AC290r.1: Influence Maximization Problem

Stephen Carr and Charles Liu

October 16, 2015

1 Introduction

1.1 Goal and Motivation

From a marketing perspective, social networking sites provide a large number of users with personal and persuasive information regarding products. With limited resources, one would want to know how to reach advertisement to the most number of people in this personalized manner. This problem can be reduced to a graph, where each user is a node and each relationship between users is represented by an edge with some weight to indicate the strength of their relationship. Which user, or set of users, has the ability to spread your product to the greatest number of potential customers?

1.2 Data set

The user network of the online restaurant reviewing site Yelp is used in this study. Each account represents the node, and the edges are to be generated by looking at which accounts posted reviews of the same restaurant within a certain time period of one another. This information (the user, their total number of reviews, and how many reviews they have written within 1 month of another user) was pulled from the Yelp site, turned into a library format, and then imported into a python development environment using network x. For most of the prototyping of a solution, a subset of the North Carolina network was taken, which consisted of 240 nodes and ???? edges.

1.3 Independent Cascade, $I(s)$

We start with a set of nodes $s = \{n_1, n_2, ..., n_k\}$ and wish to find the number of nodes which are activated through our probabilistic network of influence. So for each node n in our original set s, we look at all nodes connected to n by an edge and use the beta distribution to determine if the connected node n_c activates (we ignore n_c if it is already activated). The beta distribution $\boldsymbol{B}$ is given by:

$$\boxed{\int_{-\infty}^{\infty} \frac{dx}{\boldsymbol{B}(\alpha, \beta)} x^{\alpha-1}(1-x)^{\beta-1} = 1} \tag{1}$$

Where β is taken to be the number of reviews node n_c has written within one month of node n writing a review of the same restaurant, and α is taken to be the number of reviews n_c has written. A uniform distribution is taken between 0 and 1 and if its value is less than $\boldsymbol{B}(\alpha, \beta)$ then n_c activates and we repeat the same process for neighbors of n_c. This recursive process continues until either all nodes are activated or no activated nodes are left in the loop. The number of nodes activated is called $I(s)$.

2 Stochasticity

2.1 Averaged Trial Function, $f_N(s)$

Since each step of the independent cascade is a random sampling of the beta distribution, there is no guarantee that two different measurements of $I(s)$ will be the same, or even "close" to one another, as seen in 1.

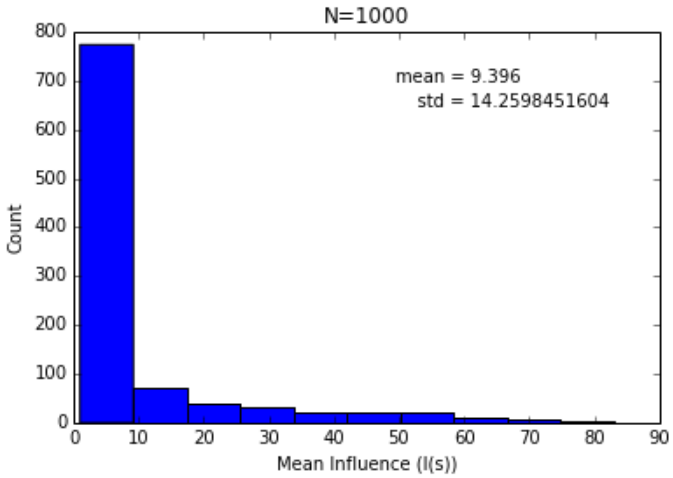


Figure 1: Histogram of values $I(s)$ over 1000 iterations for a fixed node in the 240 North Carolina subgraph.

Because of this, we must define a new measurement of a node's influence, namely its mean over N samples:

$$\boxed{f_N(s) = \frac{1}{N}\sum_{1}^{N} I(s)} \tag{2}$$

2.2 Properties of $f_N(s)$

Obviously we cannot take N arbitrarily large, so we must understand how f_N's properties vary with N. Taking a 240 node subset of North Carolina, we obtain

the following relationship between N and the standard deviation of f_N in 2.

