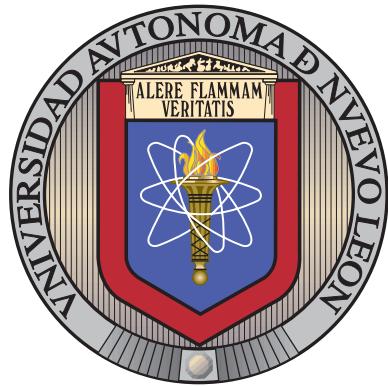


UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN

FACULTAD DE INGENIERÍA MECÁNICA Y ELÉCTRICA
POSGRADO EN INGENIERÍA DE SISTEMAS

OPTIMIZACIÓN DE FLUJO EN REDES
DRA. ELISA SCHAEFFER



RETROALIMENTACIÓN DE LAS TAREAS
REALIZADAS DURANTE EL PERÍODO
ENERO–JUNIO DE 2019

POR

DAYLI MACHADO DE ARMAS

1985275

SAN NICOLÁS DE LOS GARZA, NUEVO LEÓN

JUNIO 2019

1. Introducción

A continuación se muestra el portafolio con las correcciones realizadas en las tareas de clases.

10

Tarea No.1: Flujo en Redes

Dayli Machado de Armas

52/25

11 de febrero de 2019

1. Gráfo simple no dirigido acíclico

Un grafo simple no dirigido acíclico, es aquel que no posee dirección en sus aristas, ni bucles, y por supuesto no es reflexivo.^[1]

Este tipo de grafos suele representarse por una secuencia de nodos unidos por aristas, donde de cada nodo generalmente solo sale, o llega una arista.^[2]

En la práctica considero que este tipo de grafos puede emplearse para representar estructuras moleculares que sigan este comportamiento, por ejemplo, la representación de moléculas. Un ejemplo de estas moléculas sería la representación de alcanos de tipo lineal, donde los átomos serían los nodos y los enlaces entre estos las aristas. Este tipo de moléculas seleccionadas no poseen ciclos, ni bucles en su representación. Ver Figura 1 en la página 2 donde se muestra la representación gráfica del mismo.

```
' import matplotlib.pyplot as plt
` import networkx as nx
> H=nx.Graph()
> H.add_path([0,1,2,3,4,])
< nx.draw (H)

` plt.savefig("imagenes/Fig01.eps")'
```

2. Gráfo simple no dirigido cíclico

Un grafo simple no dirigido cíclico ~~X~~ es aquel que no posee dirección en sus aristas, pero si se forma un ciclo que represente una figura cerrada que comience y termine en el mismo vértice, entonces es llamado cíclico^[3]. En los grafos no dirigidos, el flujo puede fluir en ambas direcciones.

el gato come

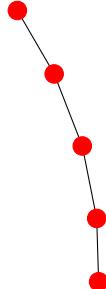


Figura 1: Representación de grafo simple no dirigido acíclico

Este tipo de grafos suele representarse por una secuencia de nodos unidos por aristas, pero estos pueden tener dentro un ciclo, o ser un ciclo en sí mismo. En la práctica, estos poseen múltiples aplicaciones, por ejemplo pudiera decirse que la topología de redes circular, es un ejemplo de aplicación de este tipo de grafo, otro ejemplo pudiera ser la representación de las autopistas de una región dada, o también la representación de otro tipo de moléculas de alcanos, en este caso serían las de alcanos que se representan como ciclos, siendo el caso del ciclo pentano por ejemplo, que serían cinco nodos unidos por aristas de manera circular.



El ejemplo a representar **se me ocurre que pudiera ser**, las llamadas que se realizan interdepartamentales en una muestra de tiempo de la jornada laboral en una empresa determinada, supongamos el departamento de producción, de marketing, comercial, calidad y planificación de la producción, donde cada nodo representa un departamento dentro de la empresa o área de esta, y las aristas las llamadas que se realizan, estas pueden tener un comportamiento que al graficarlo represente un grafo simple no dirigido cíclico. **Y si** utilidad pudiera ser analizar el comportamiento del grafo para minimizar la cantidad de llamadas, o el tiempo de estas por ejemplo. Ver ~~Figura 2~~ en la página 3 donde se muestra la representación gráfica del mismo.

```

import matplotlib.pyplot as plt
import networkx as nx

G = nx.Graph()

G.add_nodes_from([0,1,2,3,4])
pos = nx.spring_layout(G)

G.add_edge(0,1)
G.add_edge(0,3)
G.add_edge(1,4)
G.add_edge(1,2)
G.add_edge(3,1)

