Detection Algorithm of False Accounts of a Social Network

Social graphs and random walks

Morgane AUSTERN – Thibault LAUGEL

04/05/2012

[Tapez le résumé du document ici. Il s'agit généralement d'une courte synthèse du document. Tapez le résumé du document ici. Il s'agit généralement d'une courte synthèse du document.]

Table des matières

Introduction

Since the creation of Facebook in 2006, the number of social networks skyrocketed: photo, video or message sharing… The principle is quite simple, namely to organize and facilitate sharing between people and organizations, through the intermediary of social interactions via the internet.

Fake accounts have represented a problem ever since these networks appeared. These false accounts have caused sometimes very little harm (creation of an account with a friend's name for instance). However, they sometimes aim to insult or damage the reputation of someone (more than a dozen of Facebook accounts are named "Nicolas Sarkozy" or "François Hollande", the new French president). But they can also have a commercial goal, as it was for Orangina : the brand was accused of having created many fake accounts in order to boost its own Facebook page frequentation.

Some measures were taken: today, most social networks allow every user to report any account he or she finds suspect. The social network security department then investigates, sometimes leading to suppressing the suspect account.

To prevent users from creating fake accounts, several methods could be used. The most common one is probably the use of a central authority system that would issue and check the identity of the network users at the creation of their account (for instance a system that would require users to register with their social security number), or a payment for each identity. Unfortunately, the central authority can easily be the target of attacks and the origin of system failures. Furthermore, requiring sensitive information or a payment in order to use a social network may scare away many potential users.

This explains how the idea of creating computer programs which would check the identity of a user, and determine if it is an honest user or a fake account appeared. This paper presents our version of the algorithm *SybilGuard*, a novel program created to crack down on fake accounts creation. First we will present the main mathematical ideas on which our algorithm relies on. Then we will detail our Python program that implements the algorithm. Finally we will analyze the results we obtained using this program, and the issues we met during its development.

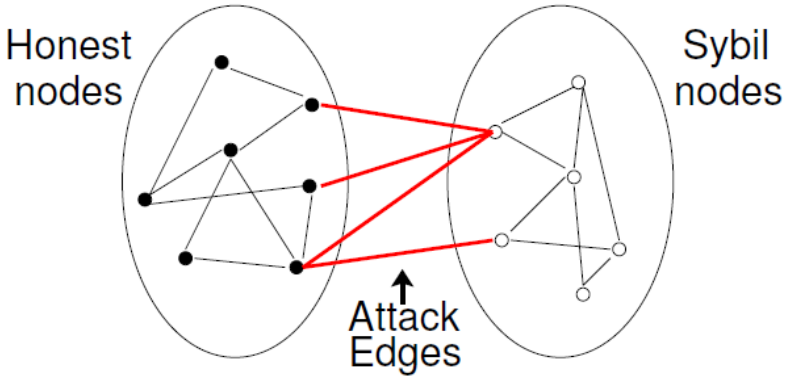
Social graphs and sybils – frame of the algorithm

1. Some definitions and assumptions

Many real-world situations can conveniently be described by a means of a diagram consisting of a set of points, some of them being linked with lines. Social networks can easily be represented using such diagrams: the points would thus represent people, or more precisely the accounts they use on the social network, with lines joining pairs of friends. The mathematical concept associated with these diagrams is called a *graph*, and in our case a *social graph*. Each point of the diagram will now be called a *node*, and the number of its friends the *degree of the node*. Numerically, a social graph is represented by an *adjacency matrix:* the size of this matrix is equal to the number of network users, and each of its coefficients $_{j}$ is equal to 1 if the users i and j are friends, and to 0 otherwise.

