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. Its implications and future research will also be analysed. Although 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 tratar de 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 a muchos sistemas complejos empíricos (representados mediante grafos) con la ausencia de ciclos que se retroalimenten (*feedback loops*) dentro de estos. Para empezar vamos a dar una pequeñas nociones sobre los elementos fundamentales del artículo:

- **Direccionalidad inherente**: hablamos de direccionalidad inherente a un sistema complejo cuando, en el grafo que lo representa, todos los nodos se pueden ordenar en un eje unidimensional, de tal manera que los enlaces tienden a apuntar 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. Por el contrario, llamamos **ciclo estructural** o simplemente ciclo de orden k a un sucesión cerrada de nodos conectados en la que no se considera el sentido de las aristas.

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 la fracción de ciclos para cada orden k que también son ciclos retroalimentados, bajo el supuesto de que existe direccionalidad en la red caracterizada por un solo parámetro γ.
El artículo se divide en tres puntos principales. En primer lugar, se diseña un modelo probabilístico para, bajo ciertas simplificaciones, obtener la probabilidad de que un ciclo de orden k sea reatroalimentado en función de cierto parámetro γ que indica la direccionalidad de la red.
A continuación se analizan redes empíricas para obtener la fracción de ciclos retroalimentados frente a estructurales en estas. Se observa que esta fracción siempre es menor en redes con direccionalidad inherente que en redes aleatorizadas. La excepción ocurre en redes socio-tecnológicas como la de los seguidores de Twitter, que analizaremos en detalle más adelante.
Finalmente, se ajustan los datos experimentales usando dos métodos diferentes. Por una lado se obtiene el valor de γ del modelo que mejor ajusta a los datos mediante mínimos

cuadrados. Luego se ajustan los datos a una curva exponencial y se compara el resultado con la predicción asintótica del modelo, también exponencial, usando la γ anterior.

El buen ajuste que aparece en el artículo entre la predicción y los datos indica claramente la relación de causalidad entre la direccionalidad de la red y la ausencia de ciclos retroalimentados.

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.

Asumimos que existe direccionalidad en esta red y la construimos aleatorizando el sentido de las aristas de la siguiente manera: para cada arista, con probabilidad $0 < \gamma < 1$ hacemos apuntar ésta en la dirección inherente, i.e.; hacia los nodos de mayor jerarquía, y con probabilidad $1 - \gamma$ hacia el sentido contrario. Al parámetro γ le llamaremos **parámetro de direccionalidad.**

Dada esta red aleatoria, nos preguntamos, dado un valor de γ, cuál es la probabilidad de que un ciclo de orden k se además retroalimentado. Denotaremos esta probabilidad por $F(k, \gamma)$.

Para ello, empezamos etiquetando con una notación jerárquica los nodos de este ciclo. Existirán un total de $k!$ posibles ciclos estructurales, tantos como permutaciones de las etiquetas. Para obtener la probabilidad buscada hemos de encontrar, de todos los ciclos anteriores, cuántos son retroalimentados. En general, la probabilidad de obtener un ciclo retroalimentado a partir de un ciclo cualquiera dependerá del número de ascensos $A(l, k)$. Este número cuenta cuántas permutaciones de la secuencia básica de longitud k con $\text{nodo}_i < \text{nodo}_{i+1}$ se tienen para l valores distintos de i. Para una secuencia no periódica, es decir, sin establecer ninguna relación entre el primer y último nodo, 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 **números eulerianos cíclicos**, denotados por $A(l, k)$. Con este interesante concepto en mente, los autores ya pueden desarrollar la expresión de la

función buscada:

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

Aquí se multiplica el número de permutaciones con l ascensos, $A(l,k)$, por su probabilidad de ocurrencia, que aparece en el corchete. Ésta se obtiene trivialmente a partir de γ que corresponde a la probabilidad de ascenso. Finalmente, se divide el resultado por el número total de permutaciones $k!$ para obtener la fracción de ciclos retroalimentados.

Usando los procesos del material suplementario, esta expresión puede aproximarse, en el caso de k grande y γ no muy pequeño, por:

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

Este proceso nos permite obtener una expresión teórica con la que comparar los datos experimentales, con los que se ajusta además muy bien. Sin embargo, no hay que perder de vista que en la obtención de la ecuación 1 se ha hecho una simplificación importante: cada ciclo se ha considerado de forma completamente independiente; se han obviado posibles relaciones entre ciclos o diferencias entre éstos a diferentes alturas de la estructura jerárquica. En el análisis del artículo esto no supone ningún problema, lo que justifica como válidas las aproximaciones anteriores. Sin embargo, en nuestro intento de reproducción sí jugarán un papel muy importante.

