CS 224w: Problem Set 2

Tony Hyun Kim

October 23, 2013

1 Chief social engineer of the world

1.1 Maximizing my network

At best, I can reach $10^1 + 10^2 + \cdots + 10^m$ people in m hops. This is achieved when there are no repeated friends when performing the breadth-first search starting from my node.

1.2 Variation in clustering and diameter

As we increase i, I expect the edges to represent weaker friendships (in real life). Intuitively, I expect (d): clustering decreases, and the diameter decreases. The reasoning is as follows: I expect strong friends (in real life) to have high clustering. Increasing i corresponds to weaker social ties, so clustering should decrease. I also expect the diameter to decrease, since the Friendarchy graph for larger i will be based on weaker social ties (hence the edges become more "random" and "long-range").

1.3 Expansion coefficients

The expansion α of graph $G(V, E)$ is defined to be

$$\alpha = \min_{S \subset V} \left[\frac{\#\,\text{edges leaving } S}{\min(|S|, |V - S|)} \right]. \tag{1}$$

1.3.1 Complete balanced tree with $n = 2^{h+1} - 1$ nodes

The expansion is

$$\alpha = 1/(2^h - 1), \tag{2}$$

obtained by taking S to be one of the direct descendant subtrees of the root node.

1.3.2 Complete graph on n nodes

The expansion is

$$\alpha = n - n_{1/2}, \tag{3}$$

where $n_{1/2} = \text{floor}(n/2)$. This expansion is attained by taking S to be any subset of the complete graph with $n_{1/2}$ nodes.

1.4 $G_{n,p}$ contains a random 3-regular subgraph with high probability

1.4.1 Probability that node v in $G_{n,p}$ has degree less than 3

Let X_v describe the degree of node v. We know that the degree of node v in $G_{n,p}$ is binomially distributed (node v activates edges to $n-1$ other nodes independently with probability p). Hence, the desired probability is given by:

$$\begin{aligned}
p_{\deg(v)<3} &= Pr(X_v = 0) + Pr(X_v = 1) + Pr(X_v = 2), && (4)\\
&= (1-p)^{n-1} + (n-1)p(1-p)^{n-2} + \frac{(n-1)(n-2)}{2}p^2(1-p)^{n-3}, && (5)\\
&= \left((1-p)^2 + (n-1)p(1-p) + \frac{(n-1)(n-2)}{2}p^2\right)(1-p)^{n-3}, && (6)\\
&= \left(1 + (n-3)p + \frac{(n-3)(n-2)}{2}p^2\right)(1-p)^{n-3}, && (7)\\
&\leq \left(1 + np + n^2p^2\right)(1-p)^{n-3}. && (8)
\end{aligned}$$

We have introduced the inequality in Eq. 8 for later steps of the derivation.

1.4.2 Probability that there exists a node in $G_{n,p}$ with degree less than 3

Let E_i denote the event that node i in $G_{n,p}$ has degree less than 3. The event E that there exists a node in the graph with degree less than 3 is then:

$$E = \cup_{i=1}^{n} E_i. \tag{9}$$

Applying the union bound, we have:

$$\begin{aligned}
Pr(E) &\leq \sum_{i=1}^{n} Pr(E_i) = n \cdot p_{\deg(v)<3}, && (10)\\
&\leq n\left(1 + np + n^2p^2\right)(1-p)^{n-3}, && (11)
\end{aligned}$$

where we have made use of the inequality in Eq. 8.

1.4.3 If $p = \frac{2\log(n)}{n}$, with high probability $G_{n,p}$ has a diameter $O(\log(n))$

We begin with the bound for $Pr(E)$ derived in Eq. 11. Substituting $p = 2\log(n)/n$ yields:

$$Pr(E) \leq n\left(1 + 2\log n + 4\log^2 n\right)\left(1 - \frac{2\log n}{n}\right)^{n-3}, \tag{12}$$

Consider the rightmost term. Using l'Hopital's rule, we find $\lim_{n\to\infty} p = 0$, and thus:

$$\lim_{n\to\infty}\left(1 - \frac{2\log n}{n}\right)^{n-3} = \lim_{n\to\infty}\left(1 - \frac{2\log n}{n}\right)^{n} = \lim_{n\to\infty}\frac{1}{n^2}, \tag{13}$$

where, in the last equality, we have used the usual substitution of $\lim_{n\to\infty}(1-x/n)^n = e^{-x}$.

Applying this result to Eq. 12, we find:

$$\lim_{n\to\infty} Pr(E) \leq \lim_{n\to\infty} \frac{1+2\log n+4\log^2 n}{n} = 0. \tag{14}$$

We interpret the result as follows. In the limit $n \to \infty$, a random Erdos-Renyi graph $G_{n,p}$ with $p = 2\log(n)/n$ consists of nodes with degree of at least 3. In other words, such $G_{n,p}$ has a "superset" of the connectivity of a random 3-regular graph. Since the diameter of the random 3-regular graph is $O(\log(n))$, our $G_{n,p}$ has at most a diameter of $O(\log(n))$.

2 Signed networks over time

2.1 The G^+ model is unbalanced in the limit $n \to \infty$

2.1.1 Lower bound for $|T|$

Here is a simple procedure for generating a disjoint-edge set of triangles on a complete graph on n nodes.