```

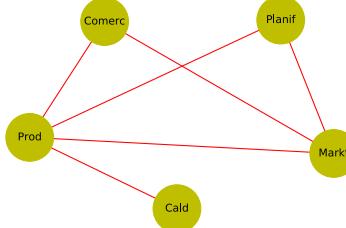


Figura 2: Representación de grafo simple no dirigido cíclico

```

G.add_edge(0,4)

nx.draw_networkx_nodes(G, pos, node_size=2000, node_color='y',
                      node_shape='o')
nx.draw_networkx_edges(G, pos, width=1, alpha=0.5, edge_color='r')

labels = {}
labels[0] = r'Markt'
labels[1] = r'Prod'
labels[2] = r'Cald'
labels[3] = r'Comerc'
labels[4] = r'Planif'

plt.axis('off')

nx.draw_networkx_labels(G, pos, labels, font_size=10)
plt.savefig("imagenes/Fig02.eps")

plt.show()

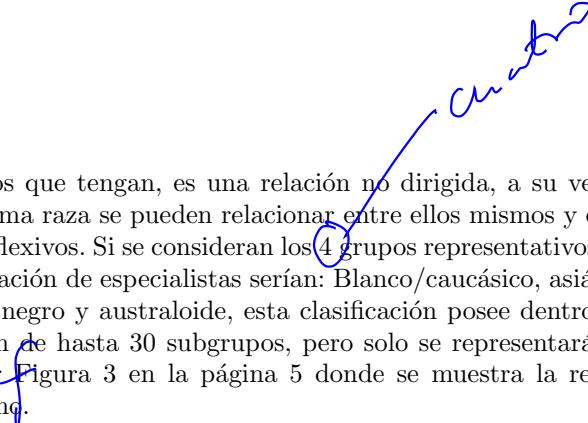
```

3. Gráfo simple no dirigido reflexivo

Un grafo simple ~~X~~ en el cual no se permiten aristas múltiples, además será no dirigido y reflexivo, si no poseen dirección en sus aristas y reflexivo si cuenta al menos con un bucle, lo que se refiere a una arista reflexiva en la que coinciden el vértice de origen y el vértice de destino ~~X~~[1].

Este grafo pudiera emplearse para representar el comportamiento del entrecruzamiento entre diferentes grupos raciales, donde cada grupo representa un nodo diferente, pudiendo existir cruzamiento entre ellos, y las relaciones entre los gru-

pos son los hijos que tengan, es una relación no dirigida, a su vez dentro del grupo de la misma raza se pueden relacionar entre ellos mismos y estarían ocurriendo lazos reflexivos. Si se consideran los 4 grupos representativos de las razas según la clasificación de especialistas serían: Blanco/caucásico, asiático/mongo-loide, negroide/negro y australoide, esta clasificación posee dentro cada grupo una clasificación de hasta 30 subgrupos, pero solo se representarán los cuatro principales. Ver Figura 3 en la página 5 donde se muestra la representación gráfica del mismo.



```

import matplotlib.pyplot as plt
import networkx as nx

G = nx.Graph()
G.add_nodes_from([1,2,3,4])

node_b = {1}
node_a = {2}
node_n = {3}
node_au = {4}

pos = nx.spring_layout(G)
#pos = {1:(5000,8020), 2:(7150,6620), 3:(6675, 5180), 4:(5475,8200)}

G.add_edges_from([(1,1), (1,2), (1,3), (1,4)])# Cada nodo es reflexivo
  pero no me sale la flechita
G.add_edges_from([(2,2), (2,1), (2,3), (2,4)])#igualmente no sale el
  reflexivo en si mismo
G.add_edges_from([(3,3), (3,1), (3,2), (3,4)])#no sale el reflexivo en
  si mismo
G.add_edges_from([(4,4), (4,1), (4,3), (4,2)])#es reflexivo en si mismo

nx.draw_networkx_nodes(G, pos, nodelist=node_b, node_size=3000,
  node_color='g', node_shape='o')
nx.draw_networkx_nodes(G, pos, nodelist=node_a, node_size=3000,
  node_color='y', node_shape='o')
nx.draw_networkx_nodes(G, pos, nodelist=node_n, node_size=3000,
  node_color='b', node_shape='o')
nx.draw_networkx_nodes(G, pos, nodelist=node_au, node_size=3100,
  node_color='r', node_shape='o')

nx.draw_networkx_edges(G, pos, width=1, alpha=0.8, edge_color='black', )

labels = {}
labels[1] = r'Blanco'
labels[2] = r'Asiatico'
labels[3] = r'Negro'
labels[4] = r'Australoide'

plt.axis('off')