Para concluir, presentamos en la gráfica 1, que también se puede ver en [1], la relación entre el parámetro de forma γ 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 las aleatorizaciones de red. Sin embargo, y esto es por lo que nos llamó la atención

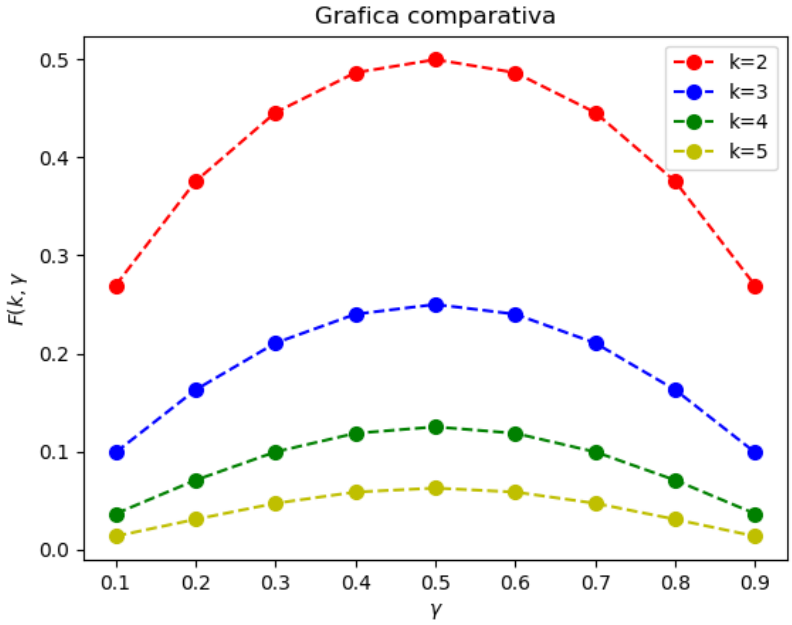

Figura 1: $F(k)$ frente a γ, se puede observar el pico en $\gamma = 1/2$

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. Esto puede observarse desde dos perspectivas. Por un lado, las X en el gráfico representan la predicción del modelo cuando γ ha sido ajusta mediante mínimos cuadrados a los datos. Aquí podemos ver que en la amplia mayoría de los casos hay un muy buen acuerdo entre los valores experimentales y el modelo. Por otro lado, se compara la predicción asintótica del modelo de la ecuación 2 con una regresión exponencial a los datos. Aquí de nuevo observamos una muy buena coincidencia.

Para observar gráficamente estos resultados podemos irnos a la gráfica 2, donde hemos

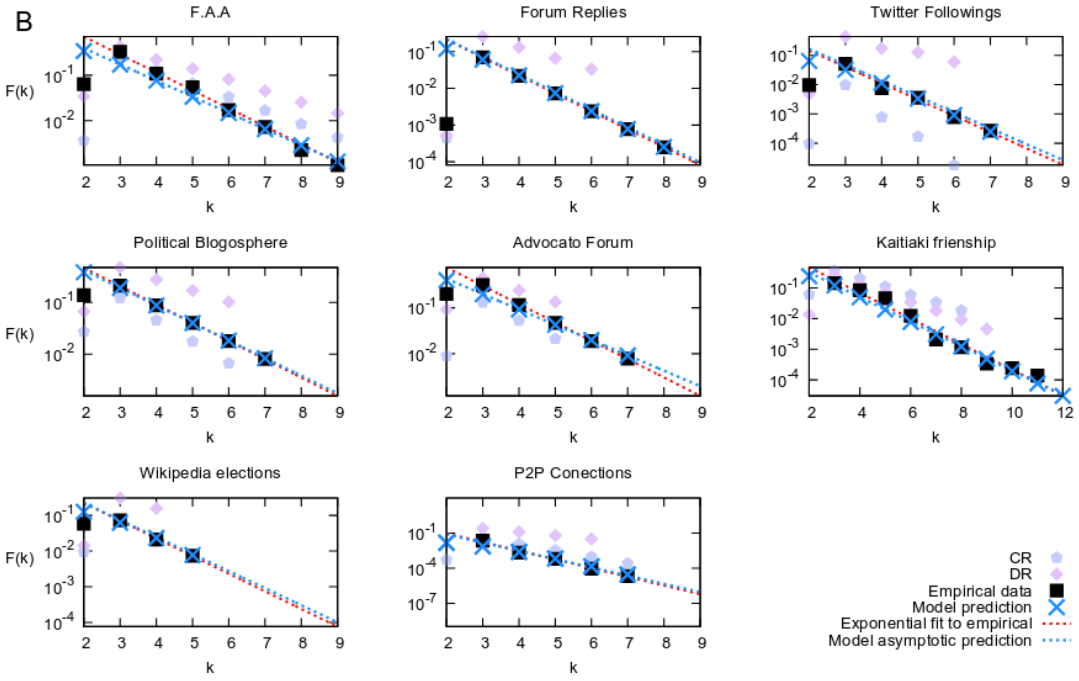

Figura 2: Gráfico extraído del artículo donde resaltamos la parte de redes tecnológicas

