A review of the paper: *Inherent directionality explains the lack of feedback loops in empirical networks*

Rubén Hurtado, Bartolomé Ortiz, Cristina Seva

Master en Física y Matemáticas
Universidad de Granada
10/06/2018

Abstract

This work is a brief revision and study about a paper called: **Inherent directionality explains the lack of feedback loops in empirical networks** written by *Virginia Domínguez García, Simone Pigolotti and Miguel A. Muñoz* . Our aim is to present its mains results, some technical highlights related to the mathematical and physical advances, and reproduce a minor result using our own methods. It will be analized its implications and future research too. Althought this work is merely a revision it could be useful as a source of code to reproduce some of the results.

Keywords: `complex networks`, `mathematics`, `complexity`, `graph theory`

1. Introducción

El objetivo de este trabajo es presentar el artículo seleccionado [1], de manera que expondremos sus principales descubrimientos, así como algunas notas sobre los procedimientos que en él se presentan.

Tras esto, vamos a realizar un pequeño intento de simulación para intentar reproducir algunos de los resultados que aparecen en el articulo.

Finalmente, para cerrar nuestro trabajo, hablaremos un poco sobre las conclusiones obtenidas de este y sobre posibles vías para avanzar.

2. Review del artículo

El artículo muestra una serie de resultados que relacionan la direccionalidad inherente al sistema complejo (representado mediante grafos) con la ausencia de ciclos que se retroalimenten dentro de este grafo. Para continuar vamos a dar una pequeñas nociones sobre los elementos fundamentales del artículo:

- **Direccionalidad inherente**: hablamos de direccionalidad inherente a un sistema complejo cuando todos los nodos se pueden ordenar en un eje unidimensional, de tal manera que los enlaces apuntan desde valores bajos a altos de sus coordenadas en dicho eje.

  Como bien se apunta en [1], la existencia de esta direccionalidad está relacionada con la existencia de una estructura jerárquica dentro de nuestra red. Así, aunque la aplicación de los resultados es muy amplia, podemos relacionarlo directamente con la temática de redes tróficas que vimos en clase.

- **Ciclo retroalimentado o feedback loop**: hablamos de ciclo retroalimentado de orden k dentro de un grafo direccional, cuando nos referimos una sucesión cerrada de k nodos (esto es, con el mismo nodo en primera y última posición), cuyo orden de aparición viene determinado por el camino que indica el sentido de las aristas que los conectan.

Debido al gran impacto de la existencia de feedback loops en la estabilidad dinámica del sistema, el artículo se centra en encontrar una herramienta predictiva para conocer, basándose en un solo parámetro γ, la fracción de ciclos de orden k que también son ciclos retroalimentados. Se observa experimentalmente que esta fracción siempre es menor en redes con direccionalidad inherente que en redes aleatorizadas.

2.1. Métodos

Exponemos ahora la forma de obtener la herramienta predictiva descrita en el artículo. Consideramos una red de N nodos, y L aristas e imaginamos que conocemos la fracción de ciclos retroalimentados de orden k: $F(k)$. Con ello , si asumimos la existencia de direccionalidad, se construye el modelo aleatorizando la direccionalidad de la siguiente manera:

- Se escoge un nodo. Con probabilidad $0 < \gamma < 1$ su arista apunta en la dirección inherente, i.e.; hacia los nodos de mayor jerarquía, y con probabilidad $1-\gamma$ apunta en sentido contrario.

Al parámetro γ se le llama **parámetro de direccionalidad.**

Si ahora nos centramos en un ciclo de orden k e imponemos una notación jerárquica para los nodos de este ciclo obtendríamos un total de $k!$ posibles ciclos.

En general, la probabilidad de tener un ciclo retroalimentado dado un ciclo cualquiera dependerá del numero de ascensos $A(l,k)$. Este numero cuenta cuántas permutaciones de la secuencia básica de longitud k con $nodo_i < nodo_{i+1}$ se tienen para l valores distintos de i. Para una secuencia no periódica, es decir, sin establecer ninguna relación entre $nodo_k$ y $nodo_1$ la solución a este problema está dada por los llamados números eulerianos.