As we said in the introduction, *sybils* (fake accounts) are a threat for these networks. In a sybil attack, a network user creates multiple accounts using multiple fake identities. The link that may exist between a sybil and a honest user is called an *attack edge*. This algorithm is based on a particular design of the social network, as drawn below:

Fig 1 : Basic form of the social network



Indeed, studies have shown that the number of attack edges was independent of the number of sybil identities, but was limited by the number of relations between an honest user and a malicious one (i.e. linked to the sybils), the chances of detection of the sybils being more important as this number rises. This explains why the hypothesis, according to which the number of attack edges being limited, on which relies this algorithm is reasonable. Furthermore, we suppose that all sybils are linked together: this hypothesis can be justified by the fact that linking all the sybils together is the easiest way to increase the number of their friends without arising suspicion among the honest users.

2. Main idea of the algorithm

To detect the sybils, our algorithm relies on a special kind of random walk in the graph called *random* routes. In a usual random walk, each step k of the walk will only be determined when arriving to the step k- 1. Our random routes rely on pre-computed permutations that will associate

each node a to the another node b, that will be the following step of the route if it arrives to the node a. Considering a certain length *w*, each step of the route will thus be determined until it reaches the step *w*. Thus defined, these random routes will have some very useful properties: first of all, two random routes entering an honest node will always continue the route passing by the same other node (convergence property). Furthermore, the outgoing path will determine the incoming path (back-traceable property).

Considering these random routes, an important choice in our algorithm will be w, the length of the random routes. The value of *w* must be sufficiently small to ensure that a verifier's random route remains entirely within the honest region, with high probability. On the other hand, *w* must be sufficiently large to ensure that honest nodes routes will intersect with high probability. The length of these random walks will be fixed according the following:

> **THEOREM1**. *For any connected and non-bipartite social network, the probability that a length-w random walk starting from a uniformly random honest node will ever traverse any of the g attack edges is upper bounded by* $\frac{gw}{n}$. *In particular, when* $w = O(\sqrt{nlog(n)})$ *and* $g = o(\frac{\sqrt{b}}{\log(n)})$ *, this probability is o*(1).

> **DEFINITION:** *a bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V; that is, U and V are independent sets.*

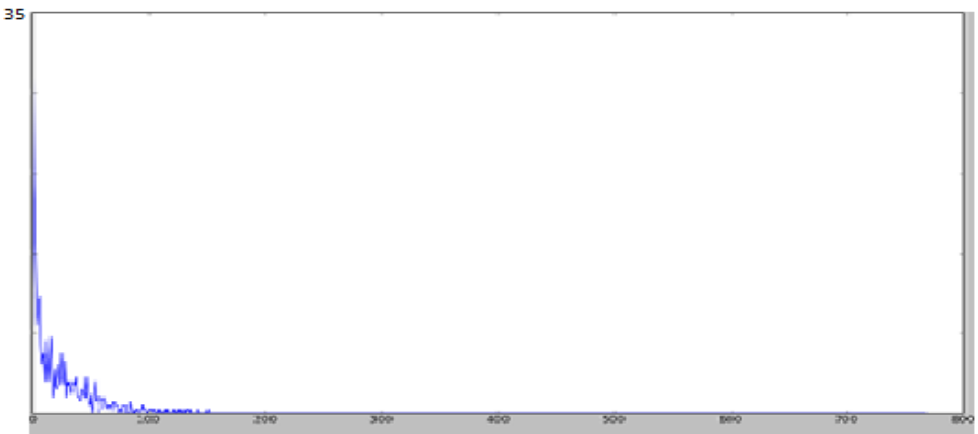
> It has been proven that this definition was equivalent to say that the graph was bipartite if it didn't have any cycle with an odd number of edges.

> We admitted the proof of this theorem in our study.

The goal of the algorithm is to compare the intersections between random routes. Indeed, due to the particular form of the social graph (Fig.1), it appears that random routes initiated in the *honest nodes zone* will remain within this region with a high probability. For instance, let's consider that a *verifier node* wants to check whether another node is a sybil or not. Using the algorithm, he may only accept this last node if its random route intersects with the verifier's random route *often enough*. This right number of intersections is thus a criterion that will allow us to determine if a network user is a sybil or not. To create a program that will be able to tell whether a certain node is sybil or not, we have decided to determine a certain limit-value of this criterion that would separate the sybils and the honest nodes, using an empirical method based on an adjacency matrix at are disposal, namely the social network of the American university Caltech.