destacado la sección de redes tecnológicas. Antes, no obstante, debemos hacer una reseña sobre la información a encontrar:

En la gráfica 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 como direcciones, pero preservando la conectividad de entrada y salida de cada nodo.

Finalmente queremos resaltar el apartado sobre cómo disponer de 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

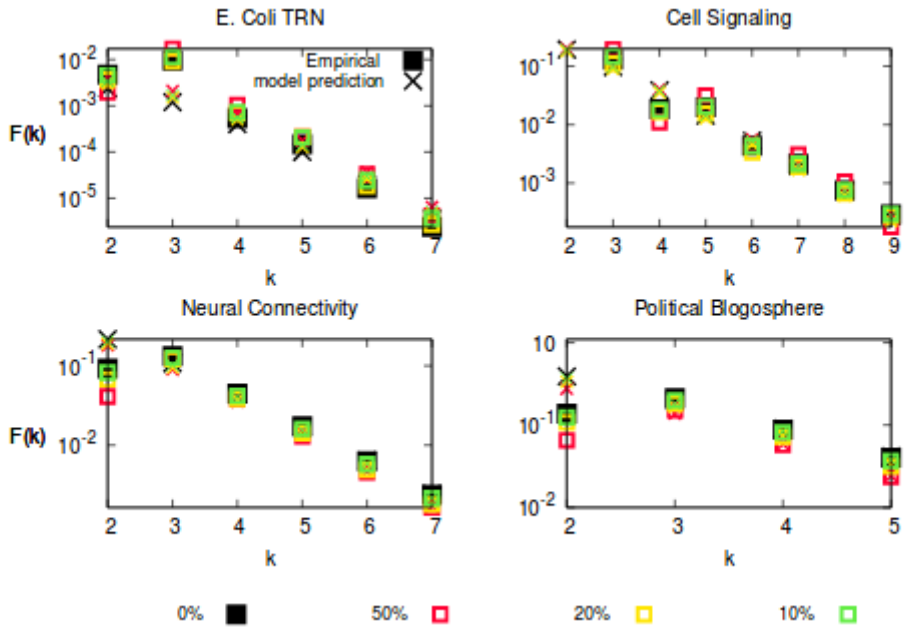

Figura 3: Gráfico extraído del artículo donde resaltamos el impacto de los nodos o conexiones no observadas

nodos que se elimina está entre 20 % y 50 % es posible 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 escoger este artículo para el trabajo. Durante su lectura acordamos que 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 tuviéramos 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.

3.1. Metodología

Una vez elegida la red pasamos a su estudio. Éste se dividió en dar partes: primero buscamos un conjunto de datos sobre los que trabajar y luego se procedió a su análisis. En la primera, inicialmente nos basamos en la base de datos expuesta en el material complementario de [1]. Lamentablemente, el volumen de los datos echaba por tierra cualquier intento por nuestra parte, debido a la poca capacidad computacional 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. Éste se generó partiendo de la red de seguidores/seguidos de uno de los miembros del grupo y ampliándola con las redes de 50 de sus seguidores, elegidos aleatoriamente. Con este proceso, obtuvimos un grafo dirigido con 40375 nodos y 41480 aristas.