Choose one node. Divide the remaining $n-1$ nodes into groups of 2. Each set creates a disjoint triangle when considered together with the initially chosen node. There are $\lfloor (n-1)/2 \rfloor$ such triangles, hence:

$$|T| \geq \lfloor (n-1)/2 \rfloor. \tag{15}$$

2.1.2 Probability that a triangle in G is balanced

For each triangle, we have to assign three signs. Of the possible sign combinations, $+++$, $+--$ are balanced, whereas $++-$ and $---$ are unbalanced. Since the probability of assigning $+$ to an edge is p, the probability that a triangle with randomly signed edges is balanced is given by:

$$Pr(\text{balanced triangle}) = p^3 + 3p(1-p)^2. \tag{16}$$

2.1.3 Upper bound on the probability that *all* of the triangles in T are balanced

Since the triangles in T are disjoint and the sign of each edge is independently assigned, each triangle is balanced or unbalanced independently. The probability that all triangles of T are balanced is then:

$$Pr(\text{T balanced}) = \left(p^3 + 3p(1-p)^2\right)^{|T|} \leq \left(p^3 + 3p(1-p)^2\right)^{\lfloor (n-1)/2 \rfloor}, \tag{17}$$

where the last equation follows since $\lfloor (n-1)/2 \rfloor \leq |T|$. If $p \neq 1$ (*i.e.* signs can be negative), the upper bound clearly approaches zero as $n \to \infty$.

2.1.4 Hence $P(G_B) \to 0$ as $n \to \infty$

If the graph G is balanced, then every triangle in G must be balanced. Hence, it is a necessary condition that all triangles in T are balanced. Since the probability of the necessary condition approaches zero as $n \to \infty$, it follows that the probability of G being balanced also approaches zero in the limit.

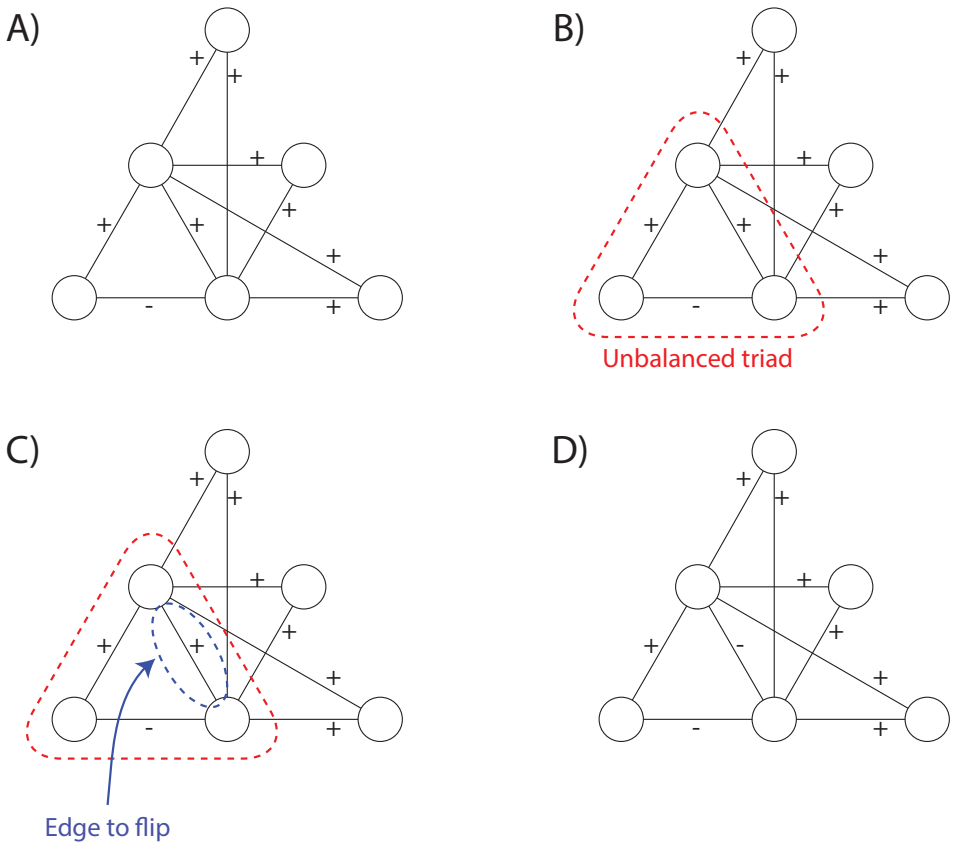

Figure 1: Demonstration of how the "dynamic" balance generation mechanism can cause a decrease in the number of balanced triads overall. Here, in the initial state (panel A) there are three balanced and one unbalanced triangle. In the final state (panel D), there are one balanced triangle and three unbalanced triangles.

2.2 Dynamic generation of balanced networks

The statement is **false**. It is possible to decrease the number of balanced triads by the dynamical process. Consider the following example, illustrated in Fig. 1:

1. Panel A: The initial state of the example network. There are three balanced triangles, and one unbalanced triangle at the bottom left.
2. Panel B: The random triad selection finds the unbalanced triad (dashed triangle) on the lower left.
3. Panel C: The random edge selection selects the edge marked in blue to be flipped.
4. Panel D: Updated state of the example network. There are now three unbalanced triangles and one balanced triangle.