Indeed, we used this matrix to model a function associating to each node degree the number of nodes that had this degree in the social network. The corresponding function's graphical representation for Caltech is:

Fig 2: variation in the number of nodes according to the nodes degree (Caltech)



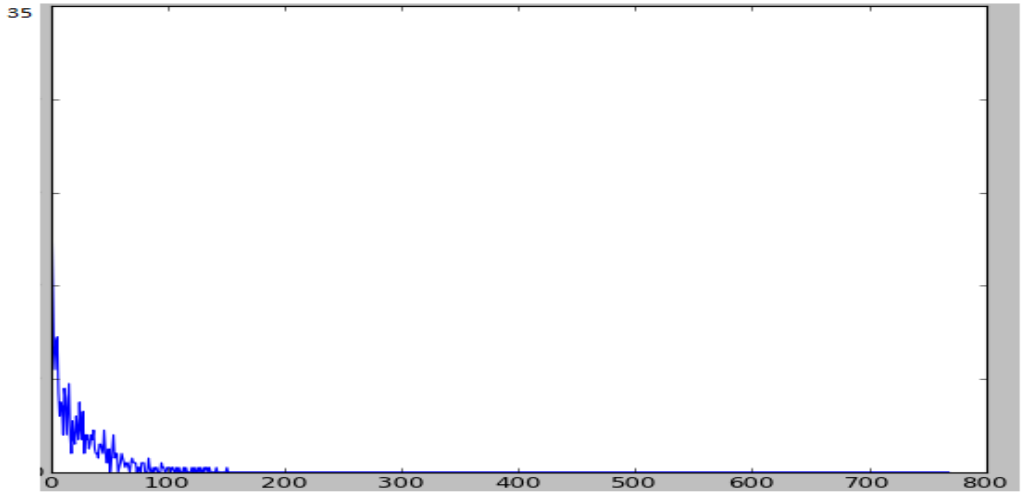
This is the modeling function we will use :

$$f : x \mapsto e^{-0.75\,log(x(1+e^{\frac{-log(0.75/35)}{x}}))} \quad (1)$$

The choice of this function will be explained in more details later, in the part entitled “Modeling the function” in the “Difficulties met” section.

Using this function, we were thus able to compute another random adjacency matrix, which had a similar size, number of nodes, and average node degree.

Fig 3 : variation in the number of nodes according to the nodes degree (new random adjacency matrix)



We now had to generate a sybil attack. Following our hypothesis of a limited number of sybils and attack edges, we found it plausible that these two numbers followed exponential probability distributions. Once the sybils were randomly linked to our adjacency matrix, we finally had to study the number of intersections between verifier nodes and a sybil nodes several times.

Details of the algorithm – Python programs

1. Construction of the detection algorithm

We first constructed the matrix composed of each person's friends list: the i^{th} row is a vector which k first terms are the friends of the node i , k being the degree of this node. The other coefficients are equal to -1.

```
def friends(A):
  n=len(A)
  B=[[-1 for i in xrange(n)] for j in xrange(n)]
  t=0
  for i in xrange(n):
    for j in xrange(n):
      if (A[i])[j]==1 and i<>j:
        (B[i])[t]=j
        t=t+1
    t=0
  return B
```

We must now create a program that would return de degree of the node of the social graph: it simply consists in counting the number of each person's friends. Then, we calculated the average node degree of the network. The corresponding programs are:

```
def degre (A,i):
  t=0
  B=friends(A)
  while (B[i])[t]!=(-1) and t<n:
    t=t+1
  return t

def degremoyen(A):
    n=len(A)
    s=0
    for i in xrange(n):
        s+=degre(A,i)
    return float(s)/float(n)
```