```

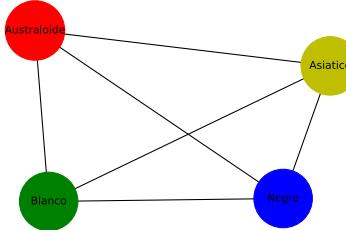


Figura 3: Representación de grafo simple no dirigido reflexivo

```

nx.draw_networkx_labels(G, pos, labels, font_size=10)

#plt.xlim (2000,16000)
#plt.autoscale (enable=true, axis='both')
plt.savefig("imagenes/Fig03.eps", bbox_inches='tight')
#nx.draw(G)
plt.show()

```

4. Gráfo simple dirigido acíclico

Un grafo simple dirigido acíclico, es aquel que posee una dirección en sus aristas, y no posee bucles o reflexividad, ni ciclos^[1].

En este caso, ejemplos de la vida real serían aquellos que posean un origen del que puede salir una o varias aristas por diferentes caminos, sin que entre ellas existan ciclos y tengan un destino final diferente al del origen. Los árboles genealógicos, los organigramas en las empresas, el flujo de procesos industriales que no posean ciclos, que posean un inicio y un fin, pudieran resultar ser ejemplos de la vida real en los que se puede aplicar este tipo de grafos. Ver Figura 4 en la página 6 donde se muestra la representación gráfica del mismo.

```

import matplotlib.pyplot as plt
import networkx as nx

G = nx . DiGraph ()

G.add_node(1)
G.add_node(2)
G.add_node(3)
G.add_node(4)
G.add_node(5)

```

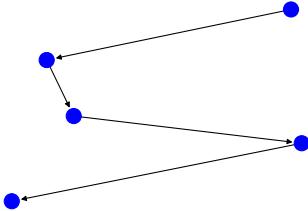


Figura 4: Representación de gráfo simple dirigido acíclico

```

G.add_edge(1,2)
G.add_edge(2,3)
G.add_edge(3,4)
G.add_edge(4,5)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos, node_size=200, node_color='b',
                      node_shape='o')
nx.draw_networkx_edges(G, pos, width=1, alpha=0.8, edge_color='black')
plt.axis('off')

plt.savefig("imagenes/Fig04.eps")
plt.show()

```

5. Gráfo simple dirigido cíclico

Un grafo simple dirigido cíclico ~~X~~ es aquel que posee una dirección en sus aristas, y posee una figura cerrada que comienza y termina en el mismo vértice.

En este caso, ejemplos de la vida real ~~serían~~ aquellos que posean un origen del que puede salir una o varias aristas por ~~diferentes~~ diferentes caminos, y que en un determinado nodo regresen a alguno anterior de modo que se forme uno o más ciclos. El destino final ~~X~~ deberá ser diferente al del origen.

Como aplicación real de este tipo de grafo, puede ser el flujo de procesos industriales o artesanales que posean ciclos, por ejemplo: la elaboración artesanal de jugo de naranja, al representar este proceso en un flujograma mediante un diagrama OPERIN, en el que se reflejen sus operaciones. Una vez que se llegue al paso de exprimir las naranjas, se caería en un ciclo volviendo a la operación anterior de seleccionar otra y otra naranja hasta cubrir toda la capacidad del extractor de jugo. En este ejemplo las operaciones serían los nodos y los enlaces serían la transformación que va teniendo la naranja. Pudiera representarse co-