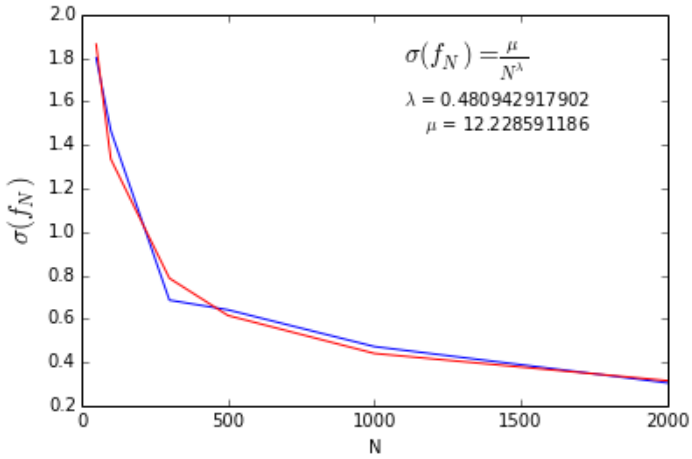


Figure 2: Fitted regression for the standard deviation of $f_N(s)$. This was with 100 trials for each N value. The blue line is the simulation, while the red line is the fitted relationship.

3 Greedy Algorithm

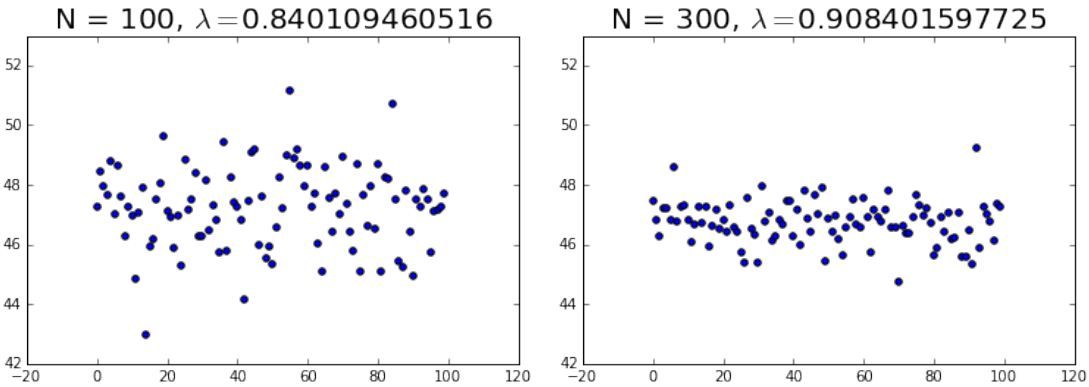
Although this problem is inherently non-linear in its parameters (the starting set of nodes), we can try to solve it in a linear way. This is called the "Greedy Algorithm": Given nodes $s_i = \{n_1, n_2, ..., n_i\}$, check every remaining node to find node n_{i+1} that maximizes $f(s_{i+1})$ where $s_{i+1} = \{n_1, n_2, ..., n_i, n_{i+1}\}$.

