ECS 289 / MAE 298, Network Theory and Applications
Spring 2014
Solution to Advanced Problem Set # 3 (aka 3b)
Topic: Configuration model random graph

1 Building a configuration model random graph

Let $\mathbf{k} = (k_1, k_2, k_3, \cdots, k_N)$ be a vector of N non-negative integers such that $\sum_{i=1}^{N} k_i$ is an even number. An instance of the configuration model random graph with degree sequence $\mathbf{k}$ can be obtained as follows.

- Make sure that $\sum_{i=1}^{N} k_i$ is an even number (give an error if not).
- Create an undirected graph containing N nodes and no edges.
- Create a vector (or list or multiset...) such that, for each $1 \leq i \leq N$, it contains k_i copies of i. This object will hereafter be known as the "stub list". *Example: The vector* $(1, 2, 3, 4, 5, 5, 6, 6, 6, 7, 7, 7)$ *is a valid stub list in the case* $\mathbf{k} = (k_1, k_2, k_3, k_4, k_5, k_6, k_7) = (1, 1, 1, 1, 2, 3, 3)$.
- Sample uniformly at random one element from the stub list; call that element i and remove it from the stub list. Sample uniformly at random another element from the (now shorter) stub list; call that element j and remove it from the stub list. Add an edge between node i and node j in the network. Repeat this step as long as the stub list is not empty.

The resulting network will be a random graph with degree sequence $\mathbf{k}$. For the purpose of this problem, we do not worry about repeated edges (i.e., more than one links between two nodes) and self loops (i.e., a node with an edge to itself).

(a) Write a computer implementation of this algorithm.
See code of function `configuration_model_from_degree_sequence` *at the end of this document.*

(b) Using the degree sequence $\mathbf{k} = (1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3)$, generate a configuration model random graphs and display the result as a figure.
See Fig. 1

(c) Let $\mathbf{n} = (n_0, n_1, n_2, \cdots, n_{k_{\max}})$ be a vector such that n_k represents the number of nodes of degree k. For example, $\mathbf{n} = (0, 8, 4, 2)$ corresponds to the $\mathbf{k} = (1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3)$. Write a function that receives $\mathbf{n}$ as an input and returns $\mathbf{k}$.
See code of function `degree_count_to_degree_sequence` *at the end of this document.*

(d) Let $\mathbf{p} = (p_0, p_1, p_2, \cdots, p_{k_{\max}})$ be a vector such that p_k denotes the probability of having a node of degree k. Write a function that receives $\mathbf{p}$ and N (number of nodes) and returns $\mathbf{k}$.
See code of function `degree_distribution_to_degree_sequence` *at the end of this document.*

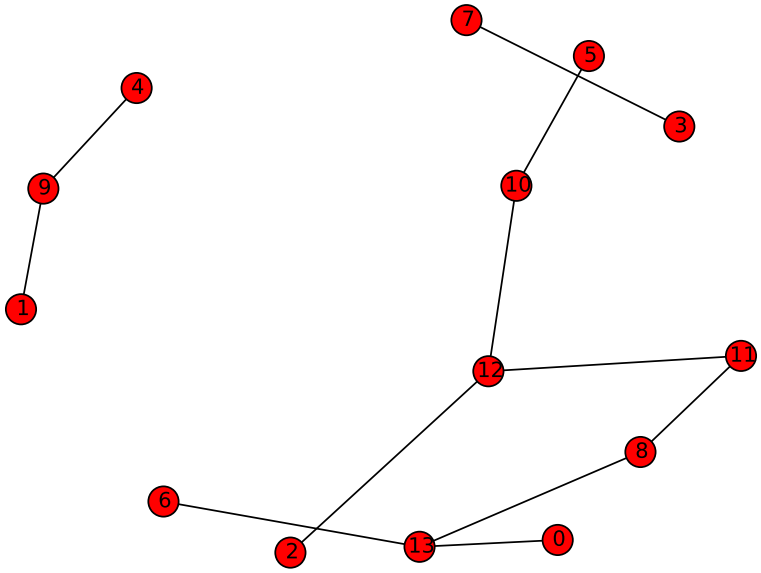

Figure 1: Output for Problem 1(b); results may vary. In particular, the algorithm may create self-loops and/or repeated edges which may not be displayed depending on plotting libraries and/or parameters.

