Assignment 6

# Information

In this assignment, you will implement two simple Influence Maximization algorithms from scratch. The idea of these algorithms is to choose a set S of k nodes to activate at the beginning of a diffusion process, such that the total number of nodes activated at the end of the process is as large as possible. We will refer to the set of nodes activated by node u as Xu. You will then run some tests with your algorithm, applying it to different types of graphs, to see how well it performs.

We will be working with a model called the Independent Cascade (IC) model. In this model, we represent the influences (activation) of nodes based on probabilities in a directed graph. Here is how the process works:

* Input: a directed finite graph G = (V,E), and a set S of nodes that starts in an *active* state
* Label all nodes not in S as *inactive*
* Label each edge (u,v) *active* and give it probability pu,v
* While there are still potential activations:
  + For each *active* node u:
    - For each *inactive* neighbor v of u:
      * If edge (u, v) is *active*, then change v’s state to active with probability pu,v.
      * Set edge (u, v) to *inactive*, regardless of whether v was activated
      * **NOTE**: Each edge fires only once. If u and v are both active and link to w, it does not matter which tries to activate w first. Both u and v can have their own chance to activate w.

This model has already been implemented for you in the class ICModel. Your job will be to find sets of influential nodes. The complication you must overcome is introduced by the activation probability pu,v. Since there is randomness involved in the activation process, each active node will activate a possibly different set of neighbors each time the experiment is run from the same initial conditions (e.g. same initially active nodes, same edges, etc.). Therefore, there is no deterministic way we can decide which nodes will reach the largest number of others.

In situations like this, the solution is to run the same experiment multiple times from the same initial conditions, and average the results. This is the approach we will take for one of our functions we will design to help us find the most influential nodes in the network.

First, go to the [**Github Classroom for Assignment 6**](https://classroom.github.com/a/MFmT-77U). There you will find a starter code file that has quite a bit already implemented for you. Below is a description of what is in the file:

**General functions**

The file begins with several functions that are generally useful for this assignment.

* union(list1, list2): A function to take the set union over two lists.
* mean(mylist): A function to calculate the arithmetic average of a list.
* coin\_flip(prob\_of\_heads): A function to return True with the given probability, and False all other times.
* get\_directed\_caveman\_graph(m, n, p): A function to create a directed version of the Relaxed Caveman Graph we’ve seen before.
* get\_directed\_small\_world\_graph(n, k, p): Same as above, but for small world graphs.
* update\_plot(fig, ax, pos, ic, isdone=False, speed='lo'): Updates the data plot as a simulation is running.
* run\_simulation(ic, animate=False): Runs a simulation to completion, animating if requested.
* get\_k\_random\_nodes(ic, k): This function returns k randomly selected nodes.

**ICModel class description**

Next, there is a class definition for our Independent Cascade model that we will use to run our simulations.

* \_\_init\_\_(self): Creates the model and containers to keep track of the active statuses of all nodes and edges.
* set\_initial\_node\_status(self): Initialize all nodes contained in self.initial\_nodes to True.
* activate\_nodes(self, nodes): Reset the value of self.initial\_nodes to nodes, and then activate them.
* set\_initial\_edge\_status(self): Initialize all edges to True, and set their activation probabilities.
* reset(self): Reset all nodes and edges to their original activation status.
* get\_num\_activated(self): Get the number of currently active nodes.
* get\_activated\_nodes(self): Get the list of currently active nodes.
* is\_done(self): Checks to see if any more activations are possible.
* update(self): Update activation statuses of all nodes and edges for a single step.
* Given an instance of this class called ic, you can examine the active state of node u by using ic.node\_stats[u]['active'].
* Given an instance of this class called ic, you can examine the active state of edge (u, v) by using ic.edge\_stats[u, v]['active'].
* Given an instance of this class called ic, you can examine the activation probability of a given edge by using ic.edge\_weights[u, v]['active'].
* Given an instance of this class called ic, you can examine or modify the networkx graph object underneath by referencing ic.graph.

**Other Functions**

Finally, there is a section of other functions you will need to complete this work. Note that all the functions described above are completed already; there is no work to do for them. The following three functions will need to be completed.

* get\_average\_influence\_set\_size(ic, node, numreps=20): Takes as input an independent cascade object, a single node number, and (optionally) a number of repetitions. This function needs to run numreps different simulations on the same input graph, and count the number of neighbors activated by node at the end of each. It will then return the average number of activated neighbors.
* get\_influenced\_neighbors(ic, node, numreps=20): The same as above, but in this case you will run numreps simulations, and keep a running list of all the vertices activated by node over all the repetitions. You can use the union function above to avoid including the same other node multiple times. The function should return the list of all nodes activated by node. Using this function, an “influential” node will be one that can reach a large number of others. For example, if you run two simulations to see which nodes are activated by node 0, and the first simulation results in the nodes [1, 2, 3] being activated, and the second results in [2, 3, 4] being activated, then the function should return [1,2,3,4].
* get\_k\_influential\_nodes\_a(ic, k): This function needs to use `1 described above to estimate the influence of every node in the graph. It should then find the k most influential of those nodes and return them.
* get\_k\_influential\_nodes\_b(ic, k): This function needs to use get\_influenced\_neighbors described above to estimate the influence set of every node in the graph. It should then find the k most influential of those nodes and return them.

# Tasks

Your tasks for this assignment are to implement two different functions to assess each node’s scope of influence in the network, and two to choose the k most influential nodes. You will then compare results across different approaches and judge how well the algorithm works.