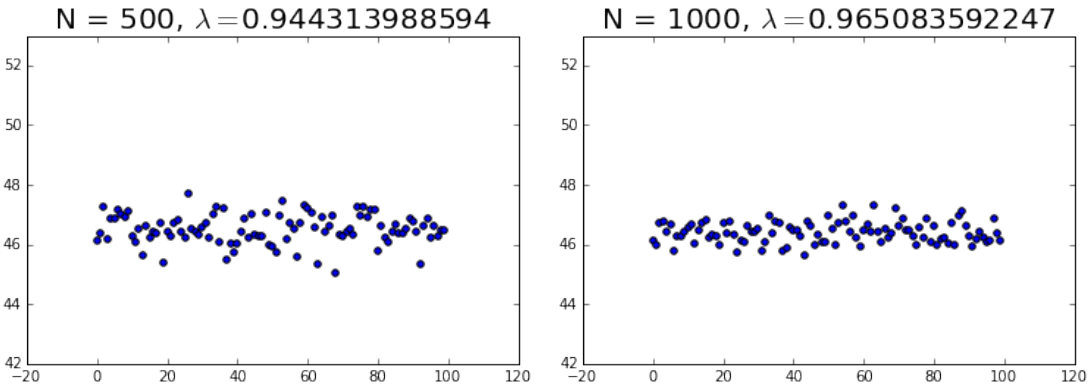
3.1 NC_Mini

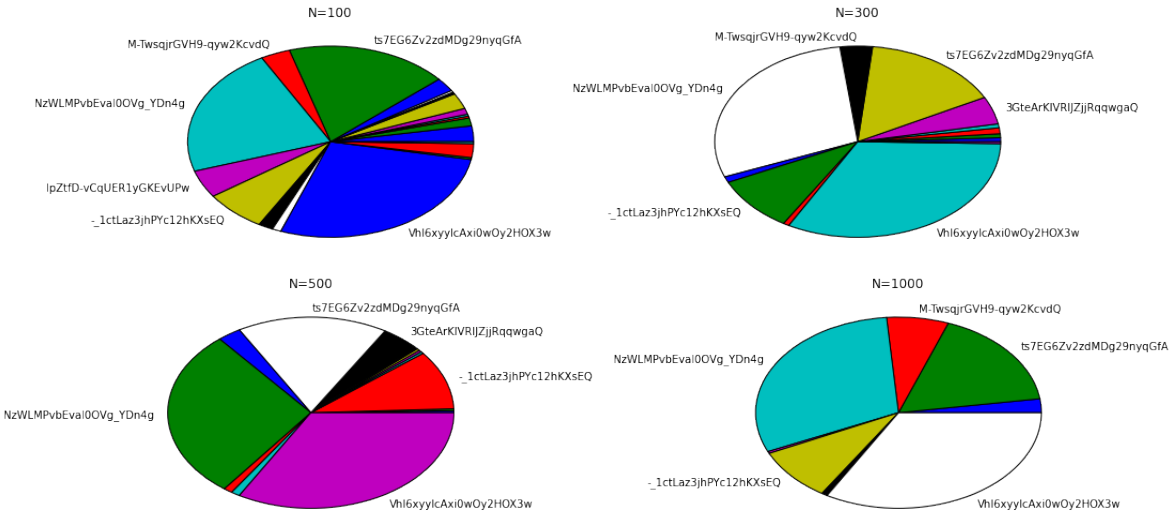
For $k = 3$ on our 240 node North Carolina subset we found variation in the maximal $I(s)$.

Table 1: 100 runs with k=3, t=10

N	min	max	λ
100	42.98	51.16	0.840
300	44.73	49.24	0.908
500	45.04	47.696	0.944
1000	45.66	47.31	0.965


N = 100, λ=0.840109460516
N = 300, λ=0.908401597725


N = 500, λ=0.944313988594
N = 1000, λ=0.965083592247


N=100
M-TwsqjrGVH9-qyw2KcvdQ
ts7EG6Zv2zdMDg29nyqGfA
NzWLMPvbEval0OVg_YDn4g
lpZtfD-vCqUER1yGKEvUPw
-_1ctLaz3jhPYc12hKXsEQ
Vhl6xyylcAxi0wOy2HOX3w
N=300
M-TwsqjrGVH9-qyw2KcvdQ
ts7EG6Zv2zdMDg29nyqGfA
NzWLMPvbEval0OVg_YDn4g
3GteArKlVRlJZjjRqqwgaQ
-_1ctLaz3jhPYc12hKXsEQ
Vhl6xyylcAxi0wOy2HOX3w
N=500
ts7EG6Zv2zdMDg29nyqGfA
3GteArKlVRlJZjjRqqwgaQ
-_1ctLaz3jhPYc12hKXsEQ
NzWLMPvbEval0OVg_YDn4g
Vhl6xyylcAxi0wOy2HOX3w
N=1000
M-TwsqjrGVH9-qyw2KcvdQ
ts7EG6Zv2zdMDg29nyqGfA
NzWLMPvbEval0OVg_YDn4g
-_1ctLaz3jhPYc12hKXsEQ
Vhl6xyylcAxi0wOy2HOX3w

The scatterplots of the max values over the 100 trials show the convergence of results as N increases. We can similarly see the convergence through the number of nodes selected over the course of our trials. More formally, let $\mathbb{N}_i$ represent the set of nodes returned over all 100 trials of $N = i$. We have