2 Percolation and spreading

Consider the three functions described below.

`percolation`:

- Receive as input a graph G and a real number p such that $0 \leq p \leq 1$.
- Make a graph with no edges and as many nodes as G has; call this graph without edges G'.
- Iterate over all the edges of G. Suppose the current edge is between nodes u and v. Get a random number uniformly distributed in the interval $[0, 1)$. If that number is lower than p, add in G' an edge between nodes u and v.
- Return the graph G'.

Hence, each edge of G has probability p to be present in G'.

`spreading`:

- Receive as input a graph G, a node index v, and a real number T such that $0 \leq T \leq 1$.
- "Mark" all nodes as "unreached" by one of the following two methods:
  - Create a vector of bool called `is_reached` containing as many entries as there are nodes in G. Initialize all its entries to "False"; **OR**

  - Give to each node in G the bool "node property" `is_reached`. Initialize them all to "False".

- Create an empty vector (or other appropriate container) of indices; call it `unresolved`, and place v in it.
- Set an integer variable `number_reached` with value 1.
- Repeat the following as long as `unresolved` is not empty. Get in u the value of an element of `unresolved`, and remove said element from `unresolved`. If u is marked as unreached (i.e., if `is_reached` is "False" for that node), do the following:
  - Increment `number_reached` by one.
  - Mark u as reached (i.e., set `is_reached` to "True" for that node).
  - For each neighbor w of u, generate a random number uniformly distributed in the interval $[0, 1)$. If the number is lower than T, add w to `unresolved`.
- Return `number_reached`.

`component_size`:

- Receive as input a graph G and a node index v.
- Call your `spreading` function for the graph G, the node index v, and using $T = 1$.
- Return the `number_reached` returned by the aforementioned function.

(a) The fifth bullet of `spreading` does not specify *which* element of `unresolved` should be removed to become u. Does the outcome `number_reached` depend on this choice? Why?
Notice that the graph structure remains unaffected throughout the process. Moreover, from a statistical viewpoint, the internal state of the random number generator is irrelevant. Hence, the only "states" that matter are the value of `number_reached`*, the content of* `unresolved`*, and the different* `is_reached` *for each node.*
When popping out an element of `unresolved` *to be used as* `u` *(hereafter known as "resolving* `u`*"), two different behaviors are possible. If* `is_reached` *is* `False` *for that* `u`*, then there is no further impact besides the fact that this instance of* `u` *is no longer in* `unresolved`*. If* `is_reached` *is* `True` *for that* `u`*, then the outcome does not depend on the specific value of* `number_reached` *(i.e., it is only incremented), nor on the remaining content of* `unresolved`*, nor on the value of* `is_reached` *for other nodes than* `u`*. Hence, the only thing that may affect the resolution for a given* `u` *are past resolutions for that same* `u`*. However, all resolutions for* `u` *beyond the first have no collateral impact because the first one marks* `is_reached` *as* `True`*. Hence, the choice of which element of* `unresolved` *is to be resolved at a given time is arbitrary.*

(b) Explain why the value returned by `component_size` corresponds to the size of the component to which v belong.
The connected component to which v belong is the maximal set of nodes C such that there exists a path from v to any node in C. We will show that the set C corresponds to the nodes for which `is_reached` *is* `True` *at the end of the execution, so the size $|C|$ of that component is thus* `number_reached`*.*