Sin embargo, como los ciclos que buscamos, al ser retroalimentados, están cerrados, se generaliza el concepto de números de Euler para el caso periódico o cíclico. Es decir, necesitamos contar el número de ascensos en un ciclo cerrado genérico, lo que en el articulo se bautiza como los **numeros eulerianos cíclicos**, denotados por $A(l,k)$. Con este interesante concepto en mente, los autores ya pueden desarrollar la expresión de la función buscada, necesitando ahora añadirle el parametro de direccionalidad:

$$F(k,\gamma) = \sum_{l=0}^{k} \frac{A(l,k)}{k!}[\gamma^l(1-\gamma)^{k-l} + \gamma^{k-l}(1-\gamma)^l]$$

Usando los procesos del material suplementario, la expresión puede aproximarse por :

$$F(k,\gamma) \approx 2exp\{\frac{k}{2}log[\gamma(1-\gamma)] + \frac{k}{24}log^2(\frac{\gamma}{1-\gamma})\}$$

Esta expresión es la que hemos usado para todas las partes replicadas de nuestra revisión. Para concluir, presentamos en la gráfica 1, que también se puede ver en [1], la relación entre el parámetro de forma y la fracción $F(k,\gamma)$

2.2. Algunos resultados a destacar

Entrando en la parte de resultados, queremos destacar algunos de los que nos han parecido más relevantes.

- La fracción de los ciclos de retroalimentación de cualquier longitud k es mucho más pequeña en redes biológicas y ecológicas de lo que cabría esperar para cualquier otro grafo similar pero aleatorizado.

- El número total de ciclos de retroalimentación también se reduce con respecto a

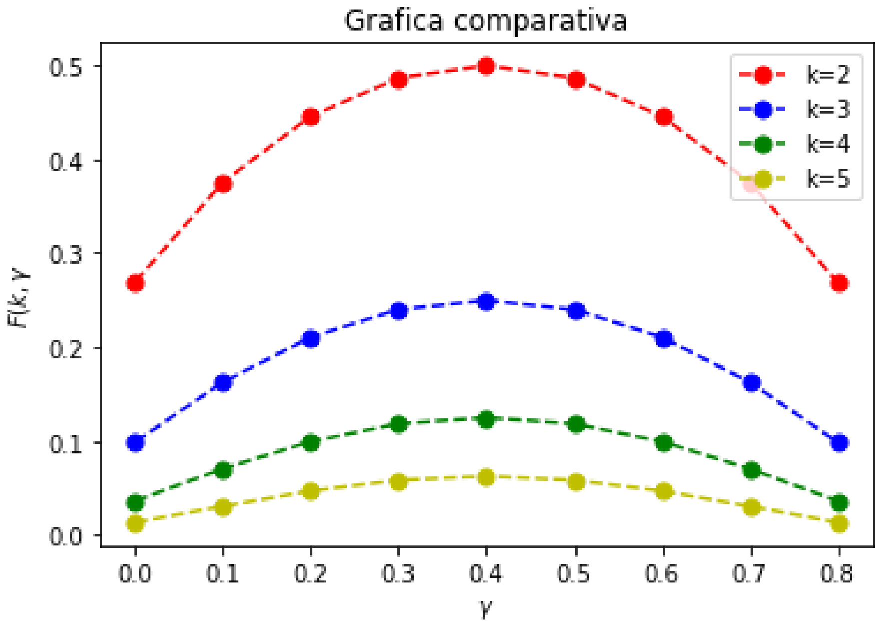

Figura 1: $F(k)$ frente a γ, se puede observar el pico en $\gamma = 1/2$

las aleatorizaciones de red. Sin embargo, y esto es por lo que nos llamó la atención realizarlo en Twitter, estas tendencias no son tan evidentes para las redes socio-tecnológicas; mientras que todas las redes consideradas tienen una fracción más pequeña de bucles de retroalimentación que sus aleatorizaciones de direccionalidad, la red social de Twitter exhibe una mayor $F(k)$ que las aleatorizaciones.