mo se muestra en la  figura 5 en la página 8 donde se muestra la representación gráfica del mismo.

```
import matplotlib.pyplot as plt
import networkx as nx

G = nx . DiGraph ()

G.add_node(1)
G.add_node(2)
G.add_node(3)
G.add_node(4)
G.add_node(5)
G.add_node(6)

pos = {1:(100, 0), 2:(150,0), 3:(200, 0), 4:(250,0), 5:(300,0),
       6:(350,0)}

G.add_edge(1,2)
G.add_edge(2,3)
G.add_edge(3,4)
G.add_edge(4,5)
G.add_edge(5,6)

G.add_cycle([5,4])
#G.add_cycle (5,4)

nx.draw_networkx_nodes(G, pos, node_size=600, node_color='r',
                      node_shape='o')
nx.draw_networkx_edges(G, pos, width=1, alpha=0.8, edge_color='black')

labels = []
labels[1] = r'$\text{O}_1$'
labels[2] = r'$\text{O}_2$'
labels[3] = r'$\text{O}_3$'
labels[4] = r'$\text{O}_4$'
labels[5] = r'$\text{O}_5$'
labels[6] = r'$\text{O}_6$'

nx.draw_networkx_labels(G, pos, labels, font_size=10)

plt.axis('off')

plt.savefig("imagenes/Fig05.eps")

plt.show()
```



Figura 5: Representación de gráfo simple dirigido cíclico

6. Gráfo simple dirigido reflexivo

La diferencia con este grafo y el anterior reflexivo visto, es que en este caso las aristas si tienen sentido y el flujo que se analice debe ir en una dirección. En este caso deberá existir un nodo que se llame a sí mismo, o sea, que posea al menos un bucle.

Un ejemplo en el que pudiera emplearse la teoría de grafos, y que pudiera seguir un comportamiento dirigido reflexivo es al analizar entre personas de grupos sanguíneos diferentes, representar de quien pueden recibir sangre en función del tipo que posean. La reflexividad está dada en los casos en que cada grupo sanguíneo puede recibir de otros grupos y de sí mismo, excepto el grupo O- que solo puede recibir del mismo tipo.

Pudiera representarse como se muestra en la Figura 6 en la página 10 donde se muestra la representación gráfica del mismo.

```

import matplotlib.pyplot as plt
import networkx as nx

G = nx.DiGraph()

G.add_node(1)
G.add_node(2)
G.add_node(3)
G.add_node(4)
G.add_node(5)
G.add_node(6)
G.add_node(7)

```

```

G.add_node(8)

node_a = {1,2}
node_b = {3,4}
node_ab = {5,6}
node_o = {7,8}

pos = nx.spring_layout(G)

G.add_edges_from([(1,1), (8,1), (7,1), (2,1)])# Cada nodo es reflexivo
    pero no me sale la flechita
G.add_edges_from([(2,2), (8,2)])#igualmente no sale el reflexivo en si
    mismo
G.add_edges_from([(3,3), (8,3), (7,3), (4,3)])#no sale el reflexivo en
    si mismo
G.add_edges_from([(4,4), (8,4)])#es reflexivo en si mismo
G.add_edges_from([(5,5), (1,5), (2,5), (3,5), (4,5), (6,5), (7,5),
    (8,5)])# es reflexivo en si mismo
G.add_edges_from([(6,6), (4,6), (8,6), (2,6)])
G.add_edges_from([(7,7), (8,7)])
G.add_edges_from([(8,8)])

nx.draw_networkx_nodes(G, pos, nodelist=node_a, node_size=800,
    node_color='g', node_shape='o', alpha=0.3)
nx.draw_networkx_nodes(G, pos, nodelist=node_b, node_size=800,
    node_color='y', node_shape='o', alpha=0.3)
nx.draw_networkx_nodes(G, pos, nodelist=node_ab, node_size=800,
    node_color='b', node_shape='o', alpha=0.3)
nx.draw_networkx_nodes(G, pos, nodelist=node_o, node_size=800,
    node_color='r', node_shape='o', alpha=0.8)
#nx.draw_networkx_nodes(G, pos, nodelist=node_o, node_size=800,
    node_color='r', node_shape='on', alpha=0.1)
nx.draw_networkx_edges(G, pos, width=1, alpha=0.8, edge_color='black', )

labels = {}
labels[1] = r'A+'
labels[2] = r'A-'
labels[3] = r'B+'
labels[4] = r'B-'
labels[5] = r'AB+'
labels[6] = r'AB-'
labels[7] = r'O+'
labels[8] = r'O-'

plt.axis('off')
|
| nx.draw_networkx_labels(G, pos, labels, font_size=10)
|
| plt.savefig("imagenes/Fig06.eps")