Una vez completo el dataset, pasamos a analizar la información mediante el paquete *NetworkX*: En primer lugar, obtuvimos los ciclos estructurales de nuestra red. Para ello utilizamos una versión 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'))
```

Este algoritmo parte de un nodo y viaja la red usando una búsqueda en profundidad (*Depth First Search*) para hallar ciclos. Esta búsqueda termina cuando se recorre toda la región accesible desde el nodo de partida o se alcanza una profundidad máxima especificada, tras lo que se repite el proceso con otro nodo hasta utilizarlos todos.

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

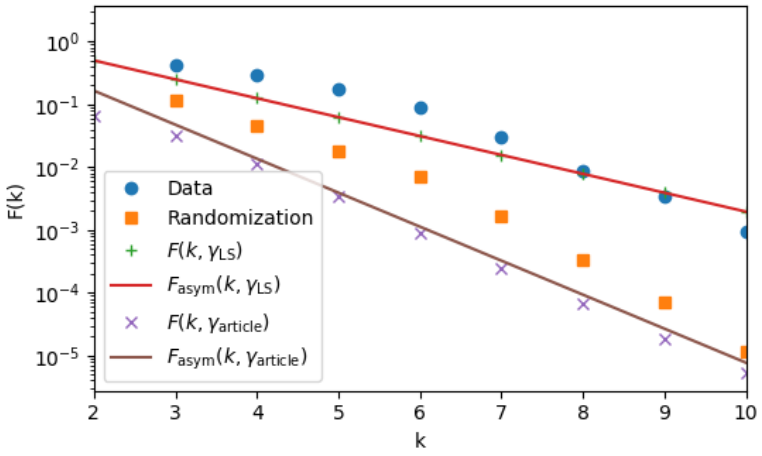

Figura 4: Fracción de los ciclos estructurales en nuestra red de Twitter que también son retroalimentados, $F(k)$, frente a su orden (círculos azules). También se muestra los valores $F(k)$ correspondientes para red aleatorizada (círculos naranjas). Junto a a estos datos se muestra la predicción del modelo $F(k, \gamma)$ para la γ que mejor ajusta a los datos por mínimos cuadrados, γ_{LS} (cruces verdes), y la correspondiente a la del artículo $\gamma_{article}$ (equis moradas). También se muestran las predicciones asintóticas del modelo para estas dos γ (línea roja y línea marrón, respectivamente).

Luego utilizamos otra función implementada en la librería de NetworkX para el conteo de ciclos de retroalimentación en grafos dirigidos, definida como :

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

Una vez obtenida la información sobre la distribución de nodos en nuestra red, pasamos a repetir el análisis del artículo

Como extra, puede consultar el código de nuestro workflow en el apéndice.

3.2. Resultados

Los resultados obtenidos según el proceso anterior se muestran en la figura 4. En ésta se muestran los valores experimentales obtenidos de $F(k)$ para nuestra red de Twitter. Podemos ver que, al igual que en el artículo, nuestra red presenta una $F(k)$ mayor al caso aleatorizado, en contraste con lo que ocurre en redes ecológicas y biológicas. El valor de $F(k)$ para la aleatorización corresponde al método DR y es el resultado de promediar sobre 1000 aleatorizaciones.

Sin embargo, al contrario que en el artículo, ajustando γ al modelo mediante mínimos cuadrados obtenemos $\gamma_{LS} \simeq 0{,}50$, que coincide muy mal con los datos. Esto implica

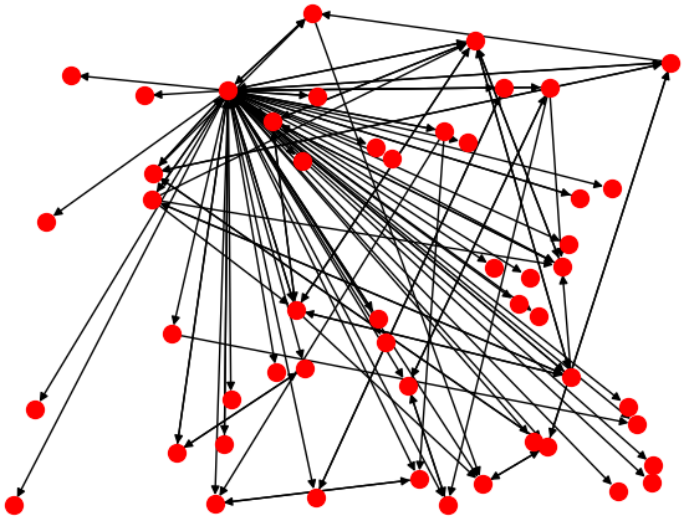

Figura 5: Representación de los nodos con grado de salida mayor que 0 en nuestra red.

que la distribución de ciclos de nuestra red no puede ser explicada mediante el modelo propuesto.

Comparando con los resultados con el γ obtenido en el artículo, $\gamma_{\text{article}} = 0{,}967$, vemos que tampoco hay un buen ajuste, aunque la pendiente para ordenes mayores parece coincidir con la de nuestros datos.

Estos resultados puede explicarse por nuestra muestra de la red de Twitter. Al haber sido generada a partir de la red de un usuario, esta se muestra completamente sesgada a éste. Esto puede observarse claramente en la figura 5, en la que están representados los nodos de nuestra red con orden de salida y de entrada mayor que cero, es decir, aquellos que pueden contribuir a ciclos.

Debido a este sesgo, la mayoría de ciclos contienen a este usuario: 228745 sobre 229062 en el caso de ciclos estructurales y 1052 sobre 1110 en el caso de los retroalimentados. Esta circunstancia incumple claramente la suposición del modelo de que los ciclos sean independientes, lo que justifica el poco acierto del modelo en nuestra reproducción.

A pesar de esto, la coincidencia de la pendiente para k grande al usar la forma asintótica de $F(k, \gamma)$ con γ_{article} parece indicar que, aunque de forma más compleja, la direcciona-

lidad de la red sigue estando presente.

Por otra parte, podemos intentar explicar la abundancia de ciclos con respecto a la red aleatorizada, presente tanto en nuestros datos como en el artículo. En Twitter, además de una clara direccionalidad hacia los usuarios más famosos, similar a lo que ocurre en redes tróficas, existe un gran número de conexiones bidireccionales entre usuarios a un mismo nivel. Estas relaciones, que no existen de forma tan marcada en redes biológicas, serían las causantes del aumento de $F(k)$.

4. Conclusiones finales

Para finalizar, remarcamos algunas de puntos más importantes de nuestro intento de reproducción.

En primer lugar, nos ha llamado la atención la gran necesidad de potencia computacional para resolver el problema. Conceptos aparentemente sencillos como los de ciclos estructurales o dirigidos se convierten en problemas tremendamente costosos a nivel computacional, que hacen irrealizables el análisis de redes relativamente grandes.

Por otra parte hemos visto que la red de Twitter, desde un punto de vista local, no satisface el modelo propuesto del artículo. Sin embargo, la coincidencia de las pendientes en el gráfico de $F(k)$ parece indicar que la direccionalidad de la red también es observable desde esta perspectiva, sólo que requeriría de correcciones para tener en cuenta la no independencia de los diferentes ciclos.

Finalmente, una dirección interesante en la que avanzar sería el estudio de la red de Twitter analizando de forma conjunta tanto su carácter direccional como la asortatividad en las conexiones recíprocas.

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 conversión y resultados