Following the construction of our algorithm, we must now compute the permutations for each node of its list of friend (tablederoutage), and thus associate each node to its transformation by the permutation (tablederoutage2). A program has also been written to print the image of a given node by these permutations at a certain step (next2). In these programs, we use the lines "print(100*float(i)/float(n))" to print the progress during the execution of the function. These functions will be used to build the random routes. The corresponding programs are:

```
def tablederoutage (A):
```

```
 B=friends(A)
n=len(B[0])
t=[[B[k][l] for k in xrange(n)] for l in xrange(n)]
for i in xrange(n):
   print(100*float(i)/float(n))
   d = degre(A,i)
   for j in xrange(d):
      k = random.randint(0, d - j)
      t[i][j],t[i][k+j] = t[i][k+j],t[i][j]
return t

def tableroutage2 (A,B,C, node):
   d=degre(A,node)
   RT=[[0 for i in xrange(d)] for j in xrange(2)]
   for i in xrange(d):
      (RT[0])[i]=(B[node])[i]
      (RT[1])[i]=(C[node])[i]
   return RT

def next2(B, RT, friend):
   n=len(RT[0])
   j=0
   while  j<n and B[friend][j]<>friend:
      j=j+1
   if j>n-1:
      return -1
   else:
      return (RT[friend])[j]
```

These permutations are then used to construct a matrix, in which every coefficient $a_{i,j}$ corresponds to the number of the final step of a route starting at the node *node* and going to the node j, after i permutations : this is the construction of the random routes. The corresponding program is:

```
def  witness(A,B,RT,node,w):       #B=friends(A). RT=tableroutage(B). w represents the
#desired length of the random routes.
   n=degre(A,node)
   l=len(A)
   WT=[[0 for h in xrange(n)] for f in xrange(w)]
   for i in xrange(n):
      (WT[1])[i]=(B[node])[i]
   for i in xrange(2,w):
      for j in xrange(n):
         WT[i][j]=next2(B,RT,WT[i-1][j])
    Return WT
```

According to the algorithm, we must now create the program that would compare two random routes, namely study their intersections. The corresponding program is:

```
def comp (A,B,RT,V,S,W):    #V and S are the first nodes of the random routes we want to
  cou=witness(A,B,RT,V,W)  #compare
  cou2=witness(A,B,RT,S,W)
  d=0
  n=len(cou[0])
  p=len(cou2[0])
  for i1 in xrange (0,n):
    c4=0
    for j1 in xrange(W):
      for i2 in xrange(p):
        for j2 in xrange(W):
          if (cou[j1])[i1]==(cou2[j2])[i2]:
            c4=c4+1
    if c4>0:
      d=d+1
  return d
```

2. Defining a detection criterion

We are now able to generate and compare random routes, and thus, according to our algorithm and its hypotheses, to detect sybils. In order to complete the algorithm, we now need to define a criterion which will make us able to determine if a network user is a sybil or not. To do so, we have to generate a plausible adjacency matrix. We then have to study for each node degree the number of nodes in our matrix Caltech that have this degree. The first of the following functions, called *statistique* returns a vector *v*, which i^{th} coordinate is equal to the number of nodes that have a degree equal to i. The rest of the following programs have been written to plot this vector. Doing this, we aim at using this thus defined function to determine a probability distribution, that would able us to compute a node given his degree. We could thus be able to randomly build a matrix C that would have the same “profile” as our matrix A (Caltech). The corresponding programs are:

```
def statistique (A):
    n=len (A)
    v=[0 for i in range (0,n)]
    for i in xrange (0,n):
        v[degre(A,i)]+=1
        print(100*float(i)/float(n))
    return v

import numpy as np
import matplotlib.pyplot as plt
print statistique (F)
from pylab import *
```

```
#To plot the graph (Fig. 2)
n=len(F)
t =np.zeros(n)

y=statistique(F)
v=np.zeros(n)
for i in xrange(n):
   v[i]=y[i]
   t[i]=i
plot(t,v)
show()
```