```

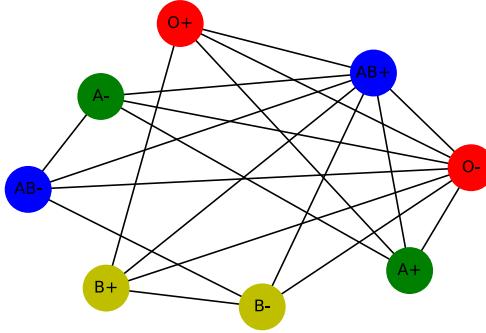


Figura 6: Representación de gráfo simple dirigido reflexivo

```
plt.show()
```

7. Multigrafo no dirigido acíclico

Cuando haya más de una arista entre un par de vértices, el grafo se llama multígrafo. Si no se permiten aristas múltiples, el grafo es simple^[1]. Será no dirigido cuando las aristas no poseen sentido restringido, sino que el flujo fluye en ambas direcciones, y en este caso no presenta ciclos.

Un ejemplo práctico de multígrafo no dirigido acíclico pudiera decirse que es la representación de determinados enlaces moleculares, como ejemplo pudiera citarse el tipo de elemento que posea un enlace doble éster. Para la representación de los elementos con este tipo de enlace puede emplearse el grafo en cuestión. Específicamente pudiera decirse que la representación de éster sulfúrico pudiera representarse con este comportamiento, como se muestra en la Figura 7 en la página 12 donde se muestra la representación gráfica del mismo.

```

import matplotlib.pyplot as plt
import networkx as nx

G=nx.MultiGraph()

G.add_node(1)
G.add_node(2)
G.add_node(3)
G.add_node(4)

```

```

G.add_node(5)
G.add_node(6)
G.add_node(7)

node_o = {4,5,6,7}
node_r = {1,2,3}

G.add_edge(4,3, weight=3)
G.add_edge(4,3, weight=5)
G.add_edge(3,7, weight=3)
G.add_edge(3,7, weight=5)
G.add_edge(1,5)
G.add_edge(5,3)
G.add_edge(3,6)
G.add_edge(6,2)

blue=[(4,3),(3,7),(1,5),(5,3),(3,6),(6,2)]
red=[(4,3),(3,7)]

pos = {1:(0,220), 2:(150,220), 3:(75, 180), 4:(75,200),
       5:(25,180),6:(125,180), 7:(75,160)}

nx.draw_networkx_nodes(G, pos, nodelist=node_o, node_size=600,
                      node_color='b', node_shape='o')
nx.draw_networkx_nodes(G, pos, nodelist=node_r, node_size=600,
                      node_color='y', node_shape='o')

nx.draw_networkx_edges(G, pos, edgelist=blue, width=6, alpha=0.5,
                      edge_color='b', style='dashed')
nx.draw_networkx_edges(G, pos, edgelist=red, width=6, alpha=0.5,
                      edge_color='r')

# Labels
labels = []
labels = {}
labels[1] = r'R1'
labels[2] = r'R2'
labels[3] = r'S'
labels[4] = r'0'
labels[5] = r'0'
labels[6] = r'0'
labels[7] = r'0'

plt.axis('off')

nx.draw_networkx_labels(G, pos, labels, font_size=10)

pos = nx.spring_layout(G)

plt.savefig("imagenes/Fig07.eps")
plt.show()

```

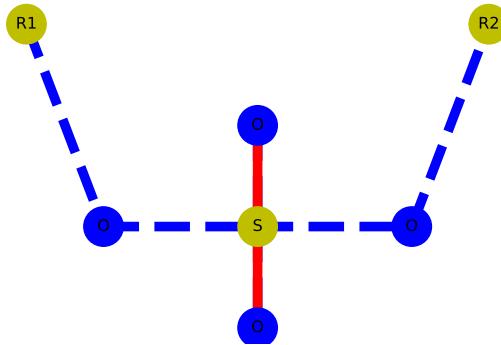


Figura 7: Representación de multigrafo no dirigido acíclico

8. Multigrafo no dirigido cíclico

Cuando haya más de una arista entre un par de vértices, el grafo se llama multígrafo. En este caso será no dirigido cuando las aristas no poseen sentido restringido, sino que el flujo fluye en ambas direcciones, y deberán existir zonas cerradas, o sea, que regresen al nodo inicial.