1. Implement the function get\_average\_influence\_set\_size. This function should run for a set number of repetitions, and calculate the total number of nodes activated by a single active node for each one. The function should then return the average number of activated nodes.
2. Implement the function get\_influenced\_neighbors. This function should run for a set number of repetitions, and calculate the set of nodes activated by a single active node for each one. The function should then return the union over all sets of activated nodes.
3. Implement the function get\_k\_influential\_nodes\_a. This function should use get\_average\_influence\_set\_size to measure the potential influence of all nodes in the network, and then return the set of k nodes that are estimated to have the most influence.
4. Implement the function get\_k\_influential\_nodes\_b. This function should use get\_influenced\_neighbors to measure the potential influence sets of all nodes in the network. To accomplish this, the function should proceed as follows:

* Initialize an empty list S to hold the influential nodes.
* Initialize another empty list T to hold the total activation set for all nodes in S - in other words, the union over all activation sets Xu for each node u in S.
* Calculate the activation sets Xu for each node u.
* While |S| < k:
  + Find the node u for which T U Xu is the largest. In other words, find the node that *adds the most other nodes* to the set T of all potentially activated nodes.
  + Set T = T U Xu
  + Set S = S U {u}
* Return S

| You are given k = 5 by default, and a scale free graph to work with. Use the get\_k\_random\_nodes function to get your set of seed nodes. Run the code to make sure everything is running as expected. How many nodes in the network end up getting activated? |  |
| --- | --- |
| Once you have implemented the functions for Tasks 1 - 4, run the code again, but using get\_k\_influential\_nodes\_a to find your seed nodes. How many nodes in the network end up getting activated? |  |
| Paste a screenshot of your network after running it with the most influential nodes found by your function. |  |
| Run the code again, but using get\_k\_influential\_nodes\_b to find your seed nodes. How many nodes in the network end up getting activated? |  |
| Paste a screenshot of your network after running it with the most influential nodes found by your function. |  |

Hopefully at this point you have both methods of finding influential nodes working properly. The scale free graph is a great tool to see the effectiveness of your algorithms. Of course there are many different algorithms available to find a set of influential nodes with the same intentions, but these are straightforward and can be quite effective. Now we will use these to examine the effects of a few different parameters.

First, let’s just compare the three methods for selecting seed nodes against each other for different styles of graph.

1. Set up a segment of code that will create a scale free graph using the function call provided, use that to create an ICModel object (with activation probability 0.5), and then run 10 simulations on your ICModel object for each of the three node-finding algorithms (30 simulations in all). Store the total number of activated nodes for each simulation (categorized by algorithm). Keep in mind that you will *need to use the exact same graph for all simulations; you should not be creating new graphs in between individual simulations or when switching to a different algorithm*.

| Create a box-and-whisker plot showing the performance distribution you observe. Each box should describe the 10 runs for one algorithm (one box for random selection, one box for average number of activations, one box for largest possible number of activations). Paste the plot to the right. |  |
| --- | --- |
| Do the same task, but using the random graph as your base network. |  |
| Do the same task, but using the caveman graph as your base network. |  |
| Do the same task, but using the small world graph as your base network. |  |
| Describe the outcomes of these simulations. Write several sentences addressing:   * Which algorithm performs best on each type of graph? Does one of the algorithms do better on most/all networks? * How did random selection compare with the other methods? * How much variation was there between methods? |  |
| When using your two newly implemented node-finding algorithms, we continually use the same graph (same edges over the same nodes) as our starting point instead of creating new randomly generated graphs. Why is it important to do this? |  |
| If we did use completely new graphs each time, what effect would that have on the quality of the algorithm? |  |

1. Next, you will set up a similar segment of code to the last task, but this time we will examine the effect of the parameter k. This code should start with the scale free graph, the random node selection function, and should run 10 simulations for each value of k from 1 to 10.

| Create a box-and-whisker plot showing the performance distribution you observe. Each box should describe the 10 runs for one value of k. Paste the plot to the right. |  |
| --- | --- |
| Do the same task, but using the random graph as your base network. |  |
| Do the same task, but using the caveman graph as your base network. |  |
| Do the same task, but using the small world graph as your base network. |  |
| Describe the outcomes of these simulations. Write several sentences addressing:   * Which value of k performs the best? * Was there a lot of variation in the effect of k between different graph structures? * Is there a point of diminishing returns? In other words, is there a number of seed nodes past which it doesn’t make sense to add more? |  |
| Do the same tasks as above, but this time change the activation probability for your edges to 0.25 instead of 0.5. You do not have to paste all of these box-and-whisker plots as you did before.  Describe the outcomes of these simulations. Write several sentences addressing:   * Which value of k performs the best? * Was there a lot of variation in the effect of k between different graph structures? * Is there a point of diminishing returns? * How did changing the value of the activation probability affect your results in comparison to the higher-probability condition? |  |

1. The last task is yours to define. So far, we have looked at differences between graph structures, algorithms for finding seed nodes, different numbers of seed nodes, and different activation probabilities. Now, decide on a question you would like to investigate (a little bit, of course). This question can be about changing graph generation parameters - e.g., you might want to look at varying the number of neighbors in small world graphs specifically - or you may want to test more parameter values than we did above, or you may want to see what happens when networks become larger and/or sparser. Feel free to design your own question. You might even want to implement another node-finding algorithm! That is perfectly fine. Design a small experiment, and describe it below. Provide the question you will be asking (What happens in a caveman graph when k = the number of communities? Does the algorithm choose nodes from the same community, or spread them out well?), what tests you are running, and your results. Feel free to use any visualization you like (scatter plots, line plots, bar graphs, and box plots are ones you already have example code for, but use anything you like!).

| Question |  |
| --- | --- |
| Tests |  |
| Results |  |

Once you have completed all of the tasks above, submit a link to this document to Moodle (make sure Sharing is turned **ON**), and commit and push your code to Github.