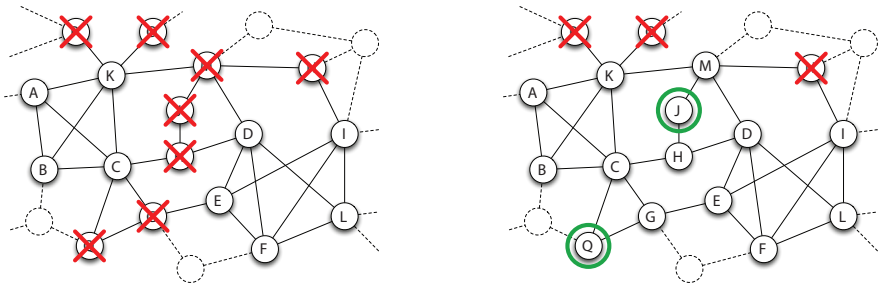

Figure 3: [Left] The equilibrium network when each node requires $k = 3$ neighbors in order to stay in the graph. [Right] The equilibrium network when nodes J and Q are "brainwashed" to stay in the network.

4 Anchored k-cores in social networks

4.1 The equilibrium network

I pruned the provided network for the 3-core. The equilibrium network includes nodes A, B, C, K; D, E, F, I, L, as shown in the left panel of Fig. 3.

4.2 Finding the k-core

The algorithm is given in Alg. 1 below.

4.3 Saving nodes

In the provided "Facebook" network, I would save nodes J and Q. This case is illustrated in the right panel of Fig. 3. With this intervention, we save 5 additional nodes beyond the original case.

Algorithm 1: Find the k-core of a graph $G = (V, E)$

```
V_0 ← set of nodes in G;
V_1 ← ∅;
while 1 do
    G_1 ← InducedGraph(G, V_0);
    V_1 ← {v ∈ G_1 | deg_{G_1}(v) ≥ k};
    if V_0 = V_1 then
        break;
    end
    V_0 ← V_1;
end
return V_1
```

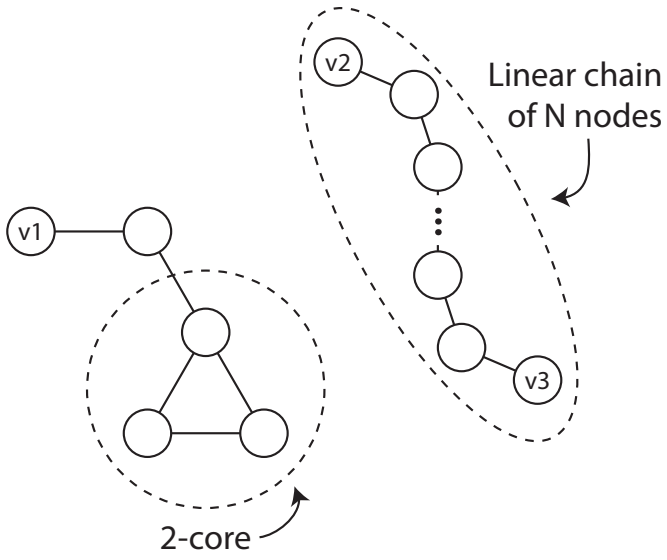

Figure 4: An example of a network for which the naive greedy algorithm for anchored k-core will fail arbitrarily badly (for $k = 2$, $b = 2$).

4.4 Failure of naive greedy

Indeed, the anchored k-core problem is greatly simplified for $k = 2$... I will demonstrate a network structure in which the naive greedy algorithm for anchored k-core will performs arbitrarily badly.

Consider the network in Fig. 4. The network consists of two disjointed parts. One of the disjointed parts is a single triad with a short, dangling segment. The other part is a linear chain of N nodes, where N is assumed to be very large.

Note that for $k = 2$, linear chains are not stable structures. For instance, the short, dangling segment connected to the triad will iteratively be eliminated (starting from v_1). Likewise, the long linear chain will also decay from the edges (starting from v_2 and v_3). In the basic network, only the triad persists in the equilibrium network.