Este tipo de grafo es de gran utilidad para representar el comportamiento de redes sociales y neuronales por ejemplo. El ejemplo práctico que se me ocurre mostrar, es cuando tenemos necesidad de comunicarnos con alguien y las vías de comunicación que tenemos en la actualidad, donde cada vía de comunicación seleccionada tendrá un costo (peso), y existen muchas vías para comunicarnos con otras personas, y a su vez para que esa persona se comunique con nosotros, por lo que no es dirigido, pero sí cíclico, debido a los lazos que se pueden formar entre las personas que necesitan comunicarse. El ejemplo se restringe a solo dos vías de comunicación, ya sea por whatsapp, o por llamadas telefónicas, aunque en la práctica existen muchas más. Este grafo se muestra en la Figura 8 en la página 14 donde se muestra la representación gráfica del mismo.

```

import matplotlib.pyplot as plot
import networkx as nx

G = nx . MultiGraph()
G.add_node(1)
G.add_node(2)

```

```

G.add_node(3)
G.add_node(4)
G.add_node(5)

G.add_edge(1,2, weight=3)
G.add_edge(1,2, weight=4)
G.add_edge(2,3, weight=3)
G.add_edge(2,3, weight=4)
G.add_edge(3,4, weight=3)
G.add_edge(3,4, weight=4)
G.add_edge(4,5, weight=3)
G.add_edge(4,5, weight=4)
G.add_edge(5,1, weight=3)
G.add_edge(5,1, weight=4)
G.add_edge(1,3, weight=3)
G.add_edge(1,3, weight=4)
G.add_edge(5,3, weight=3)
G.add_edge(5,3, weight=4)

green=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]
yellow=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]

pos = nx.spring_layout(G)

plt.axis('off')

nx.draw_networkx_nodes(G, pos, node_size=400, node_color='black',
node_shape='o')

nx.draw_networkx_edges(G, pos, edgelist=green, width=3, alpha=0.5,
edge_color='g', style='dashed')
nx.draw_networkx_edges(G, pos, edgelist=yellow, width=4, alpha=0.5,
edge_color='y')

plt.savefig("imagenes/Fig08.eps")

plt.show()

```

9. Multigrafo no dirigido reflexivo

(*última lección X*)

Este caso se refiere a un multígrafo en el que como se había mencionado existe más de una arista entre un par de vértices, sin dirección entre estas, pero de un nodo deben salir más de una arista que regresen al mismo nodo, sin pasar por otro.

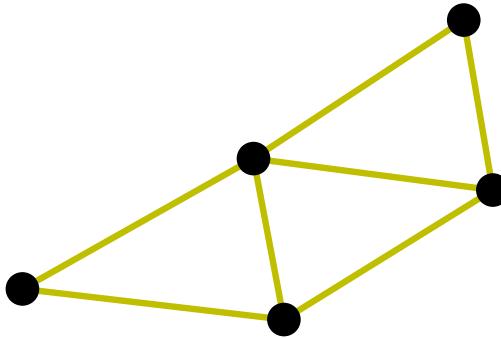


Figura 8: Representación de Multigrafo no dirigido cíclico

Una aplicación pudiera ser la relación que existe entre 5 signos zodiacales, donde cada nodo representa a las personas de un signo determinado y de cada uno de ellos puede salir una arista que represente las relaciones positivas entre los diferentes signos y otra las relaciones negativas, y a su vez las personas del mismo signo se relacionarán con sus iguales de manera positiva o negativa, lo que representa la reflexibilidad. Este grafo se muestra, para una muestra de 5 signos cualesquiera, en la Figura 9 en la página 16 donde se muestra la representación gráfica del mismo.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Feb 11 16:09:42 2019