2.3 Simulation of dynamic balance

2.3.1 Mechanism for checking whether a complete graph is balanced

The fact that we are dealing with a complete graph gives a simple way of checking whether a particular signed graph is balanced or not. The simplicity comes from the fact that, since all nodes are connected, there are no "implicit" factions (assuming the graph to be balanced). Instead, by looking at the signed connectivity of any one node, we can partition the nodes into two potential factions.

Once the two (potential) factions have been identified from a single node, it remains to be seen if they are consistent with the remainder of the graph. Here, I use the following test: if a set of nodes are indeed a faction in a signed graph, then the set of nodes represented in vector form (*i.e.* $g_i^{(k)} = 1$ if node i is an element of faction $k = 1, 2$, $g_i^{(k)} = 0$ otherwise) will be an eigenvector of the "positive adjacency matrix" A_p of the graph (*i.e.* $A_p(i, j) = 1$ if nodes i and j are connected by a + edge, $A_p(i, j) = 0$ otherwise). The corresponding eigenvalue will be the number of nodes in that faction. For more details, please see the attached Matlab code.

2.3.2 Simulation results

In my 100 simulations of the dynamic balance process, all 100 ended up balanced.

2.4 Not possible to add balanced node to an unbalanced $++-$ triangle

It is not possible to add a node D to the unbalanced $++-$ graph consisting of A, B, C without introducing an unbalanced triangle involving D. I enumerated all possibilities of the signs, and found for each case an unbalanced triangle containing D. See Table 1 below.

2.5 Not possible to add balanced node to an unbalanced graph

I considered the task of adding a node D to an unbalanced $---$ graph consisting of A, B, C. It can be shown by enumeration that it is not possible to add a node D to the $---$ graph that does not create an unbalanced triangle containing D.

D–A	D–B	D–C	Unbalanced triangle
+	+	+	BCD
+	+	−	ACD
+	−	+	ABD
+	−	−	ABD
−	+	+	ABD
−	+	−	ABD
−	−	+	ACD
−	−	−	BCD

Table 1: Enumeration of all possible signs that D can have with the triangle ABC. For each case, there is an unbalanced triangle containing D.

It follows that it is not possible to add a new node X to an unbalanced graph and form edges to all existing nodes such that X does not become involved in any unbalanced triangles. If the graph is unbalanced, then there is at least one existing unbalanced triangle in the graph. We have shown that it is not possible to add a node X to the $++-$ or $---$ unbalanced graphs that does not involve X in an unbalanced triangle. Hence, the new node must be involved in unbalanced triangles.

```m
clear all; close all;

n = 10;
p = 0.5;

Nexp = 100;
results = zeros(Nexp,1);

for exp = 1:Nexp
    fprintf('Experiment %03d...\n',exp);
    A = getRandomSignedGraph(n,p);

    Niter = 1e6;
    for iter = 1:Niter
        balanced = isGraphBalanced(A);
        if (balanced)
            break;
        end

        % I) Pick a triad at random
        triad_nodes = randsample(n, 3, false); % Sample without replacement
        triad_edges = [triad_nodes circshift(triad_nodes,-1)];

        % II) Check if triad is balanced. Use the fact that, in a balanced
        %       triangle, the product of the signs of the three edges is
        %       positive.
        triad_balanced = 1;
        for k = 1:3
            triad_balanced = triad_balanced * ...
                             A(triad_edges(k,1),triad_edges(k,2));
        end
        triad_balanced = triad_balanced > 0;

        % III) If the triad is not balanced, then choose one of the edges
        %       at random and flip its sign
        if (~triad_balanced)
            k = randi(3,1);
            A(triad_edges(k,1),triad_edges(k,2)) = ...
                -A(triad_edges(k,1),triad_edges(k,2));

            A(triad_edges(k,2),triad_edges(k,1)) = ...
                -A(triad_edges(k,2),triad_edges(k,1));
        end
    end

    results(exp) = balanced;
end
```



```m
function balanced = isGraphBalanced(A)

N = size(A,1);
Ap = A > 0; % "Positive adjacency matrix"

% First, we identify the candidate faction by looking
%   at the edges of node 1. We encode the candidate factions
%   into vectors g1 and g2.
%-------------------------------------------------------------
g1 = zeros(N,1); g1(1) = 1;
g2 = zeros(N,1);
for i = 2:N
    if (A(1,i) > 0)
        g1(i) = 1;
    else
        g2(i) = 1;
    end
end

% Second, we check if the candidate faction vectors are eigenvectors
%   of the positive adjacency matrix, with eigenvalue equal to the
%   size of the respective faction
%-------------------------------------------------------------
n1 = sum(g1);
n2 = sum(g2);

balanced = all(Ap*g1 == n1*g1) && ...
           all(Ap*g2 == n2*g2);
```



```m
% Returns a signed adjacency matrix of a complete graph on n nodes,
%   where each edge has probability p (1-p) of being + (-)
function A = getRandomSignedGraph(n,p)

A = 1/2*eye(n); % We'll later set A = A + A'
for i = 1:(n-1)
    for j = (i+1):n
        if (rand()<p)
            A(i,j) = 1;
        else
            A(i,j) = -1;
        end
    end
end

A = A + A';
```