Given a function *g,* the following program returns a vector which i^{th} coordinate is equal to the sum of the images of the i first integers by *g*. This program will be applied to a function created from our matrix A. Of course, f is not exactly a mathematical function since it only applies to integers. However, this can be easily fixed:

```
def somme (g,n):
   c=0
   v=[0.0 for i in range(n)]
   for i in xrange(1,n):
      v[i]=g(i)+v[i-1]
   return v

#definition of the function modeling the number of nodes by degree
def f(i):
   if i==0:
      c=0.0
   else:
      c=0.25*1000*(math.exp(-0.75*math.log(i*(1+math.exp(-math.log(0.75/35)/i)))))
   return c
n=len(F)
v=somme(f,n)
```

This function f can now be used to create our new adjacency matrix. The following program thus computes a random node degree, according to the probability distribution associated with the function f. (1)

```
def prob(v):
   n=len(v)
   c=n
   d=random.random()
   for i in range(n-1):
      if  ((v[i])/v[n-1])<=d<((v[i+1])/v[n-1]):
         c= i
```

```
    return c
```

We are now able to compute a random adjacency matrix, which has the same size as Caltech, and a similar "number of nodes of a given degree" profile, using the previous function. The corresponding program is:

```
def matricerandom2(A,v):
  n=len(A)
  B=[[0 for i in range(n)]for j in range(n)]
  for i in xrange(n):
    h=prob(v)
    print(100*float(i)/float(n))
    g=[0 for s in range(h)]
    for f in range(h):
      s=random.randint(0,n)
      if s<>i :
        g[f]=s
      else :
        g[f]=random.randint(0,n)
    for j in xrange(i):
      if j in g:
        B[i][j]=1
  for i in xrange(n):
    for j in xrange(i-1,-1):
                B[i][j]=B[j][i]
  return B
```

Let's now check that this new matrix has the same characteristics as Caltech. The following program plots the same graph as before, but associated to our new matrix: **(Fig.3)**

```
C=matricerandom2(F,v)
t =np.zeros(n)
y=statistique(C)
v=np.zeros(n)
for i in xrange(n):
   v[i]=y[i]
   t[i]=i
plot(t,v)
show()
```

As we can see on **Fig. 3**, the graph representing for each degree the corresponding number of nodes having this degree associated with the new matrix has the same shape as the one associated with Caltech **(Fig. 2).**

We have now to generate a random sybil attack. As it was explained earlier, we decided to use exponential probability distributions to determine the number of sybil nodes and attack edges. According to our hypotheses, a sybil is a user who can make as many false accounts as he wants, but

will however create, with a high probability, a very limited quantity of these accounts. In particular, this quantity will be very small in comparison of the size of the real graph. We thus chose to use two exponential distribution, one with a greater parameter than the other. This is a choice which represents our view of Sybil attacks. To create the program *k*, we use the fact that given random variable U which follows a uniform distribution, and given a cumulative distribution function F, F will be the cumulative distribution function of the random variable $F^{-1}(U)$. The following program generates an exponential probability distribution:

```
def k(v):                                   #v is the parameter of the exponential distribution
  d=random.random()
  s=(math.log(1-d)/v)
  return -s
```

To complete our computed social graph, we have now to link the sybil network with the real one. The following program randomly determines which nodes will link these two networks:

```
def generation(n,f):        #n is the size of the matrix, and f the number of attack edges
   c=[[(i,j) for i  in range(f)]for j  in range(n)]
   for i in xrange(f):
     for j in xrange(n):
        k = random.randint(0, n - j-1)
        c[j][i],c[k+j][i] = c[k+j][i],c[j][i]
   return  c
```