In the network of Fig. 4, the optimal solution for anchored k-core is to select v_2 and v_3 which, when taken together, will prevent the long linear chain from being eliminated. The optimal solution will save the N nodes involved in the linear chain.

On the other hand, the naive algorithm that only considers one anchor at a time, will initially select v_1, since that choice will provide an incremental improvement of two nodes (v_1 and its neighbor) compared to an incremental improvement of one node (for any other choice). In the second iteration, naive would then choose some random node in the long linear chain. Unfortunately, for $k = 2$, you can't save a chain without capping both ends!

So, with $b = 2$, the optimal solution would have saved N nodes, whereas the greedy algorithm only saves 3 nodes. Hence, as N is made large, the naive method fails arbitrarily badly.

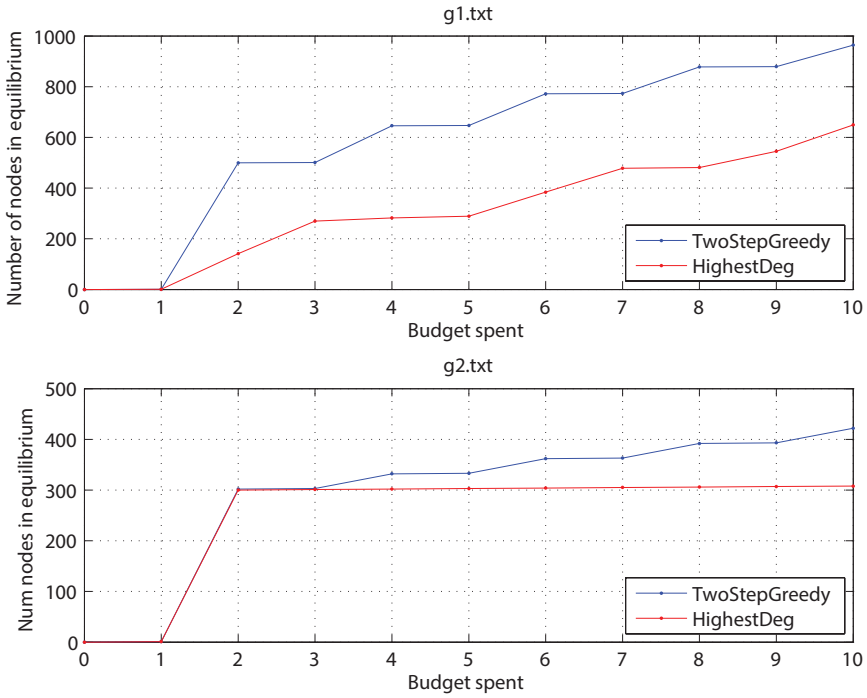

Figure 5: The performance of `TwoStepGreedy` (blue) and `HighestDeg` algorithms on `g1.txt` and `g2.txt`.

4.5 RemoveCore

Note that nodes in the k-core (C_k) of the unanchored network will always be in the equilibrium network when anchors are added. Hence, we do not need to explicitly consider the elements of C_k when searching for the optimal anchor set.

Instead, we can perform the k-core problem for G' (where edges in C_k are removed) where nodes in C_k are considered to be anchored for "free" (*i.e.* without using up the budget). In effect, we are optimizing the incremental saves beyond the baseline case, rather than the absolute number of nodes in equilibrium.

4.6 TwoStepGreedy

Nothing.

4.7 Data exploration

My implementation of `TwoStepGreedy` and `HighestDegree` is attached at the end of this submission.

Fig. 5 shows the performance of `TwoStepGreedy` and `HighestDegree` algorithms for anchored $K = 2$-core on `g1.txt` and `g2.txt`. It is interesting that the `HighestDeg` algorithm is stalled on `g2.txt` for $b \geq 2$ (even though there is room for improvement, as illustrated by the results of `TwoStepGreedy`).