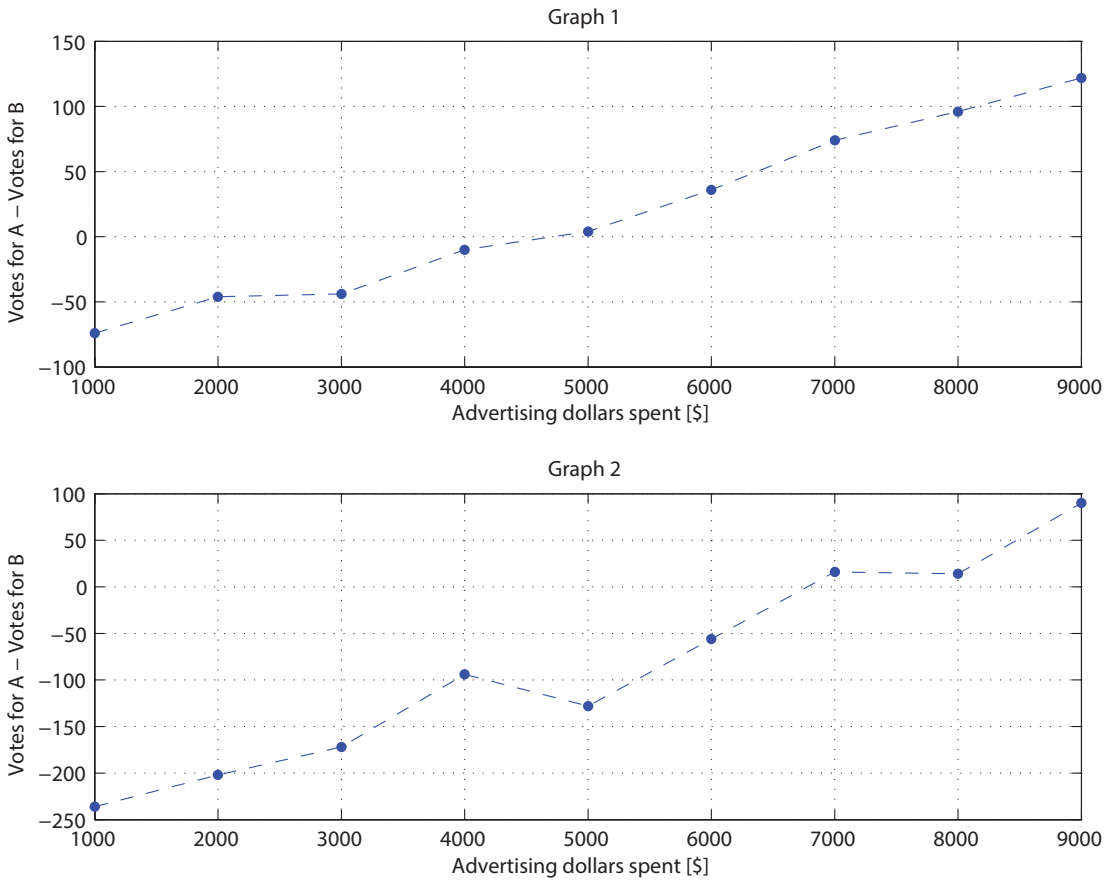

Figure 2: Using advertising money to increase the votes for A.

3 Decision-based cascades

3.1 Basic setup and forecasting

The results are as follows:

- Graph 1: Candidate B wins, by 96 additional votes;
- Graph 2: Candidate B wins, by 256 additional votes.

3.2 TV advertising

The result of spending k dollars on advertising is shown in Fig. 2. In graph 1, we need to spend about \$5000 in order to win; in graph 2, we need to spend a bit more, around \$7000.

Graph 1		Graph 2	
Node id	Degree	Node id	Degree
354	38	12	527
896	38	11	504
7035	38	10	443
682	36	17	413
804	36	16	402
1878	36	15	386
3685	36	6	353
5190	36	26	343
2704	35	18	339

Table 2: List of the high-rollers (nodes with maximal degree) in graphs 1 and 2. A salient difference between the two graphs is that graph 2 has nodes that have significantly high degrees.

Graph 1		Graph 2	
Node id	Degree	Node id	Degree
354	38	17	413
896	38	16	402
7035	38	15	386
804	36	6	353
1878	36	26	343
3685	36	18	339
2704	35	14	322
3307	35	5	257
7137	35	4	255

Table 3: List of the high-rollers in graphs 1 and 2 that are *not* hard-wired to vote for candidate A (*i.e.* the last digit of the node ID is not $0 - 3$).

3.3 Wining and dining the high rollers

First, I found it helpful to identify the "high-rollers" (nodes with the largest degrees in the social graph). In the context of the problem, we will be inviting at most 9 such high-rollers, so it suffices to find the nine "highest-rollers" from the two graphs.

Note: I used a method for finding the nodes with the maximum out-degree, that is slightly inaccurate. At each iteration, I use Snap.py's `snap.GetMxDegNId` to find the node with the highest out degree. I then remove the chosen node from the graph, and iterate. Now, it is possible that the explicit removal of nodes may affect the ordering of the highest rollers. However, I expect that the effect will be negligible for this problem.

The list of the top 9 high-rollers in graphs 1 and 2 are shown in Table 2.