@author: lapi7
"""

import matplotlib.pyplot as plot
import networkx as nx

G = nx.MultiGraph()

G.add_node(1)
G.add_node(2)
G.add_node(3)
G.add_node(4)
G.add_node(5)

G.add_edge(1,2, weight=4)
G.add_edge(2,3, weight=3)
```

```

G.add_edge(2,3, weight=4)
G.add_edge(3,4, weight=3)
G.add_edge(3,4, weight=4)
G.add_edge(4,5, weight=3)
G.add_edge(4,5, weight=4)
G.add_edge(5,1, weight=3)
G.add_edge(5,1, weight=4)
G.add_edge(1,3, weight=3)
G.add_edge(1,3, weight=4)
G.add_edge(5,3, weight=3)
G.add_edge(5,3, weight=4)

node_A = {1}
node_S = {2}
node_E = {3}
node_V = {4}
node_C = {5}

green=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]
yellow=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]

pos = nx.spring_layout(G)

plt.axis('off')

nx.draw_networkx_nodes(G, pos, nodelist=node_A, node_size=300,
                      node_color='g', node_shape='o')
nx.draw_networkx_nodes(G, pos, nodelist=node_S, node_size=300,
                      node_color='y', node_shape='o')
nx.draw_networkx_nodes(G, pos, nodelist=node_E, node_size=300,
                      node_color='b', node_shape='o')
nx.draw_networkx_nodes(G, pos, nodelist=node_V, node_size=300,
                      node_color='r', node_shape='o')
nx.draw_networkx_nodes(G, pos, nodelist=node_C, node_size=300,
                      node_color='black', node_shape='o')

#nx.draw_networkx_nodes(G, pos ,node_size=400, node_color='black',
#                      node_shape='o')
#nx.draw_networkx_nodes(G, pos ,node_size=400, node_color='black',
#                      node_shape='o')
#nx.draw_networkx_nodes(G, pos ,node_size=400, node_color='black',
#                      node_shape='o')
#nx.draw_networkx_nodes(G, pos ,node_size=400, node_color='black',
#                      node_shape='o')
#nx.draw_networkx_nodes(G, pos ,node_size=400, node_color='black',
#                      node_shape='o')

nx.draw_networkx_edges(G, pos, edgelist=green, width=3, alpha=0.5,
edge_color='g', style='dashed')

```

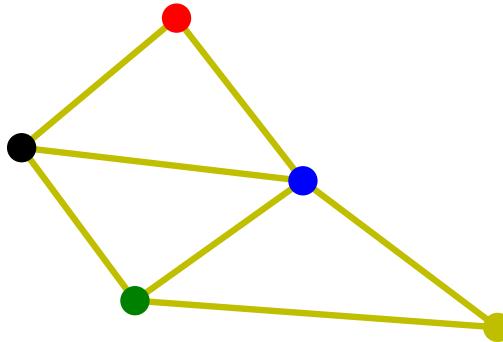


Figura 9: Representación de Multigrafo no dirigido reflexivo

```

nx.draw_networkx_edges(G, pos, edgelist=yellow, width=4, alpha=0.5,
edge_color='y')

plt.savefig("imagenes/Fig09.eps")

plt.show()

```

10. Multigrafo dirigido acíclico

Para este tipo de grafo deben existir más de una arista entre un par de nodos, con dirección y no debe formarse ninguna figura cerrada dentro del grafo.

Este ejemplo aplicado a la práctica pudiera ser la representación de un viajero que desea ir de una ciudad a otra para visitarlas y tiene que elegir entre varias rutas posibles para llegar, o pudiera elegir también entre diferentes medios de transporte para trasladarse entre las ciudades, o una combinación de ambos. En este caso las ciudades serían los nodos, y las rutas o medios de transporte posibles a elegir entre un nodo u otro serían las aristas. Una ruta pudiera ser ir de la ciudad de Matanzas en Cuba, a la Habana, de esta a Monterrey, de Monterrey a Torreón y de ahí a Sinaloa. Este grafo se muestra en la Figura 10 en la página 18 donde se muestra la representación gráfica del mismo.