- Por supuesto, el modelo desarrollado reproduce bastante bien las muestras experimentales obtenidas, que muestran un decaimiento exponencial de $F(k)$ conforme aumenta k.

Para observar gráficamente estos resultados podemos irnos a la gráfica 2, donde hemos destacado la sección de redes tecnológicas. Antes, sin embargo debemos hacer una reseña sobre la información a encontrar:

En la grafica se compara la fracción medida de los ciclos retroalimentados $F(k)$ con dos aleatorizaciones de la misma red. La primera, que llaman aleatorización de la direccionalidad (DR) - conserva las aristas existentes, pero aleatoriza completamente sus direcciones. La segunda, aleatorización de configuración (CR), aleatoriza tanto aristas

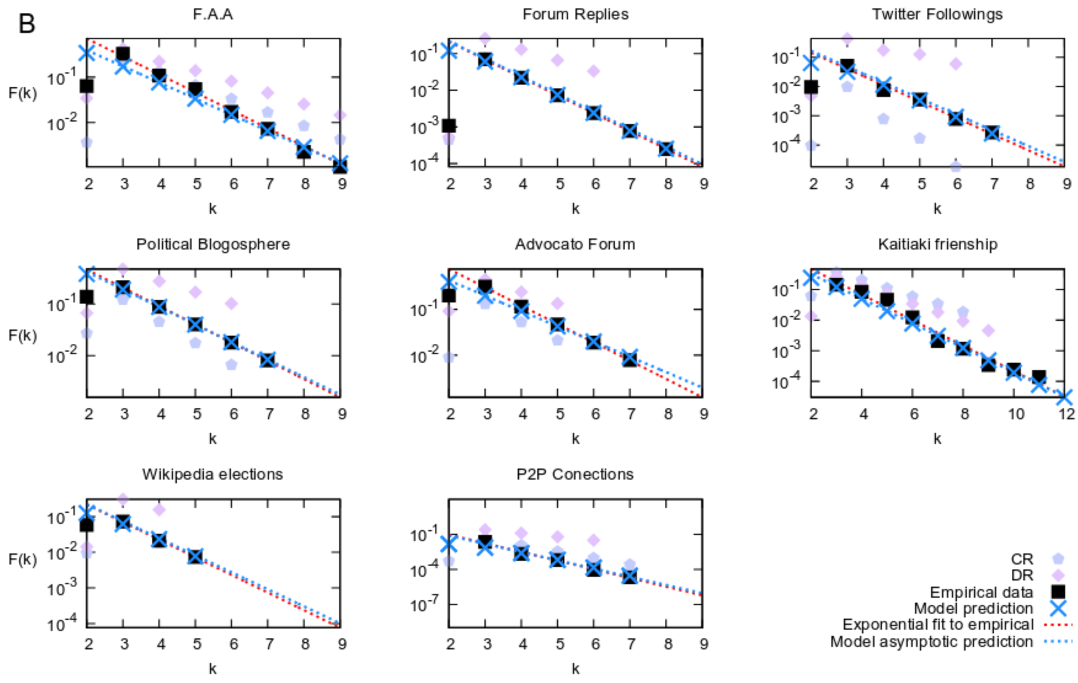

Figura 2: Gráfico extraído del artículo donde resaltamos la parte de redes tecnológicas

como direcciones, pero preservando la conectividad de entrada y salida de cada nodo.
Finalmente queremos resaltar el apartado sobre cómo información sobre nuevos nodos puede alterar lo que sabemos hasta ahora por las predicciones.

Para probar la robustez de su predicción, los investigadores simularon el efecto de desconocimiento sobre las redes que ya tenían, eliminando una fracción de los enlaces al azar, y repitieron el análisis anterior.