We first show that all nodes in C are reached by the algorithm. Clearly, $v \in C$ is reached by the algorithm. Because $T = 1$, all the neighbors of a node reached by the algorithm are also reached by the algorithm. Suppose, for contradiction, that $x \in C$ is not reached by the algorithm. The fact that $x \in C$ implies that there exists a path from v to x. Notice that the n-th node of that path is neighbor to the $(n-1)$-th node of that path. Because the first node of that path (i.e., v) is reached, we can show by mathematical induction that x is reached, a contradiction.
We now show that only nodes in C are reached by the algorithm. Suppose, for contradiction, that $x \notin C$ is reached by the algorithm. Thus either x is v (in which case it is part of C, a contradiction), or x is the neighbor $w = x$ of a node $u \in C$. But $u \in C$ means that there is a path from v to u, and there is a path from u to x because x is u's neighbor. There is thus a path from v to x, which means that $x \in C$, a contradiction.

(c) Let G' be a graph returned by `percolation` (with parameters G and p). Show (by hand) that calling `spreading` with the parameters G', v and T is statistically equivalent to calling `spreading` with the parameters G, v and pT. From this result, deduce a relationship between `spreading` and the size of the component to which v belong in a graph returned by `percolation`.
The third tick of the fifth bullet of ***`spreading`*** *above, says "For each neighbor w of u, generate a random number uniformly distributed in the interval $[0, 1)$. If the number is lower than T, add w to* `unresolved`*." We now show that, for a given edge, only the first time that this test is performed may matter to the outcome of the process (i.e., the value of* `number_reached`*). Suppose that the test has already been done before for that edge using $u = a$ and $w = b$, which means that a is currently marked as reached. There are two possibilities for this test to be done again for that edge: either $u = a$ and $w = b$, or $u = b$ and $w = a$. The case $u = a$ and $w = b$ cannot happen again because a is already marked as reached. In the case $u = b$ and $w = a$, the outcome of the test does not matter: adding a to* `unresolved` *will only lead to it being silently resolved later on. There are no remaining cases, only the first time that the test is done for a given edge may matter. Note that the argument also holds in the presence of repeated edges and self loops.*
Now notice that the consequences of failing that test are the same as those of not taking the test in the first place. Hence, we may emulate the absence of an edge in G' by running the algorithm on G using pT instead of T, the factor of p accounting for the probability for the edge to be there. A possible problem with this approach is that whether the edge is or is not in G' ***does not change*** *during the simulation: the outcome of the first test and the one of any future tests for a given edge are correlated among themselves. However, we just showed that only the first of these tests matters, so this correlation is not a problem here.*

(d) Implement these three functions.
See the appropriate codes at the end of this document.

(e) We shall now use the code you have written up to this point to understand percolation/spreading on a random network with a given degree sequence. For a given graph, you will simulate spreading with a given probability of transmission (T) and compute the probability distribution for the total number of nodes reached starting from a node at random. The process is described below. Write code to do the following:

- Receive `number_simulations`, T, $\mathbf{n}$ as inputs.
- Initialize a vector `bin_outcome` to store the result. This vector should have as many entries as there are nodes in the network, and all these entries should initially be zero. (Reason: the spreading process could reach at most N nodes.)