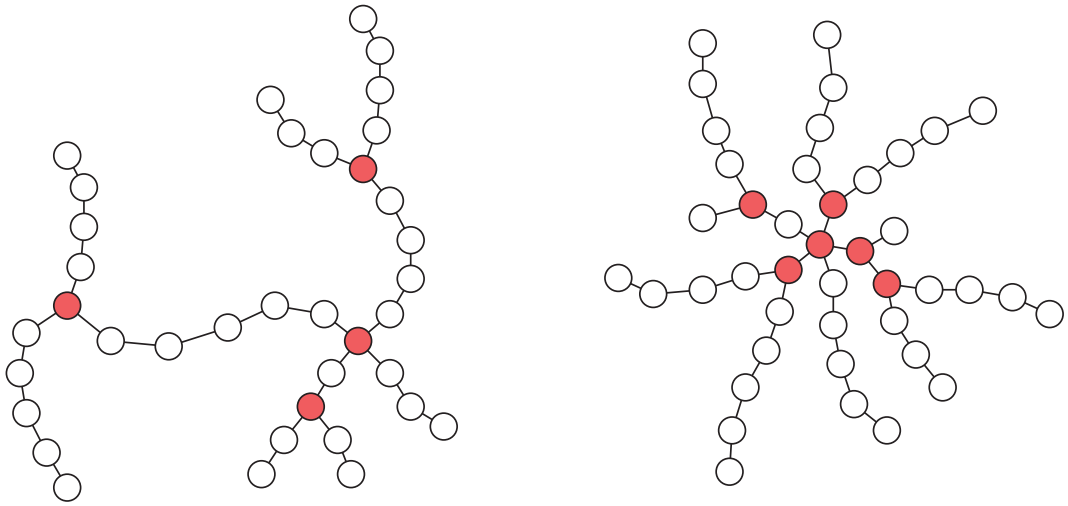

Figure 6: Two network structures in which the `HighestDeg` algorithm will perform well (left) and poorly (right). The salient difference is the average distances between the nodes of high degree (highlighted in red).

4.8 Possible structures for G_1 and G_2

A notable feature of the K-core problem for $K = 2$ is that we can save linear chains by anchoring the two endpoints. In `HighestDeg` algorithm, we are "hoping" that the nodes with large degrees are likely to be endpoints of chains that may be saved.

In Fig. 6, I show two structures where the `HighestDeg` algorithm will perform well on one (save many nodes in addition to the anchor), and poorly on the other. The distinction between two network structures is the average distance between high-degree branching points.

The graph in the left panel of Fig. 6 has relatively large distances between the nodes of high degree (marked in red) that are likely to be picked by the `HighestDeg` algorithm. In this case, the degree of a node is a decent proxy for effective anchors, since the nontrivial chains between the selected (red) nodes will also be saved. I suspect that `g1.txt` may be of this structure.