Now, as a campaign manager for candidate A with a finite war chest, I am interested in targeting high-rollers who were going to (originally) vote for candidate B. (There is no point to "wasting" a $1000 dinner on folks who are die-hard candidate A fans anyway.) So, Table 3 shows a list of high-rollers in the two graphs who were not in the candidate A camp to begin with.

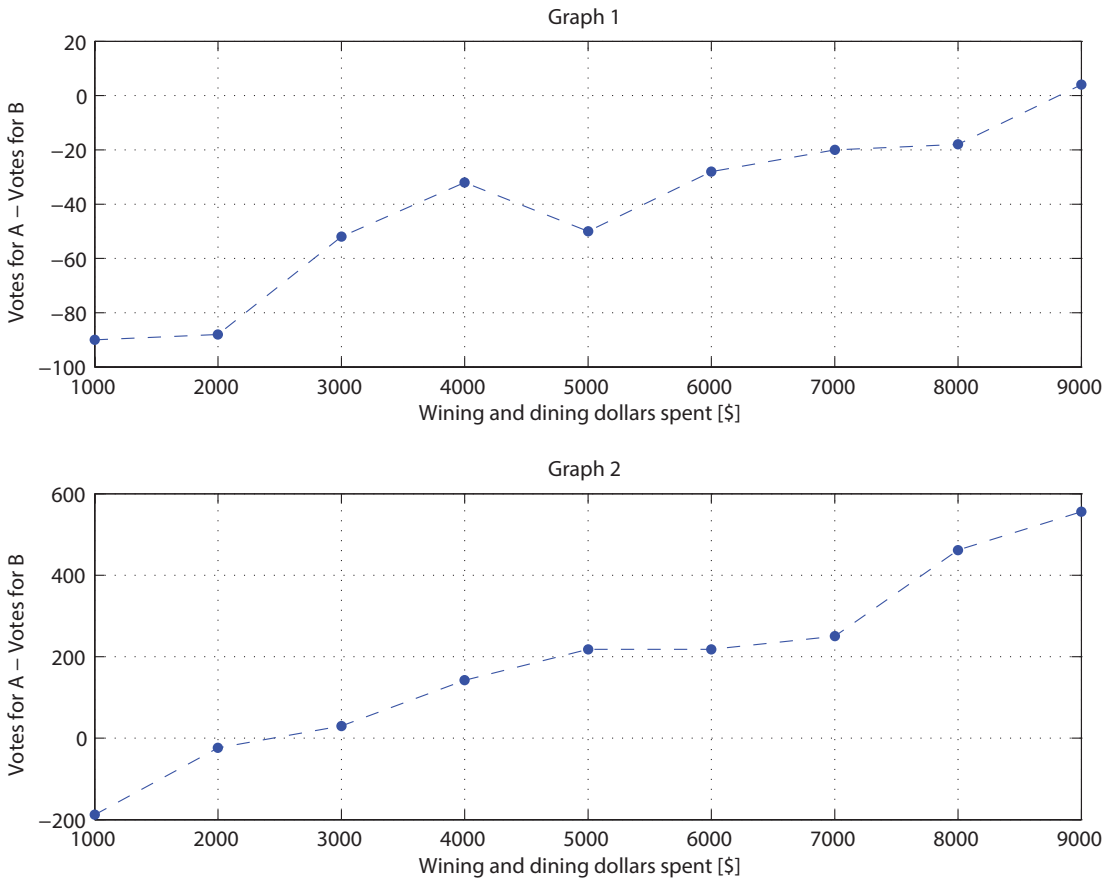

Figure 3: Using targeted fine wining and dining to increase the votes for A.

The effects of fine wining and dining are shown in Fig. 3. For graph 1, we have to exhaust our budget in order to get sufficient votes to win (and just barely, at that). In graph 2, we can spend about \$3000 (*i.e.* fine wine and dine 3 people) to achieve victory for candidate A.

3.4 Analysis

As shown in Table 2, the salient difference between graphs 1 and 2 is that, in the latter, there exist a few nodes that are highly influential (large out degree compared to the average degree of $\bar{k} \approx 10$ in both graphs).