Aunque esta operación afecta claramente a la cantidad de enlaces, las conclusiones del modelo apenas varían, lo cual nos parece muy interesante. Incluso, cuando la cifra de nodos que se elimina está entre $20-$ y $50-$ podemos observar esta tendencia en la gráfica obtenida del material suplementario 3

3. Intento de reproducción

Desde el principio nos atrajo la cantidad de aplicaciones que la investigación podía tener, por eso decidimos coger este articulo para el trabajo. Durante su lectura acordamos que

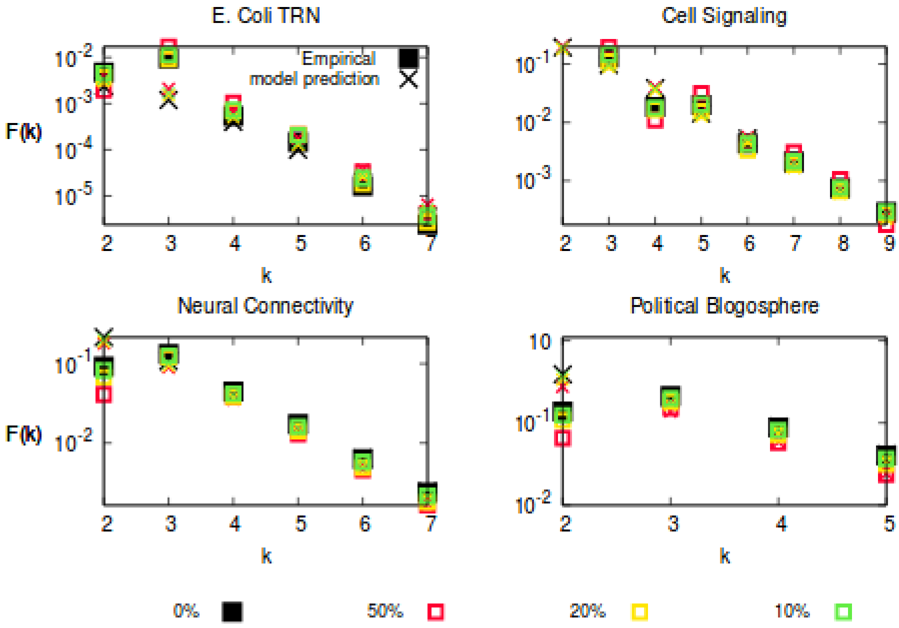

Figura 3: Gráfico extraído del artículo donde resaltamos el impacto de los nodos o conexiones no observadas

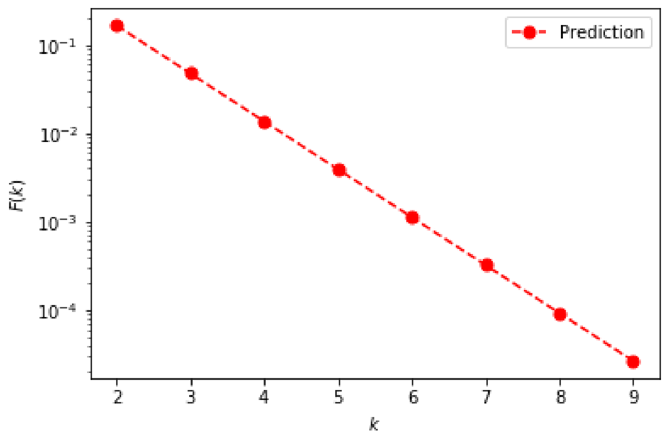

Figura 4: Gráfico extraído del artículo donde resaltamos la parte de redes tecnológicas

la mejor manera de intentar comprenderlo era la reproducción parcial de los resultados y, con tal fin, pensamos que, debíamos escoger alguna red a la que tuvieramos fácil acceso. Pasó por nuestra cabeza la posibilidad de fijarnos en las redes tecnológicas que se exponían al final, pues contenían detalles interesantes como hemos descrito antes.

En particular, nos centramos en la reproducción de redes en Twitter. En este caso estaríamos ante una predicción como la que se observa en la gráfica 4.

Por tanto, escogimos esta red social y, basándonos en la base de datos expuesta en el material complementario de [1], nos pusimos a probar.

Lamentablemente, el volumen de datos echaba por tierra cualquier intento por nuestra parte, debido a la poca potencia de nuestros ordenadores.