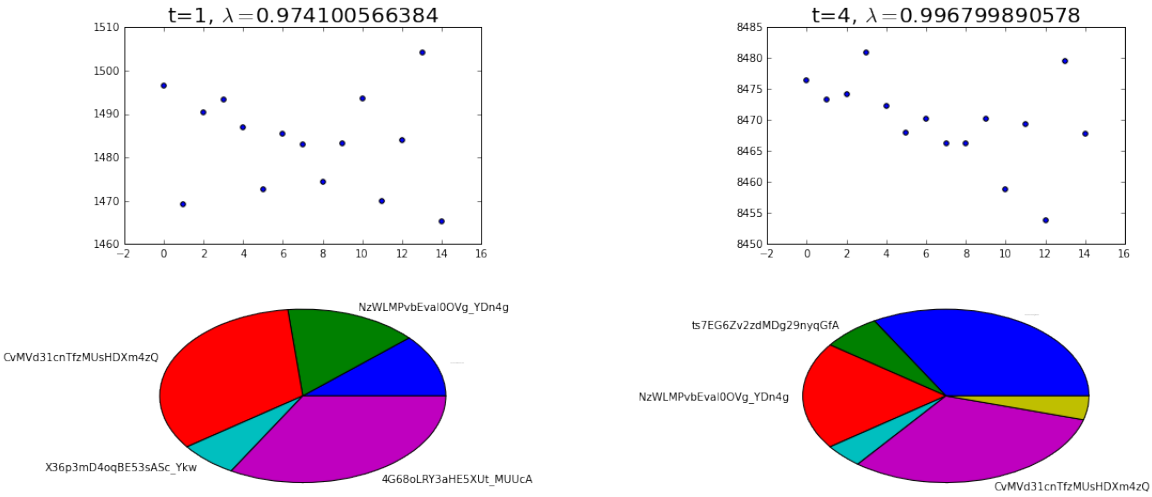
$$\boxed{\mathbb{N}_i \subset \mathbb{N}_j, i > j} \tag{3}$$

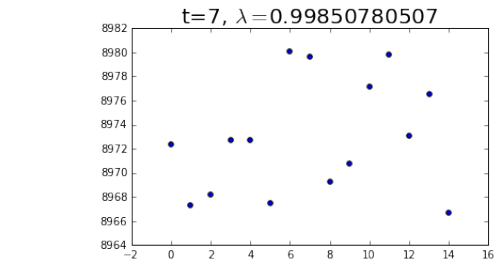
3.2 NC_Full

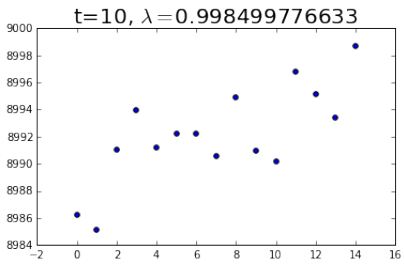
We wanted to do the same analysis with respect to the depth of search. For the larger NC_full graph, our results were:

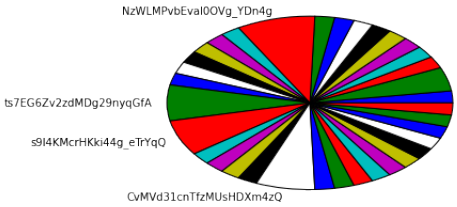
Table 2: 15 runs with k=3, N=50

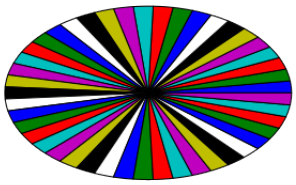
t	min	max	λ
1	1465.32	1504.28	0.9741
4	8453.82	8480.96	0.9968
7	8966.66	8980.06	0.9985
10	8985.16	8998.66	0.9985











Due to computational limitations, we were only able to run 15 trials for each depth. However, we were still able to see some interesting patterns. First as a sanity check, we saw the influence function return higher values as depth increased - this of course should be the case. Second, we see λ increase as depth increases for a consistent N. From $t = 7$ to $t = 10$ this seems to plateau, which may point to some bounds on the diameter of the largest connected components in the graph. Finally, unlike with increasing N, we see that the set of nodes returned over the trials increases as depth increases. In fact for $t = 10$, 45 different nodes returned meaning there was a different node in every result.

What this implies is that there is a large connected component within NC_full that, regardless of the nodes chosen, will give you roughly the optimal number from the greedy algorithm. We wrote a function to calculate the number of connected components from the set of nodes returned and found such:

Table 3: Connected Components for t

t	Total Nodes	Connected Components	Max Component Size
1	5	1	5
4	6	1	6
7	35	6	30
10	45	8	38

This helps to explain why we see an increase of roughly 500 nodes reached going from $t = 4$ to $t = 7$ whereas only about 20 extra nodes reached from $t = 7$ to $t = 10$.

Given that we know this information now, we can smartly choose three nodes that cover different components and run on a larger N. We should also beat the heuristic implementation of choosing the three nodes with the most edges, because as it turns out those three nodes are all in the same large component.

