**REPORT FOR TEAM CPU**

**September 30, 2015**

**Part a: Introduction**

**1. Problem**

Before working with the large-scale national yelp reviewer data, we would like to begin with a sample of North Carolina data. For the pre-built graph of Yelp Reviewer network, we will use NetworkX to implement the Independent Cascade function.

**2. Independent Cascading**

Independent cascading is implemented to find how many nodes a starting node can finally have effect on in the whole network. Here, it is for exploring how big a single reviewer’s influence can be. Notice that independent cascading is first applied to a single starting node.

**3. Influence Function**

After applying the independent cascading function, we can get the influence that a starting node have on the whole networking. However, the result may not be accurate enough, so we repeat the same process N times and get the mean of these N numbers. By doing this, we can get a more accurate result with respect to a certain node’s influence.

**Part b: Algorithms**

**1. Greedy Algorithm**

A greedy algorithm is a mathematical process that looks for optimal solution. Based on the current state, it simply decides which next step will provide the most obvious benefit. This algorithm doesn’t consider the larger problem as a whole. Once a decision has been made, it is never reconsidered.

1. Initialize the current node as basis, the node set, max influence and max node
2. For node\_i in the node set
3. Compute the combined influence i of current node and node i
4. If influence i > max influence,

set influence i as max influence, and node i as max node

1. Return max node

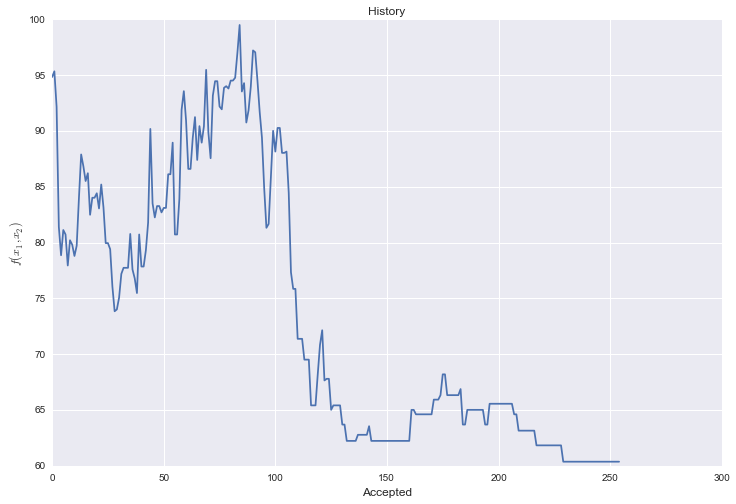
**2. SA (Salesman Annealing)**

Salesman Annealing is about the following problem: given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city. This is an optimization problem that can be solved by simulated annealing.

1. Initialize x0, T, L, where L is the step size
2. For i through imax:
3. Find a new x\* in (xi, L), change the path for this new x\* by switching index
4. Generate u from uniform distribution (0, 1)
5. If u < P(X = x\*),

accept x\* as the new xi

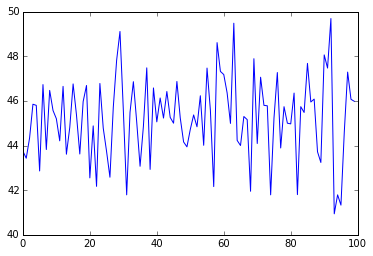
1. Update T and L with new xi
2. Return the list of xi and the corresponding T, from which find the optimizer



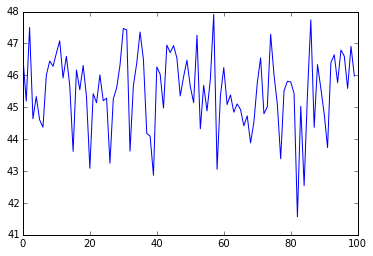
The plot with salesman annealing is as above. From the plot, as the number of acceptances increases, the distance first increases then decreases and finally wanders at a relatively low level. Also notice that at ranges 100 to 150, the decrease is rapid and tremendous.

**Part c: Uncertainties**

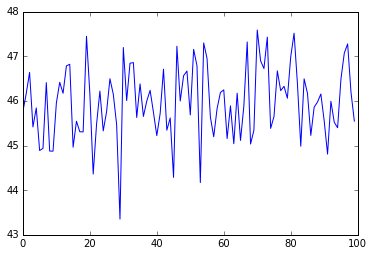
**1. Sigma p ¡Þ N¦Á**

****

N = 100 ratio = 0.8239

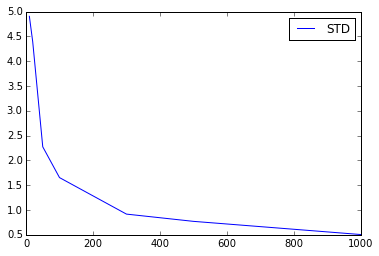


N=200, ratio=0.8541



N=500 ratio = 0.9095

**2. Optimization and uncertainties**

****

When repeating the individual cascading for different N’s, we can see that as N gets larger, the standard deviation decrease, making the measurement of reviewers’ influence on the whole network more accurate. So this is a factor that may bring in uncertainties.

**Part c: Parallelism for better performance**

1. **Rewrite influence function for parallel computing**
2. Approach A

Instead of dividing the NC\_Graph into different partitions and conducting BFS on partitioned nodes, our first intuition is to parallel the computation in influence function, that is, we send NC\_Graph to each worker using RDD.broadcast and each worker performs cascade functions multiple times (N/ partition number) to compute nodes’ influence, and we reduce the resultsby computing average to get the final influence of the initial node set.

The codes below shows the core part of our parallel algorithm:

**def** influence\_function(N, init\_nodes, partition\_num):

#Broadcast NC\_digraph to each nodes  
 nodes\_set\_broadcast = sc.broadcast(NC\_digraph)

# Create RDD using initial nodes for map functions  
 activated\_num\_rdd = sc.parallelize([init\_nodes]\*N, partition\_num)

# Get the influence by sending the cascade function call to all the workers and reduce by averaging returned results.  
 activated\_num = activated\_num\_rdd.map(**lambda** x: cascade(x, nodes\_set\_broadcast)).reduce(**lambda** x, y: (x+y)/2.0)

**return** activated\_num

Performance evaluation:

By parallelize the cascade function call, we are able to reduce the overall running time of greedy algorithm by 2/3.

1. Approach B

We are also considering partition the NC\_Graph to different RDDs to conduct BFS algorithm, however, considering the number of iterations performed in influence function and greedy algorithm, this approach could result in considerable overhead in communication as well as shuffle, so we decide not to further implement this approach at this stage and save it as one of our future backup plans.