This means that if the voting behavior is based on graph 1, using the campaign funds I want to convert as many people as possible without regards to their relative influence – so I would proceed with the advertising strategy. On the other hand, if graph 2 is the true underlying network, then I would use the "fine wine and dine" strategy to convert the highly influential nodes in the graph.

```
# Tony Hyun Kim
# 2013 10 22
# CS 224w, PS 2, Problem 3

import numpy as np
import snap

# Load the graphs
#-------------------------------------------------------------
G1 = snap.LoadEdgeList(snap.PUNGraph, "g1.edgelist", 0, 1)
G2 = snap.LoadEdgeList(snap.PUNGraph, "g2.edgelist", 0, 1)

# Initialize the voting state
#   I will use the following integer encoding:
#       0: Undecided
#       1: Candidate A
#      -1: Candidate B
#-------------------------------------------------------------
G = G1

# I decided to use a numpy vector for holding state, because
#   there seems to be issues with Snap.py attributes
N = 10000 # Number of nodes
state = np.zeros((N,1), int)

# The intial voting state is based on the last digit of nid
init_vote = {0: 1,
             1: 1,
             2: 1,
             3: 1,
             4: -1,
             5: -1,
             6: -1,
             7: -1,
             8: 0,
             9: 0}
for nid in range(N):
    state[nid] = init_vote[nid % 10]

tiebreak = 1 # Global variable for breaking ties

for day in range(1,11):

    state_next = np.zeros((N,1), int)

    # Iterate over all nodes in increasing Id
    for ni_id in range(N):
        # Only the initially undecided voters get modified
        if (init_vote[ni_id % 10] != 0):
            state_next[ni_id] = state[ni_id]
        else: # Undecided voter
            ni = G.GetNI(ni_id)
            friends_votes = 0
            for nj_id in ni.GetOutEdges():
                # If possible, use values from current iteration
                if (nj_id < ni_id):
                    friends_votes += state_next[nj_id]
                else:
                    friends_votes += state[nj_id]

            if (friends_votes == 0):
                state_next[ni_id] = tiebreak
                tiebreak *= -1
            elif (friends_votes > 0):
                state_next[ni_id] = 1
            else:
                state_next[ni_id] = -1

    # Update state
    state = state_next
    print "At end of day {0:02d}, sum(state)={1:d}".format(day, np.sum(state))
```



```
# Tony Hyun Kim
# 2013 10 22
# CS 224w, PS 2, Problem 3

import numpy as np
import snap

# Load the graphs
#-------------------------------------------------------------
G1 = snap.LoadEdgeList(snap.PUNGraph, "g1.edgelist", 0, 1)
G2 = snap.LoadEdgeList(snap.PUNGraph, "g2.edgelist", 0, 1)

# Initialize the voting state
#   I will use the following integer encoding:
#       0: Undecided
#       1: Candidate A
#      -1: Candidate B
#-------------------------------------------------------------
G = G2

# I decided to use a numpy vector for holding state, because
#   there seems to be issues with Snap.py attributes
N = 10000 # Number of nodes
state = np.zeros((N,1), int)

# The intial voting state is based on the last digit of nid
init_vote = {0: 1,
             1: 1,
             2: 1,
             3: 1,
             4: -1,
             5: -1,
             6: -1,
             7: -1,
             8: 0,
             9: 0}

# The following {id: vote} dict has the highest priority in
#   determining vote pattern
overrides = {}
for k in range(3000,3090):
    overrides[k] = 1

for nid in range(N):
    if nid in overrides:
        state[nid] = overrides[nid]
    else:
        state[nid] = init_vote[nid % 10]

tiebreak = 1 # Global variable for breaking ties

for day in range(1,11):

    state_next = np.zeros((N,1), int)

    # Iterate over all nodes in increasing Id
    for ni_id in range(N):
        if ni_id in overrides: # "Die-hard" voters
            state_next[ni_id] = overrides[ni_id]
        elif(init_vote[ni_id % 10] != 0):
            state_next[ni_id] = init_vote[ni_id % 10]
        else: # Undecided voter
            ni = G.GetNI(ni_id)
            friends_votes = 0
            for nj_id in ni.GetOutEdges():
                # If possible, use values from current iteration
                if (nj_id < ni_id):
                    friends_votes += state_next[nj_id]
                else:
                    friends_votes += state[nj_id]

            if (friends_votes == 0):
                state_next[ni_id] = tiebreak
```



```
                tiebreak *= -1
            elif (friends_votes > 0):
                state_next[ni_id] = 1
            else:
                state_next[ni_id] = -1

    # Update state
    state = state_next
    print "At end of day {0:02d}, sum(state)={1:d}".format(day, np.sum(state))
```



```
# Tony Hyun Kim
# 2013 10 22
# CS 224w, PS 2, Problem 3

import numpy as np
import snap

# Load the graphs
#-------------------------------------------------------------
G1 = snap.LoadEdgeList(snap.PUNGraph, "g1.edgelist", 0, 1)
G2 = snap.LoadEdgeList(snap.PUNGraph, "g2.edgelist", 0, 1)

# Initialize the voting state
#   I will use the following integer encoding:
#       0: Undecided
#       1: Candidate A
#      -1: Candidate B
#-------------------------------------------------------------
G = G2

# I decided to use a numpy vector for holding state, because
#   there seems to be issues with Snap.py attributes
N = 10000 # Number of nodes
state = np.zeros((N,1), int)

# The intial voting state is based on the last digit of nid
init_vote = {0: 1,
             1: 1,
             2: 1,
             3: 1,
             4: -1,
             5: -1,
             6: -1,
             7: -1,
             8: 0,
             9: 0}

# The following {id: vote} dict has the highest priority in
#   determining vote pattern
'''
# Hit list for G1
overrides = {354: 1,
             896: 1,
             7035: 1,
             804: 1,
             1878: 1,
             3685: 1,
             2704: 1,
             3307: 1,
             7137: 1,
            }
'''
# Hit list for G2
overrides = {17: 1,
             16: 1,
             15: 1,
             6: 1,
             26: 1,
             18: 1,
             14: 1,
             5: 1,
             4: 1,
            }

for nid in range(N):
    if nid in overrides:
        state[nid] = overrides[nid]
    else:
        state[nid] = init_vote[nid % 10]

tiebreak = 1 # Global variable for breaking ties

for day in range(1,11):
```

```
    state_next = np.zeros((N,1), int)

    # Iterate over all nodes in increasing Id
    for ni_id in range(N):
        if ni_id in overrides: # "Die-hard" voters
            state_next[ni_id] = overrides[ni_id]
        elif(init_vote[ni_id % 10] != 0):
            state_next[ni_id] = init_vote[ni_id % 10]
        else: # Undecided voter
            ni = G.GetNI(ni_id)
            friends_votes = 0
            for nj_id in ni.GetOutEdges():
                # If possible, use values from current iteration
                if (nj_id < ni_id):
                    friends_votes += state_next[nj_id]
                else:
                    friends_votes += state[nj_id]

            if (friends_votes == 0):
                state_next[ni_id] = tiebreak
                tiebreak *= -1
            elif (friends_votes > 0):
                state_next[ni_id] = 1
            else:
                state_next[ni_id] = -1

    # Update state
    state = state_next
    print "At end of day {0:02d}, sum(state)={1:d}".format(day, np.sum(state))
```



```
# Tony Hyun Kim
# 2013 10 22
# CS 224w, PS 2, Problem 3

import numpy as np
import snap

# Load the graphs
#-------------------------------------------------------------
G1 = snap.LoadEdgeList(snap.PUNGraph, "g1.edgelist", 0, 1)
G2 = snap.LoadEdgeList(snap.PUNGraph, "g2.edgelist", 0, 1)

# Determine the highest rollers of the graph
#-------------------------------------------------------------
init_vote = {0: 1,
             1: 1,
             2: 1,
             3: 1,
             4: -1,
             5: -1,
             6: -1,
             7: -1,
             8: 0,
             9: 0}

G = G2

high_rollers = []
while (len(high_rollers) < 20):
    nid = snap.GetMxDegNId(G)

    # Target nodes that were not hard-wired for candidate A
    #if (True):
    if (init_vote[nid % 10] != 1):
        high_rollers.append(nid)
        n = G.GetNI(nid)
        print "node {} has degree {}".format(nid, n.GetOutDeg())

    G.DelNode(nid)
```