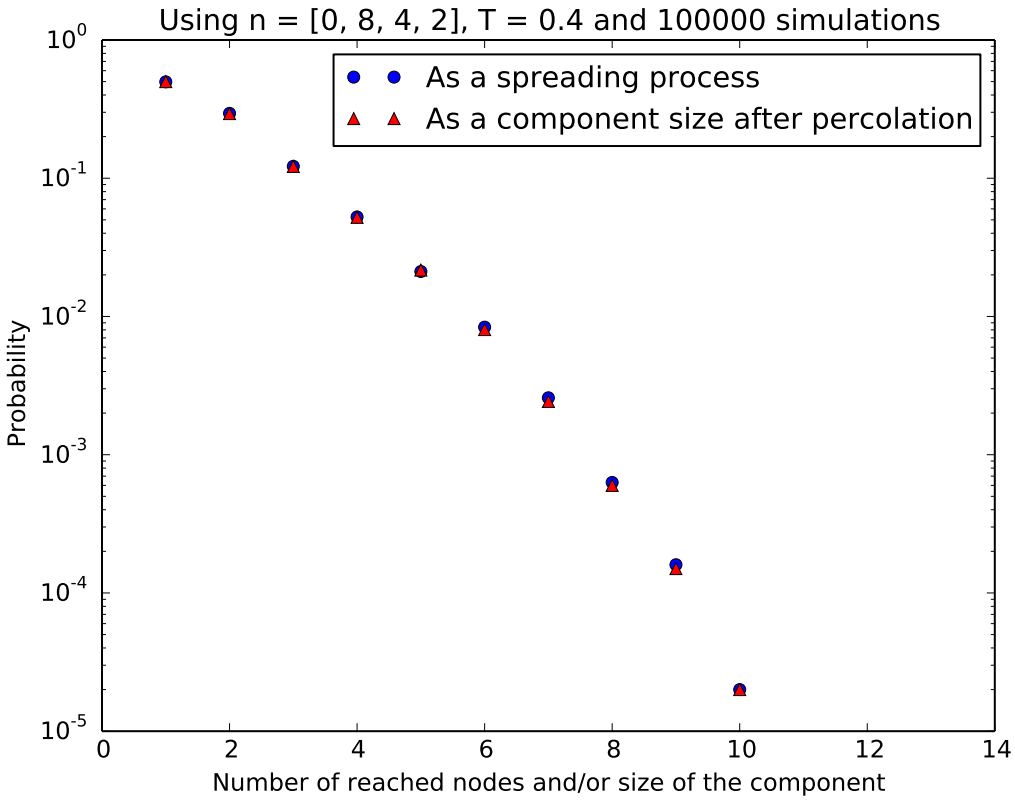

Figure 2: Output for Problem 2(f); results may vary.

- Repeat the following steps `number_simulations` times:
  - Create a configuration model random graph with $\mathbf{n}$ as the input.
  - Select a node v uniformly at random.
  - Do $\mathcal{A}$ or $\mathcal{B}$ (See below). Store the result in s.
  - Increment the sth entry of the vector `bin_outcome` by 1.
- Normalize the vector `bin_outcome` such that component s gives the probability of the process spreading to s nodes starting from a random node. This is the result we are interested in.

$\mathcal{A}$: Call `spreading` starting at node v.
$\mathcal{B}$: Call `percolation`, and then `component_size` for node v.
See code of function `bin_many_simulations` *at the end of this document.*

(f) Run the above process for $\mathbf{n} = (0, 8, 4, 2)$ and $T = 0.4$. Report the result.
See Fig. 2.