We can now link the two social networks using the attack edges we just defined. In this program we chose for the number of attack edges a parameter equal to 0.1 and equal to for the number of false accounts. We made sure that the number of fake accounts and of attack edges were at least equal to 1. Thus, we first used the program *generation* to randomly create n (n being the size of our matrix A) new vectors, each of them having a size k. (k being the number of false accounts). Finally, we filled the matrix by the k new vectors, which have a size k+n. The corresponding program is:

```
def ajoute(A):
  n=len(A)
  f=k(0.1)
  s=k(1)
  if f>=n :                          #this happens with a really low probability
    f=n-1
  if s>=n :
    s=n-1
  f=int(f)+1                         #+1 because we need at least one sybil node
  s=int(s)
  G=generation(n,f)
  V=[[0 for i in range(f)]for j in range(n)]

  for i in xrange(f):
    for j in xrange(s+1):          #+1 because we need at least one connection
```

```
            (a,b)=G[j][i]
            V[b][a]=1
        for i in xrange(n):
            A[i]=A[i] +V[i]
        for j in xrange(f):
            b=[0 for s in range(n+f)]
            for g in xrange(n):
                if A[g][n+j]==1:
                    b[g]=1
            for g in xrange(n+1,n+f):
                b[g]=1
            A=A+ [b]
        return A
```

Finally, we can apply to our computed adjacency matrix the program that creates the random routes for each node and study the intersections between these routes. The following program returns the list of the different numbers of intersections between the routes of two nodes : a verifier node *v*, and a suspected one *s*. The mechanism is iterated *m* times:

```
def listpour (A,v,s,m):
    B=[0 for i in range (m)]
    Q=friends(A)
    q=tablederoutage(A,Q)
    n=len(A)
    w=int(sqrt(n)*math.log(n))
    g=A
    h=A
    d=Q
    c=Q
    j=q
    I=q
    for s in xrange(m):
        B[s]=comp(g,d,j,v,s,w)
        g=h
        d=c
        j=I
    return B
```

The next estimation function uses listpour and returns the empirical expectation:

```
def estimation(A,v,s,m):
    V=listpour(A,v,s,m)
    s=somme(V,m)
    return s/m
```

Given a matrix D (we will use the matrix we will create) , a vector *v* (we will use somme(f,n)) , a node *s* (which will be the verifier node), n the length of A (our real adjacency matrix) and finally x a number we will use as a criterion. This function will return the matrix of size (2, 2) J, $J_{(0,0)}$ being equal to the number of honest nodes detected as honest nodes. $J_{(1,0)}$ is equal to the number of sybil nodes detected as honest nodes, $J_{(0,1)}$ is equal to the number of honest nodes detected as sybil nodes and finally $J_{(1,1)}$ is equal to the number of sybil nodes detected as a sybil nodes. *k-n* will be the number of Sybil nodes in the matrix D. Since this function could be really long to execute, we will only use the *k-n* first nodes of D as honest nodes. However this will not be a problem, sine the matrix D is determined randomly. The goal of this function is to determine the criterion x which will minimize $J_{(0,1)}$ and $J_{(1,0)}$. If two criteria seem acceptable we will choose the one with the lower $J_{(0,1)}$: indeed, since this criterion will be used to determine if a node is sybil in comparison to the node *s*, the user will prefer failing to detect some sybils rather than banishing an honest node.

```
def houbre(D,v,s,x,n):
    f=friends(D)
    B=tablederoutage(D,f)

    k=len(D)
    print k

    w=int(math.sqrt(k)*math.log(k)
    J=[[0 for q in rang(2)] for d in range(2)]
    xou=[comp(D,f,B,I,s,w) for I in range(k-n)
    xou2=[comp(D,f,B,I,s,w) for I in range(n,k)]
    print xou, xou2

    for I in xrange((k-n)):
            if xou[i]==x:
          J[0][0]=J[0][0]+1
       if xou[i]>x:
          J[0][0]=J[0][0]+1
       if xou[i]<x:
          J[0][1]=J[0][1]+1
    for I in xrange(n,k):
            if xou2[i]==x:
          J[1][1]=J[1][1]+1
       if xou2[i]>x:
          J[1][0]=J[1][0]+1
       if xou[i]<x:
          J[1][1]=J[0][1]+1
    return J
```