In contrast, the graph in the right panel has very short distances between the nodes of high degree. In this case, high-degree nodes are not effective anchors. I suspect that `g2.txt` may have such a structure, which is why the performance is capped at $b \geq 2$ (Fig. 5).

```
# Tony Hyun Kim
# CS 224w, PS 4, Problem 4.7
# Implementation of TwoStepGreedy
#-------------------------------------------------------------
from __future__ import division

import snap
import sys

# Some support functions
#-------------------------------------------------------------
def GetGraphCopy(G):
    G2 = snap.TUNGraph.New()
    for n in G.Nodes():
        G2.AddNode(n.GetId())
    for e in G.Edges():
        G2.AddEdge(e.GetSrcNId(), e.GetDstNId())
    return G2

def RemoveCore(G, C2, A, S):
    '''
    G  (snap.TUNGraph)
    C2: List of node IDs in the unanchored 2-core
    A:  List of node IDs of the anchored set
    S:  List of node IDs of the saved set
    '''
    N = []
    N.extend(C2)
    N.extend(A)
    N.extend(S)

    for u in N:
        for v in N:
            if (u != v):
                G.DelEdge(u,v)

def FindLongestBranchInTree(GT, root_id):
    '''
    GT (snap.TNGraph): Directed tree
    root_id (int): ID of the root node in tree

    Returns:
        branch: List of node ids in the longest branch
    '''
    # First, compute the maximum hop in the tree
    HopCntV = snap.TIntPrV()
    snap.GetNodesAtHops(GT, root_id, HopCntV, True)

    max_hop = -1
    for HopCnt in HopCntV:
        hop = HopCnt.GetVal1()
        if (hop > max_hop):
            max_hop = hop

    # Second, find the leaf node in the longest branch
    leaves = snap.TIntV()
    snap.GetNodesAtHop(GT, root_id, max_hop, leaves, True)

    # Finally, enumerate the branch in reverse order, i.e.
    #   from the leaf back to the root. Note that for k=2 core
    #   the nodes in the BFS tree cannot have more than one
    #   parent (otherwise we have a contradiction)
    branch = []

    nid = leaves[0]
    while (nid != root_id):
        branch.append(nid)
        n = GT.GetNI(nid)
        nid = n.GetInEdges().next()

    return branch

# Load the graph
```



```
#-------------------------------------------------------------
#source = "g1.txt"
source = sys.argv[1]
G = snap.LoadEdgeList(snap.PUNGraph, source, 0, 1)
#print "Loaded graph '{}'".format(source)

# Precompute the 2-core of G
G_C2 = snap.GetKCore(G, 2)
C2 = []
for n in G_C2.Nodes():
    C2.append(n.GetId())
#print "The 2-core of unanchored graph has {} nodes".format(len(C2))

# Maximum budget to consider
b_max = int(sys.argv[2])
b     = b_max

A = [] # List of anchored nodes
S = [] # List of nodes saved by anchored nodes

# Implementation of TwoStepGreedy!
B1 = [] # Best (longest) branch
B2 = [] # Second-best branch

while b > 0:
    #print "* Remaining budget: b={}".format(b)
    G2 = GetGraphCopy(G)
    RemoveCore(G2, C2, A, S)

    # Loop over the trees in R (roots are nodes in C2). We are
    #   looking for the longest branches in the tree
    #-----------------------------------------------------------
    for root_id in C2:
        R = snap.GetBfsTree(G2, root_id, True, False)

        # We will remove the elements of R from G2. What remains
        #   in G2 is T
        for n in R.Nodes():
            G2.DelNode(n.GetId())

        B = FindLongestBranchInTree(R, root_id)
        if (len(B) > len(B1)):
            # If we found the best branch in the current tree, also
            #   need to check if the second-best branch is in the
            #   same tree
            B2 = B1
            B1 = B

            # Need to delete the previously identified branch, up to
            #   the nearest branching point (since a branch point
            #   represents another "chain" that we might save)
            for n_id in B1:
                n = R.GetNI(n_id)
                children = list(n.GetOutEdges())
                # Note that we are destroying the branch from leaf on
                #   up. Hence, a single strand should always have
                #   len(children) == 0
                if (len(children) > 0):
                    break
                R.DelNode(n_id)

            B = FindLongestBranchInTree(R, root_id)
            B = [nid for nid in B if nid not in B1] # Don't double count
            if (len(B) > len(B2)):
                B2 = B
        elif (len(B) > len(B2)):
            B2 = B
    #print "B1: {}".format(B1)
    #print "B2: {}".format(B2)