```
# -*- coding: utf-8 -*-

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


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

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

# Removes nodes with out degree 0
remove = [node for node in DG1.nodes if DG1.out_degree(node) <= 0]
DG1.remove_nodes_from(remove)

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


# Count cycles
# ============================
# This function 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 get all loops via networkx function
cycle_list_DG1 = simple_cycles_undirected(undirected1, maxlength=10)


# Saves cycles to file
# ====================================
# Saves directed cycles
savecycles("sevaseviene_directedcycles.dat", feedback_cycle_list_DG1)
# Saves 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)
```

C. Biblioteca con las funciones implementadasa

```
import csv
import networkx as nx
import numpy as np
from collections import defaultdict

# 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]
    _.

    Note: this functions counts loops of size bigger than 2 twice, one
    for each direction.

    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
    G_dir = G.to_directed()
    subG = type(G_dir)(G_dir.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) and (len(path)>2):
                    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

def F(k,gamma):
    """
    Equation that give the theoretical approximation of the fraction loops
    thata
    are feedback loops
    k= loop length
    gamma= parameter

    returns the result
    """
    return 2*np.exp(k/2*np.log(gamma*(1-gamma))+k/24*(np.log(gamma/(1-gamma
    )))**2)


def cyclic_eulerian_novec(l, k):
    """Calculate the cyclic Eulerian number of order k for l ascents.

    """
    # Check if the result have been memoized
    if (l, k) in cyclic_eulerian_novec.cache:
        result = cyclic_eulerian_novec.cache[(l, k)]
    # Else, calculate it
    else:
        if (l == 0) or (l >= k):
            result = 0
        else:
            result = (float(k*(k - l))/(k - 1)*cyclic_eulerian(l - 1, k -
    1)
                    + float(l*k)/(k - 1)*cyclic_eulerian(l, k - 1))

            # Store the result in the cache
            cyclic_eulerian_novec.cache[(l, k)] = result
```

```
    return result

cyclic_eulerian_novec.cache = {(0, 1) : 1}

cyclic_eulerian = np.vectorize(cyclic_eulerian_novec)

def F_exact_novec(k, gamma):
    """Calculate the exact value of the fraction of feedback loops.

    Note: this function is not vectorized.

    """
    suma = 0
    for l in range(int(k) + 1):
        suma += float(cyclic_eulerian(l, k))/np.math.factorial(k)*(
                np.power(gamma, l)*np.power(1 - gamma, k - l)
                + np.power(gamma, k - l)*np.power(1 - gamma, l))

    return suma

# Vectorize function
F_exact = np.vectorize(F_exact_novec)
```