Estando en esta situación, decidimos que la mejor manera de sobreponernos era obtener nuestra propia fuente de datos, de una de nuestras redes (más pequeña que la usada en el estudio) de Twitter.

Para ello usamos el paquete *Tweepy* el cual nos ofrece la posibilidad de conectarnos a la API de twitter para la obtención de nuestro grafo de seguidores. Hemos de destacar que las restricciones de petición de usuarios a la aplicación hicieron que nos retrasásemos, pues nuestro dataset tardó días en estar completo.

Una vez completo el dataset, analizamos la información mediante el paquete *NetworkX*:

- En primer lugar analizamos los ciclos de en nuestra red. Para ello utilizamos una version no-recursiva de iterador/generador del algoritmo de Johnson, implementada por nosotros basada en la librería de NetworkX, definida como :

```
def simple_cycles_undirected(G, maxlength=float('inf'))
```

Aquí nos volvimos a topar con la velocidad de ejecución. Aun siendo el algoritmo elegido bastante bueno en cuanto a complejidad y tiemp ode ejecucion, nuestros ordenadores no eran lo suficientemente rápidos lo que nos hizo perder mucho tiempo en obtener resultados.

- Luego utilizamos otra función implementada en la librería de NetworkX para el conteo de ciclos de retroalimentacion en grafos dirigido, definida como :

```
feedback_cycle_list_DG1 = (list(simple_cycles(DG1)))
```

con todo ello, disponíamos de información suficiente para hacer una comparativa: Como extra, puede consultar nuestro workflow en el apéndice.

4. Conclusiones finales

aaa

Referencias

[1] V. Domínguez-García, S. Pigolotti, M. A. Munoz, Inherent directionality explains the lack of feedback loops in empirical networks, Scientific reports 4 (2014) 7497.