    # If we did not find any interesting nodes to save, then just
    #   pick an node at random (makes sense to take nodes not in
    #   C2, A, or S. I'll proceed with blacklisting A only, per the
```

```
    #   problem instructions
    if (len(B1)==0):
        #print "Used random selection for B1"
        blacklist = []
        #blacklist.extend(C2)
        blacklist.extend(A)
        #blacklist.extend(S)

        rnd_id = G.GetRndNId()
        while (rnd_id in blacklist):
            rnd_id = G.GetRndNId()

        B1 = [rnd_id]

    # Next, we consider the trees in T (G2). First, we segment the
    #   graph into connected components. In each component, we
    #   need to look for the maximal chain that we can save by
    #   placing two anchors at the chain endpoints.
    #------------------------------------------------------------
    B3 = []

    wccs = snap.TCnComV()
    snap.GetWccs(G2, wccs)
    for wcc in wccs:
        # We begin by forming a tree with a randomly chosen node
        #   in the wcc
        root_id = wcc.GetRndNId()
        T = snap.GetBfsTree(G2, root_id, True, False)
        B = FindLongestBranchInTree(T, root_id)

        # We need to regenerate the tree from the leaf of the
        #   deepest branch in order to guarantee we have the
        #   longest possible chain
        if (len(B) > 0):
            root_id = B[0]
            T = snap.GetBfsTree(G2, root_id, True, False)
            B = FindLongestBranchInTree(T, root_id)
            B.append(root_id) # Note that B includes both endpoint nodes

            if (len(B) > len(B3)):
                B3 = B
    #print "B3: {}".format(B3)

    # Finally, we decide for the best option
    if ((len(B1)+len(B2) > len(B3)) or (b==1)): # Choose 1 solution
        A.append(B1[0])
        S.extend(B1[1:])
        b = b-1

        # Bump up the second possible path
        B1 = B2
        B2 = []
    else:
        A.append(B3[0])
        A.append(B3[-1])
        S.extend(B3[1:-1])
        b = b-2

    #print "A: {}".format(A)
    #print "S: {}".format(S)

# Compute the final results
equilibrium_set = []
equilibrium_set.extend(C2)
equilibrium_set.extend(A)
equilibrium_set.extend(S)

'''
print "* Final results for b={}".format(b_max)
print "C2: {}".format(C2)
print "A: {}".format(A)
print "S: {}".format(S)
print "Equilibrium set (size {}): {}".format(len(equilibrium_set), equilibrium_set)
```



```
'''
print "{}, {}".format(b_max, len(equilibrium_set))
```

```
# Tony Hyun Kim
# CS 224w, PS 4, Problem 4.7
# Implementation of HighestDegree
#-------------------------------------------------------------
from __future__ import division
import snap
import sys

# Some support functions
#-------------------------------------------------------------
def GetGraphCopy(G):
    G2 = snap.TUNGraph.New()
    for n in G.Nodes():
        G2.AddNode(n.GetId())
    for e in G.Edges():
        G2.AddEdge(e.GetSrcNId(), e.GetDstNId())
    return G2

def Ints2TIntV(vals):
    ret = snap.TIntV()
    for val in vals:
        ret.Add(val)
    return ret

def GetAnchoredKCore(G,A,K):
    '''
    Compute the anchored K-core by my algorithm
        given in Prob 4.3
    '''
    V0 = set()
    for n in G.Nodes():
        V0.add(n.GetId())

    while True:
        G1 = snap.GetSubGraph(G, Ints2TIntV(V0))

        V1 = set()
        for anchor in A:
            V1.add(anchor) # Always preserve the anchors

        for n in G1.Nodes():
            if (n.GetOutDeg() >= K):
                V1.add(n.GetId())

        if (V1.issubset(V0) & V0.issubset(V1)): # Set equivalence
            break

        V0 = V1

    return V1

# Load the graph
#-------------------------------------------------------------
#source = "toy3.txt"
source = sys.argv[1]
G = snap.LoadEdgeList(snap.PUNGraph, source, 0, 1)


# Maximum budget to consider
b_max = int(sys.argv[2])
b     = b_max

K = 2
A = set()            # Set of anchored nodes
Gs = GetGraphCopy(G) # "Scratch" graph

V1 = GetAnchoredKCore(G,A,K) # V1 is the equilibrium set
print "{}, {}".format(0, len(V1))

while b > 0:
    # Grab the node with the highest degree, but don't add
    #   nodes already in the equilibrium set, since that
    #   would be basically throwing away the budget
```



```
    n_id = snap.GetMxOutDegNId(Gs)
    if (not (n_id in V1)):
        A.add(n_id)
        b = b-1

        V1 = GetAnchoredKCore(G,A,K)
        #print "A={}".format(A)
        print "{}, {}".format(b_max-b, len(V1))

    Gs.DelNode(n_id)
```