For the smartly chosen nodes from different components, we want to choose the two nodes with the most edges in two other components in our set of 8 in the $t = 10$ run. It turns out that every node in the other components had only 1 edge, so we chose two at random.

Table 4: Heuristic vs. Best Components Mix on N=1000

Nodes	Mean
Heuristic	8960.0698
Component Mix	8970.6878

Heuristic nodes:

[CvMVd31cnTfzMUsHDXm4zQ,
NzWLMPvbEval0OVg_YDn4g,
4G68oLRY3aHE5XUt_MUUcA]

Component mix:

[wadTky-Lw9MVqDhiJUqPJA,
OYrEBnzYbL3dkoVdy1C10Q,
CvMVd31cnTfzMUsHDXm4zQ]

We have a lower value than our max from the $t = 10$ run, which is expected as we saw a similar convergence in the NC_mini tests. We do in fact see a better result from our component mix than the heuristic.

3.3 US Graph

Using what we learned from our experiments on NC_mini and NC_full, we test three different algorithms for finding the most influential users.

Heuristic Algorithm

This is the standard heuristic algorithm that specifies the three nodes with the most edges. We found with N=100, t=10:

Nodes =

[Iu3Jo9ROp2IWC9FwtWOaUQ,
OaFcpi3W4AwxrD8W2pgC_A,
glRXVWWD6x1EZKfjJawTOg]

Influence = 184194.68

Heuristic Algorithm V2

Given the possibility of a heavily connected component, we can adjust our heuristic algorithm to choose the node with the most edges, remove all connected nodes to that edge, and reiterate until we have 3 nodes. We found with N=100, t=10:

Nodes =

[Iu3Jo9ROp2IWC9FwtWOaUQ,
2rghC4XAXTXuY_ZGocVB1g,
znSMP7mggycgF-NP1rz93g]

Influence = 184188.89

Greedy Algorithm

We devise a greedy algorithm that checks the connected components and iterates on N and t. We define the following variables:

N_0	Initial N
t_0	Initial t
I_N	Increment for N
I_t	Increment for t
S	Set of nodes
K	Set of selected nodes

1. Initialize $N = N_0, t = t_0, S =$ all nodes, $K = \emptyset$
2. Run cascade function and return set of max nodes X
3. Divide X into connected components, select node n with most edges in most populated connected component
4. $K = K \cup [n]$
5. $S = S \setminus \{\text{Nodes connected to } n\}$
6. $N = N + I_N$
7. $t = t + I_t$
8. Repeat steps 2-7 until $|K| = k$

As of October 16, 2015, we have not been able to run this on the full US graph.

3.4 Future steps

The initial and incremental N and t in the greedy algorithm can be calibrated. It's effect will be a tradeoff of the number of max nodes returned (and therefore the possible size of solutions) and the runtime of the algorithm.

3.5 Source Code

*Section header is a link to github

Spark_N.ipynb

NC_mini runs for N=100, 300, 500, 1000

Greedy-Full

NC_full runs on Azure

Greedy_Analysis
Compilation and Analysis of test runs

sparkruns
Folder of json's of all tests

4 Implementation

4.1 MCMC and Simulated Annealing

For a large problem with N nodes, the greedy algorithm would need to check roughly N^k node arrangements (first N, then $N-1$, etc.). For $k > 2$ this will not be feasible. Instead, we implement a Monte Carlo method of picking sets of nodes at random and evaluating their influence. Although this method will not guarantee we find the optimal solution, we can define a iterative method (Markov Chain) that on average will find a better solution after each step.

The idea of a Markov Chain Monte Carlo (MCMC) algorithm is to use the function of interest to assign a weight to every position in parameter space, and use this weight to add control to the "random steps" of a Monte Carlo method. Our function is the mean influence, $f_N(s)$, and our parameter space is the unordered combination of k nodes $s = \{n_i\}$. Instead of always taking a "random step" around parameter space, we assign to each step a probability of success:

$$\boxed{P(s_1, s_2) = e^{\frac{-\Delta E}{T}}, \Delta E(f(s_2), f(s_1))} \tag{4}$$

Where s_1 is our current set of nodes and s_2 is a new proposed set. Controlling the Monte Carlo method in this manner is known as Simulated Annealing, as the probability formula mimics the Boltzmann factor of statistical mechanics. The Boltzmann factor describes the relative probability of a particle being in state s when it is in a thermal equilibrium of temperature T. Annealing such a system means heating it up and then slowly cooling it down, allowing the particle(s) to settle into a new arrangement, hopefully of lower energy. For our problem we take the "Energy" to be $-f_N(s)$, since we want to maximize $f_N(s)$.