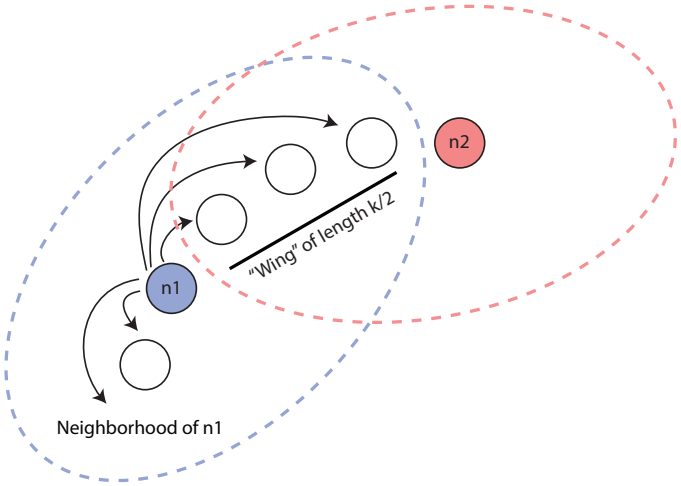

Figure 4: The maximum overlap of neighborhoods is $k/2$ if the two focal nodes are outside of the neighborhood of the other.

4 Complex contagions

We assume k is even.

4.1 Maximum overlap between two neighborhoods

The maximum overlap between two neighborhoods whose focal nodes do not lie in the other's neighborhood is $k/2$.

Consider Fig. 4. We first fix the first focal node n_1 (blue). The neighborhood includes k nodes nearest to n_1. On the ring topology, the neighborhood consists of two "wings" of length $k/2$ on each side of n_1. Next, we choose the location of focal node n_2 (red). We wish to maximize the overlap of the two neighborhoods, but n_2 cannot lie in the direct neighborhood of n_1. So, choose n_2 to be just outside the "wing" of n_1. Then, the overlap of the two neighborhoods will be $k/2$.

4.2 Maximum width $W_{\max}$ of a bridge

The maximum width of a bridge is $W_{\max} = \sum_{i=1}^{k/2} i = \frac{(k/2)(1+k/2)}{2}$.

Again, we consider the case where the two focal points n_1 and n_2 (with neighborhoods N_1 and N_2) are as close to each other as possible, but without being contained in the neighborhood of the other.

Consider Fig. 5(a). We have already shown that the maximal intersection $N_1 \cap N_2$ consists of $k/2$ elements. Call them x_i, $i = 1, 2, \cdots, k/2$. The $\{x_i\}$ are the potential source nodes of the bridge edges between N_1 and N_2. We then count the number of edges from $\{x_i\}$ that terminate in $N_2 - N_1$.