Content of homework_3b.py

```
"""
This is part of the answer to Homework 3b.
"""

import random as rnd
import matplotlib.pyplot as plt
from networkx import Graph, draw
from bisect import bisect_left
from itertools import repeat

def configuration_model_from_degree_sequence(k):
    """
    Create a configuration model random graph with degree sequence `k`.

    This is the answer to Problem 1(a).
    """
    #*** Some checks and preparation. ***
    #Make sure that the sum of the degree sequence is an even number.
    assert( (sum(k) % 2) == 0 ) #Reminder: the operator "%" means "modulo".
    #Make an empty graph.
    G = Graph()
    #Add as many nodes as there are entries in k.
    G.add_nodes_from(range(len(k)))
    #Make a list of stubs.
    stubs = [ i for i in range(len(k)) for _ in repeat(None,k[i]) ]

    #*** Create the edges by randomly pairing stubs. ***
    #Method based on shuffling the stubs. See below for alternative.
    rnd.shuffle(stubs) #Shuffle the stubs.
    for (i,j) in zip(stubs[0::2],stubs[1::2]): #Pair the stubs.
        G.add_edge(i,j) #Add an edge between the corresponding nodes.
    #More straightforward method: pick the stubs two by two.
    #(Uncomment the following lines and comment the previous ones.)
#     while stubs: #Loop as long as any stub remains.
#         i = stubs.pop(rnd.randint(0,len(stubs)-1)) #Pick the source.
#         j = stubs.pop(rnd.randint(0,len(stubs)-1)) #Pick the target.
#         G.add_edge(i,j) #Add an edge between source and target.
    return G

def degree_count_to_degree_sequence(n):
    """
    Obtain a degree sequence given the `n`, the count of each degrees.

    This is the answer to Problem 1(c).
```

```
    """
    return [degree for degree in range(len(n)) for _ in repeat(None,n[degree])]

def reinventing_accumulate(to_be_accumulated):
    """
    Does the same as Python 3.2+'s `itertools.accumulate()`.

    Used in `specified_distribution`.
    """
    total = 0 #Initialize to 0.
    for x in to_be_accumulated: #Loop over the values to be accumulated.
        total += x #Accumulate this value.
        yield total #`yield` basically means "return this, but not done yet".

class specified_distribution:
    """
    A random distribution with explicitly specified probabilities.

    Used in the answer to Problem 1(d) below.
    """
    def __init__(self, p, outcomes=None, source_of_randomness=rnd.random):
        """Initialize for a specified distribution `p`."""
        #Store the cumulative distribution.
        self.cdf = list(reinventing_accumulate(p))
        #The distribution `p` should be normalized.
        assert( abs(1.0-self.cdf[-1]) < 1e-14 )
        #Force the last entry of the cdf to take the value 1.0.
        self.cdf[-1] = 1.0 #(Could introduce bias of at most 1e-14.)
        #The default is to return the index of the corresponding entry in `p`.
        if outcomes==None:
            self.outcomes = list(range(len(p)))
        else:
            self.outcomes = outcomes
        #There should be as many outcomes as specified probabilities.
        assert( len(p) == len(self.outcomes) )
        #A nullary function returning a random number in the range [0.0, 1.0).
        self.random = source_of_randomness
    def draw(self):
        return self.outcomes[bisect_left(self.cdf,self.random())]

def degree_distribution_to_degree_sequence(N,p):
    """
    Obtain a degree sequence given `N`, the number of nodes in the network,
    and `p`, the degree distribution.

    This is the answer to Problem 1(d).
    """
```

```
    #Prepare `degree_urn` to return a degree according to `p`.
    degree_urn = specified_distribution(p)
    #Call `degree_urn.draw()` `N` times.
    k = [degree_urn.draw() for _ in repeat(None,N)]
    #Technically, `k` is an OK answer for the problem as formulated in
    #Homework 3b. However, we want a degree distribution that has an even
    #total degree, so lets enforce it!
    parity = sum(k) % 2 #If `parity==0`, then everything is fine already.
    while parity != 0:
        index_of_discarded = rnd.randint(0,N-1) #This guy will be booted out.
        replacement_value = degree_urn.draw() #This guy will replace him.
        parity = (parity-k[index_of_discarded]+replacement_value)%2 #Update.
        k[index_of_discarded] = replacement_value #Do the substitution.
    return k #Now it should be fine!