Results

Given a matrix A (the matrix representing the Caltech's matrix), a vector *v* (statistique(f,n)), and the nodes v and s, the function will return True if the node is sybil and False if the node is honest. After 6 executions of houbre, we decided to fix the value of this criterion to 50.
Hereunder is an execution example:
The list of the number of intersections with honest nodes:
[163, 156, 156, 140, 149, 144, 130, 161, 140, 140, 147, 160, 136, 165, 151, 147, 158];
The list of the number of intersection with the node of sybil nodes:
[16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0].

```
def resul(A,v,s,ve):
    n=len(A)
    verite=False
    Fr=friends(A)
    RT=tablederoutage(A,Fr)
    w=int(math.sqrt(n)*math.log(n))
    r=estimation(A,ve,s,m)
    if r<50:
        verite=True
    return verite
```

Difficulties and problems met

1. Heavy calculations

Due to a significant number of loops in our code and to our unawareness, the first executions of the algorithm took a long time (more than several hours). Fortunately, we quickly managed to figure out what the issue was: several command lines used in every function were included in these functions. For instance, the instruction B=friends(A) was included in every following function. Considering the huge size of the matrix 'Caltech' we applied our code on (769, 769), recalculating friends(A) in every function was really slowing down Python's executing time. Thus, we simply changed every function and made them depend not only on the matrix A, but also on another matrix B, defined as B=friends (A) at the beginning of our code.

2. Modeling the function f

Finding a function that was following as much as possible the variations of the graph on **Fig. 1** was much more difficult.

Due to the global shape of the graph, we started using a function of the form:

$$g: x \to a * f(x) * x^{-j}$$

To determine one unknown parameter, we studied the asymptotes of this function, which had to be the same as f's: $f(0) = 35$, and $\lim f(x)_{x \to +\infty} = 0$.

Finally, to determine the last unknown parameter, we used the following program and imposed the average node degree to be equal to Caltech's average node degree:

```
from pylab import *
import numpy
t = arange(0, 769, 1)
s=np.zeros(769)
for i in xrange (1,769):
s[i]=0.25*1000*(exp(-0.75*log(i*(1+exp(-log(0.75/35)/i)))))
plot(t, s)
show()

def fou(w):
   s=0
  for i in xrange(1,769):
     s+=i*1000*(exp(-100*log(i*(1+exp(-log(100/35)/i)))))
     print(100*float(i)/float(769))
  return s/769
print fou(10000)
#(exp(-1.3567*log(i*(-1+(1/(exp(-i*(35/1.367))))))))
#s[i]=1000*(exp(-1.3567*log(i*(1+exp(-log(1.3567/35)/i))))"""
```

3. How to generate a random matrix which wil represents A

In order to find a criterion that would allow us to decide if a node is sybil or not, we needed in a first time to create a random plausible adjacency matrix D. We thought about creating a function which would create a matrix D which coefficients would all be randomly determined thanks to a Barnoulli probability distribution, which had an expectation estimated by the mean node degree of A. We created the corresponding functions but then realized that such a matrix would not have the same profile as our first matrix. In fact, even if the estimate expectation was asymptotically equal to the average node degree, the number of nodes that had the same degree could diverge. This could have a big impact in our case because we use random routes which are linked with degrees.

Conclusion

We created an algorithm that is able to detect if a node is sybil or honest. The response this algorithm returns doesn't give a definitive answer because it is based in random procedure. However, it can still be used to help to detect if a node is sybil or not. One of our main hypotheses concerned the shape of the social graph: our program works only in graphs that have approximately the same profile than the Caltech graph. This algorithm can be extended to other social graphs using non-linear regression.