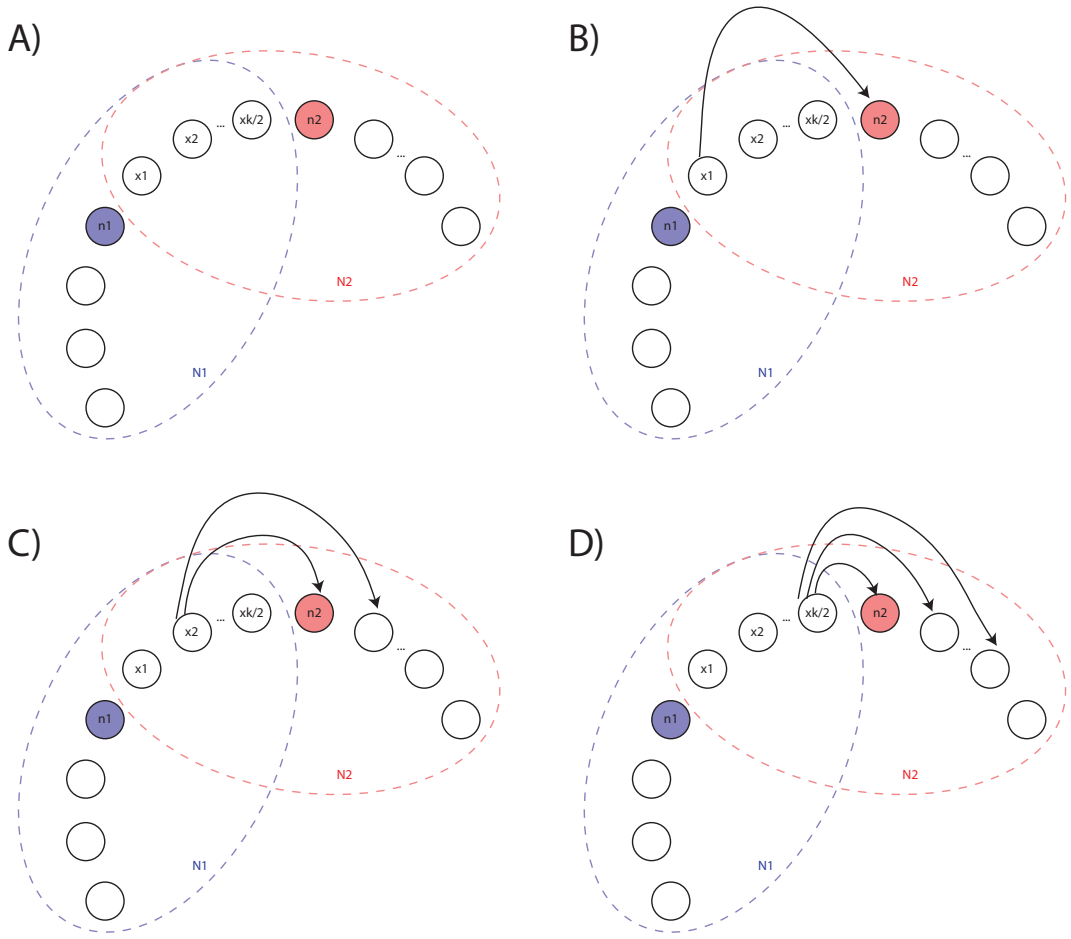

Figure 5: The maximum width between two neighborhoods is $W_{\max} = \sum_{i=1}^{k/2} i = \frac{(k/2)(1+k/2)}{2}$.

From Fig. 5(b,c,d), it can be seen that x_1 has 1 edge that lands in $N_2 - N_1$. Similarly x_2 has 2 edges into $N_2 - N_1$, and so on. So, in this configuration, there are a total of $\sum_{i=1}^{k/2} i = \frac{(k/2)(1+k/2)}{2}$ bridge edges.

It remains to be shown that the configuration in Fig. 5 produces the maximum width. We consider two perturbations from the shown configuration:

- Consider the case that n_2 is shifted clockwise, *i.e.* away from n_1. In this case, the size of the intersection $N_1 \cap N_2$ will decrease, and the number of bridge edges from $N_1 \cap N_2$ into $N_2 - N_1$ must decrease.
- Consider the case that n_2 is shifted counterclockwise, *i.e.* towards n_1. In this case, although the size of the intersection $N_1 \cap N_2$ will increase, the number of nodes ($\{x_i\}$) that can reach into $N_2 - N_1$ will not change, since all nodes left of n_1 (including n_1) have no edges into $N_2 - N_1$. On the other hand, as the N_2 is shifted counterclockwise, some of the edges from $\{x_i\}$ will no longer be contained in $N_2 - N_1$.

Thus, the configuration of Fig. 5(a) achieves the maximum width.

4.3 Critical width W_C as a function of a

The critical width is given by $W_C = \sum_{i=1}^{a} i = \frac{a(1+a)}{2}$.

Consider again Fig. 5(a). It can be readily observed that n_2 receives $k/2$ edges from $N_1 \cap N_2$, and that each node to the right of n_2 receives $k/2-1$, $k/2-2$, $\cdots$ edges. (The rightmost node of N_2 receives no edges from $N_1 \cap N_2$.)

The basic observation is that the node nearest N_1 receives the maximum number of edges from N_1.

Now, in determining the critical width W_C we do not want to "waste" redundant edges. So, we will let n_2 receive a edges (just enough to cause an infection). In that case, each node to the right of n_2 receives $a-1$, $a-2$, $\cdots$, 1 bridge edges. Summing all such edges, we find that the critical width is $W_C = \frac{a(1+a)}{2}$.

4.4 Required relation to prevent contagion spread

The contagion will be unable to spread in a given network if $W_{\max} < W_C$.

Equivalently, the contagion will be able to spread in a k-regular ring graph when $a \leq k/2$. This is a pretty straightforward result.