def percolation(G, p=1.0):
    """
    Obtain a graph where each edge of `G` has probability `p` to be present.

    This is part of the answer to Problem 2(d).
    """
    #Initialize an empty graph.
    G_prime = Graph()
    #Put in all the nodes of `G`.
    G_prime.add_nodes_from(G.nodes())
    #Loop over the edges of `G`.
    for (u,v) in G.edges():
        if rnd.random() < p: #Check if that edge should be added.
            G_prime.add_edge(u,v) #Yes! Do it.
    return G_prime

def spreading(G, v, T=1.0):
    """
    Count the number of nodes reached by a spreading process on `G`,
    starting at `v` and following edges with probability `T`.

    This is part of the answer to Problem 2(d).
    """
    #Mark all nodes as unreached.
    for u in G.nodes():
        G.node[u]['is_reached'] = False
    #Initialize the unresolved.
    unresolved = [v]
    #Initialize number_reached.
    #NOTE: There was a typo in the question; no penalty for initializing to 1.
    number_reached = 0
    #Repeat the following as long as there are unresolved left.
```

```
    while unresolved:
        u = unresolved.pop(-1) #Get the next unresolved.
        if G.node[u]['is_reached'] == False: #Is `u` already reached?
            number_reached += 1 #No! So increment the number of reached.
            G.node[u]['is_reached'] = True #`u` now reached, so mark as such.
            for w in G.neighbors(u): #Loop over `u`'s neighbors.
                if rnd.random() < T: #Do the following with probability `T`.
                    unresolved.append(w) #Add that neighbor to the unresolved.
    return number_reached

def component_size(G, v):
    """
    Obtain the size of the component of `G` containing node `v`.

    This is part of the answer to Problem 2(d).
    """
    return spreading(G, v, 1.0)

def bin_many_simulations(n, T, the_simulator, number_simulations):
    """
    Run `the_simulator` `number_simulations` times and bin the results.

    The third parameter must be such that `the_simulator(G,v,T)` calls the
    appropriate simulation.

    This is the answer to Problem 2(e).
    """
    #Initialize `res` to N+1 zeros (here we keep an entry for the outcome `0`).
    res = [0]*(sum(n)+1)
    #Translate `n` to a degree sequence `k`.
    k = degree_count_to_degree_sequence(n)
    #Main loop.
    for _ in repeat(None,number_simulations):
        #Make a configuration model random graph.
        G = configuration_model_from_degree_sequence(k)
        #Pick a node at random in G.
        v = rnd.choice(G.nodes())
        #Do the simulation and increment `res` according to the result.
        res[the_simulator(G,v,T)] += 1
    normalization = float(sum(res))
    return [ float(x)/normalization for x in res]

if __name__ == '__main__':
    #*** Answer to Problem 1(b). ***
    #Create the requested configuration model random graph.
    G = configuration_model_from_degree_sequence([1,1,1,1,1,1,1,1,2,2,2,2,3,3])
    #Draw it.
```

```
    draw(G)
    #Save it to a file.
    #(Here "0842" means 0 nodes of degree 0, 8 nodes of degree 1 etc.)
    plt.savefig("confmod_0842.pdf",format='pdf')
    plt.clf()

    #*** Simple test that the solution to Problem 1(c) works. ***
#     print(degree_count_to_degree_sequence([0, 8, 4, 2]))

    #*** Simple test that the solution to Problem 1(c) works. ***
#     print(degree_distribution_to_degree_sequence(10, [0.0, 0.2, 0.5, 0.3]))

    #***Part of the answer to Problem 2(f)***
    #Set some parameters.
    n = [0,8,4,2]
    T = 0.4
    number_simulations = 100000
    #Run the simulations.
    result_spreading = bin_many_simulations(n,T,spreading,number_simulations)
    combo = lambda G_,v_,T_: component_size(percolation(G_,T_),v_)
    result_percolation = bin_many_simulations(n,T,combo,number_simulations)
    #Make a plot of the results.
    x_axis_values = list(range(sum(n)+1))
    plt.semilogy(x_axis_values,result_spreading,'bo',
                 x_axis_values,result_percolation,'r^')
    plt.xlabel('Number of reached nodes and/or size of the component')
    plt.ylabel('Probability')
    plt.legend(['As a spreading process',
                'As a component size after percolation'])
    plt.title("Using n = " + str(n) + ", T = " + str(T) + " and " +
              str(number_simulations) + " simulations" )
    plt.savefig("probabilities_spread_percolation.pdf",format='pdf')
    plt.clf()
```