Appendices

A. Workflow de conversion y resultados

```
# -*- coding: utf-8 -*-
"""
Created on Mon Jun 11 13:34:41 2018

@author: booort, ruhugu
```

```
"""

from networkx import Graph, DiGraph, simple_cycles, find_cycle, cycle_basis
import networkx as nx
from collections import defaultdict
import csv

# Define functions
# ==========================================
""" This functions has been modified from the function simple_cycles
    in networkx package, which is distributed under a BSD license.
        networkx github page: https://github.com/networkx/

"""
def simple_cycles_undirected(G, maxlength=float('inf')):
    # TODO: Update docs!
    """Find simple cycles (elementary circuits) of a undirected graph.

    A `simple cycle`, or `elementary circuit`, is a closed path where
    no node appears twice. Two elementary circuits are distinct if they
    are not cyclic permutations of each other.

    This is a nonrecursive, iterator/generator version of Johnson's
    algorithm [1]_. There may be better algorithms for some cases [2]_ [3]
    _.

    Parameters
    ----------
    G : NetworkX Graph
       A undirected graph

    maxlength : int
       Maximum length of the cycle.

    Returns
    -------
    cycle_generator: generator
       A generator that produces elementary cycles of the graph.
       Each cycle is represented by a list of nodes along the cycle.
```

```
Examples
--------
>>> edges = [(0, 0), (0, 1), (0, 2), (1, 2), (2, 0), (2, 1), (2, 2)]
>>> G = nx.DiGraph(edges)
>>> len(list(nx.simple_cycles(G)))
5

To filter the cycles so that they don't include certain nodes or edges,
copy your graph and eliminate those nodes or edges before calling

>>> copyG = G.copy()
>>> copyG.remove_nodes_from([1])
>>> copyG.remove_edges_from([(0, 1)])
>>> len(list(nx.simple_cycles(copyG)))
3


Notes
-----
The implementation follows pp. 79-80 in [1]_.

The time complexity is $O((n+e)(c+1))$ for $n$ nodes, $e$ edges and $c$
elementary circuits.

References
----------
.. [1] Finding all the elementary circuits of a directed graph.
   D. B. Johnson, SIAM Journal on Computing 4, no. 1, 77-84, 1975.
   https://doi.org/10.1137/0204007
.. [2] Enumerating the cycles of a digraph: a new preprocessing
strategy.
   G. Loizou and P. Thanish, Information Sciences, v. 27, 163-182,
1982.
.. [3] A search strategy for the elementary cycles of a directed graph.
   J.L. Szwarcfiter and P.E. Lauer, BIT NUMERICAL MATHEMATICS,
   v. 16, no. 2, 192-204, 1976.

See Also
```

```
--------
cycle_basis
"""
def _unblock(thisnode, blocked, B):
    stack = set([thisnode])
    while stack:
        node = stack.pop()
        if node in blocked:
            blocked.remove(node)
            stack.update(B[node])
            B[node].clear()

# Johnson's algorithm requires some ordering of the nodes.
# We assign the arbitrary ordering given by the strongly connected
comps
# There is no need to track the ordering as each node removed as
processed.
# Also we save the actual graph so we can mutate it. We only take the
# edges because we do not want to copy edge and node attributes here.
# Create a symemtric directed graph from the given undirected one
subG = type(G.to_directed())(G.edges())
sccs = list(nx.strongly_connected_components(subG))
while sccs:
    scc = sccs.pop()
    # order of scc determines ordering of nodes
    startnode = scc.pop()
    # Processing node runs "circuit" routine from recursive version
    path = [startnode]
    blocked = set()  # vertex: blocked from search?
    closed = set()   # nodes involved in a cycle
    blocked.add(startnode)
    B = defaultdict(set)  # graph portions that yield no elementary
circuit
    stack = [(startnode, list(subG[startnode]))]  # subG gives comp
nbrs
    while stack:
        thisnode, nbrs = stack[-1]
        if nbrs and (len(path) <= maxlength):
            nextnode = nbrs.pop()
```

```
                if nextnode == startnode:
                    yield path[:]
                    closed.update(path)
#                        print "Found a cycle", path, closed
                elif nextnode not in blocked:
                    path.append(nextnode)
                    stack.append((nextnode, list(subG[nextnode])))
                    closed.discard(nextnode)
                    blocked.add(nextnode)
                    continue
            # done with nextnode... look for more neighbors
            if not nbrs or (len(path) > maxlength):  # no more nbrs
                if thisnode in closed:
                    _unblock(thisnode, blocked, B)
                else:
                    for nbr in subG[thisnode]:
                        if thisnode not in B[nbr]:
                            B[nbr].add(thisnode)
                stack.pop()
#                 assert path[-1] == thisnode
                path.pop()
        # done processing this node
        subG.remove_node(startnode)
        H = subG.subgraph(scc)  # make smaller to avoid work in SCC routine
        sccs.extend(list(nx.strongly_connected_components(H)))


def savecycles(filename, cycles_iter):
    with open(filename, "w") as file_out:
        for cycle in cycles_iter:
            for node in cycle:
                file_out.write("{0},".format(node))

            file_out.write("\n")

def loadcycles(filename):
    cycles = list()
    with open(filename, "r") as file_in:
        for line in file_in:
```

```
            cycles.append(line.strip(",\n").split(","))

    return cycles


# Read CSV file
def csv2dict(filename, delimiter=";"):
    """Reads a 2-column csv file into a dict.

    The value in the first column is used as dictionary key. The value
    of the dict entry is a list with the value of the second column.
    If several rows have the value in the first column, their
    second column values are stored in the same list.

    Parameters
    ----------
        filename : str
            Name of the file to be read.

        delimiter : str
            Delimiter used in the csv file.

    """
    with open(filename) as csvfile:
        reader = csv.reader(csvfile, delimiter=delimiter)

        dictionary = {}

        for entry in reader:
            key = entry[0]
            value = entry[1]

            # If the key is already in the dict, append the value
            # to the list
            if key in dictionary:
                dictionary[key].append(value)
            # In other case, create a new dict entry
            else:
                dictionary[key] = [value, ]
```

```
    return dictionary

"""
OJITO definitiva contiene un sample mucho mas grande de una red de twitter
mi ordenador no tira no con ella, por eso esta comentada

"""

# Load data
# ============================================
# Get the information from both files
dictionary1 = csv2dict("sevaseviene2network.csv", delimiter=";")
#dictionary2 = csv2dict("Definitiva.csv", delimiter=";")

# These function transform the dictionary to a directed graph networkx-
    style
DG1 = DiGraph(dictionary1)
#DG2 = DiGraph(dictionary2)

# These function transform the dictionary to an undirected graph networkx-
    style
undirected1 = DG1.to_undirected()
#undirected2 = DG2.to_undirected()


# Count cycles
# ================================
# These functions counts all the feedback cycles in our graph
feedback_cycle_list_DG1 = (list(simple_cycles(DG1)))
#cycle_list_DG2 = (list(simple_cycles(DG2)))


# Finally this part should gets all loops via networkx function
cycle_list_DG1 = simple_cycles_undirected(undirected1, maxlength=10)


# Save cycles to file
# ============================================
```

```
# Save directed cycles
savecycles("sevaseviene_directedcycles.dat", feedback_cycle_list_DG1)
# Save undirected cycles
savecycles("sevaseviene_undirectedcycles.dat", cycle_list_DG1)



# Play with the data
# ======================================
print(len(feedback_cycle_list_DG1))
#print(len(cycle_list_DG1))

# feedback_number[i]=number of cycle with i+1 longitude
feedback_number = [0,0,0,0,0,0,0,0,0,0]

#counting feedback numbers
for i in range(0,len(feedback_cycle_list_DG1)):
    if len(list(feedback_cycle_list_DG1[i])) == 1:
        feedback_number[0]+=1
    elif len(list(feedback_cycle_list_DG1[i]))==2:
        feedback_number[1]+=1
    elif len(list(feedback_cycle_list_DG1[i]))==3:
        feedback_number[2]+=1
    elif len(list(feedback_cycle_list_DG1[i]))==4:
        feedback_number[3]+=1
    elif len(list(feedback_cycle_list_DG1[i]))==5:
        feedback_number[4]+=1
    elif len(list(feedback_cycle_list_DG1[i]))==6:
        feedback_number[5]+=1
    elif len(list(feedback_cycle_list_DG1[i]))==7:
        feedback_number[6]+=1
print(feedback_number)
```