The simplest form for ΔE is $f(s_1) - f(s_2)$. In this case, if $f(s_2) > f(s_1)$ we take s_2 with probability 1, otherwise there is a chance of taking s_2 even though its estimated influence is less than our current state. This gives us an additional parameter to control our MCMC method: the "Temperature" T controls how likely we are to take a step away from a maximum. So when we are not "near" a local maximum, a high T allows us to sweep the parameter space quickly, but as we near a solution we take a low T to make the MCMC very sensitive to small changes in $f_N(s)$.

4.2 Advantages and Shortcomings

As the MCMC is an iterative process that gives a solution from the first step that slowly improves with each iteration, it is better than the greedy algorithm which needs to finish in its entirety before an answer can be output. Furthermore, the greedy algorithm does not guarantee to find the answer, and it should find the same sub-optimal solution everytime (provided N is large enough). The MCMC will "eventually" find the global maximum influence, as it will eventually cover all possible permutations of nodes. So for large graphs, the MCMC method is clearly superior, so the only work left to do is to find how to speed up each iteration without sacrificing accuracy. A number of methods to speed up the MCMC process is covered next.

4.3 Choice of Parameters for Speed

4.3.1 Δ E

For small graphs, $\Delta E = f(s_1) - f(s_2)$ will work. But for large graphs, this will allow $P(s_1, s_2)$ to be unsuitable large or small. Instead we take $\Delta E = log(f(s_1)) - log(f(s_2))$, allowing $P(s_1, s_2)$ to be tractable both conceptually and computationally.

4.3.2 N

Table of μ and λ as a function of $f_N(s)$ for NC small,NC large, (And US?)

4.3.3 T

Start with $T = 1$. After each successful step, set $T_{n+1} = \alpha * T_n$ where $\alpha < 1$.

4.3.4 Allowed Nodes (s_i)

Only allow the top 5% or 10% of nodes (by edge count or by edges of neighbors). For generating s_{n+1} given s_n, randomly swap one of the nodes of s_n to any other node ("closeness" is controlled by keeping the other nodes the same, NOT by staying "near" the changed node on the graph).

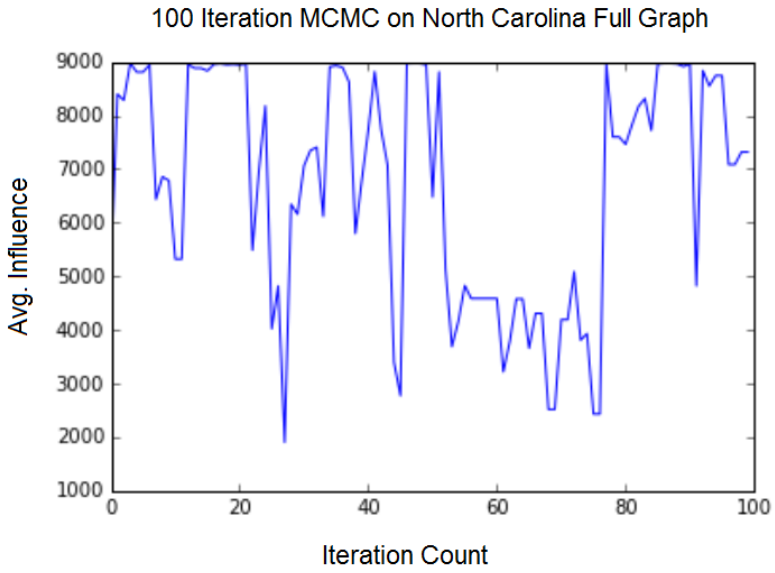
4.3.5 Re-annealing Control

If no new local maximum is found in k attempts, set $T = 1$ to "restart" the search process (i.e. allow for "large steps" again but start from the current set of nodes).

5 Results

5.1 North Carolina (mini)

5.2 North Carolina (full)



Max influence = 8975.7

Nodes (by hash): `u'9ki_I7oW6YxnUZBCmxs5Q', u'GllP-lYGO1m61G4ly5fYTQ', u'iCePpGSAgAOSYZ9TkCQ09w'`

All MCMC runs seem to max out around 9000 influence...?