B. Obtención de datos en twitter

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri May 25 19:25:59 2018

Modifications by thebooort
```

```
@author: Mik
"""
import csv
import time

import tweepy

# Copy the api key, the api secret, the access token and the access token
    secret from the relevant page on your Twitter app

api_key = '—'
api_secret = '—'
access_token = '—'
access_token_secret = '—'

# You don't need to make any changes below here # This bit authorises you
    to ask for information from Twitter
auth = tweepy.OAuthHandler(api_key, api_secret)
auth.set_access_token(access_token, access_token_secret)
# The api object gives you access to all of the http calls that Twitter
    accepts
api = tweepy.API(auth)

#User we want to use as initial node
user='sevaseviene'


#This creates a csv file and defines that each new entry will be in a new
    line
csvfile=open(user+'_twitter_network.csv', 'w')
spamwriter = csv.writer(csvfile, delimiter=' ',quotechar='|', quoting=csv.
    QUOTE_MINIMAL)

#This is the function that takes a node (user) and looks for all its
    followers #and print them into a CSV file... and look for the followers
     of each follower...
def fib(n,user,spamwriter):
    if n>0:
```

```
        #There is a limit to the traffic you can have with the API, so you
    need to wait
        #a few seconds per call or after a few calls it will restrict your
    traffic
        #for 15 minutes. This parameter can be tweeked
        time.sleep(40)

        #This is for private users that we wont be able to see their
    followers
        try:
            users=tweepy.Cursor(api.followers, screen_name = user,
    wait_on_rate_limit = True).items()
            for follower in users:
                print(follower.screen_name)
                spamwriter.writerow([user+';'+follower.screen_name])
                fib(n-1,follower.screen_name,spamwriter)
                #n defines the level of autorecurrence

        except tweepy.TweepError:
                print("Failed to run the command on that user, Skipping..."
    )



n=2
fib(n,user,spamwriter)
```