```

import matplotlib.pyplot as plt
import networkx as nx

```

```

G = nx . MultiDiGraph ()

G.add_node(1)
G.add_node(2)
G.add_node(3)
G.add_node(4)
G.add_node(5)

G.add_edge(1,2, weight=3)
G.add_edge(1,2, weight=4)
G.add_edge(2,3, weight=3)
G.add_edge(2,3, weight=4)
G.add_edge(3,4, weight=3)
G.add_edge(3,4, weight=4)
G.add_edge(4,5, weight=3)
G.add_edge(4,5, weight=4)

medio=[(1,2),(2,3),(3,4),(4,5)]
ruta=[(1,2),(2,3),(3,4),(4,5)]

pos = nx.spring_layout(G)

nx.draw_networkx_nodes(G, pos, node_size=500, node_color='b',
node_shape='o')

nx.draw_networkx_edges(G, pos, edgelist=medio, width=6, alpha=0.8,
edge_color='black', style='dashed')

nx.draw_networkx_edges(G, pos, edgelist=ruta, width=4, alpha=0.5,
edge_color='r')

labels = {}
labels[1] = r'Mat'
labels[2] = r'Hab'
labels[3] = r'Monty'
labels[4] = r'Torr'
labels[5] = r'Sing'

plt.axis('off')

nx.draw_networkx_labels(G, pos, labels, font_size=10)

plt.savefig("imagenes/Fig10.eps")
plt.show()

```

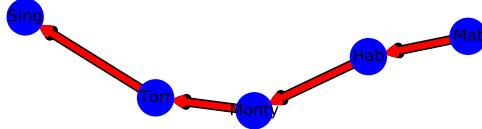


Figura 10: Representación de Multigrafo dirigido acíclico

11. Multigrafo dirigido cíclico

Debe existir más de una arista entre un par de nodos, con dirección y además formarse al menos una figura cerrada dentro del grafo.

Un ejemplo que representa este tipo de grafo sería la representación del flujo que siguen los pacientes una vez que asisten a la consulta de cuerpo de guardia del ISSSTE, en la que las operaciones del proceso serían los nodos y la clasificación del tipo de paciente y su recorrido dentro de la instalación las aristas. En este caso el cliente al llegar pasa por la consulta de clasificación y ahí le asignan un color en función de la gravedad de su dolencia, se asumen tres colores, rojo, amarillo y verde, que van de mayor gravedad a menor respectivamente y de esta dependerá el tiempo de espera para pasar a la siguiente consulta donde se encuentra el doctor, existe tres consultas con doctores disponibles y de ahí pueden pasar al laboratorio, u a otras de las salas. Luego, del laboratorio pueden pasar nuevamente a la consulta del doctor para revisar los resultados y ocurriría un ciclo, o del doctor directo a la sala, luego al laboratorio y ~~etornar~~ nuevamente al doctor, y ocurriría otro ciclo. Este grafo se muestra en la Figura 11 en la página 20 donde se muestra la representación gráfica del mismo.

```

import matplotlib.pyplot as plt
import networkx as nx

G = nx.MultiDiGraph()

G.add_node(1)
G.add_node(2)
  
```

```

G.add_node(3)
G.add_node(4)
G.add_node(5)

G.add_edge(1,2, weight=3)
G.add_edge(1,2, weight=4)
G.add_edge(1,2, weight=5)
G.add_edge(2,3, weight=3)
G.add_edge(2,3, weight=4)
G.add_edge(2,4, weight=3)

G.add_edge(2,5, weight=3)
G.add_edge(3,2, weight=4)
G.add_edge(3,2, weight=3)

y=[(1,2),(2,3),(3,2)]
g=[(1,2),(2,3),(3,2)]
r=[(1,2),(2,4),(2,5)]

pos = nx.spring_layout(G)

nx.draw_networkx_nodes(G, pos, node_size=400, node_color='b',
node_shape='o')

nx.draw_networkx_edges(G, pos, edgelist=y, width=10, alpha=0.2,
edge_color='y', style='dashed')

nx.draw_networkx_edges(G, pos, edgelist=g, width=6, alpha=0.5,
edge_color='g')

nx.draw_networkx_edges(G, pos, edgelist=r, width=2, alpha=0.8,
edge_color='r')

labels = {}
labels[1] = r'1'
labels[2] = r'2'
labels[3] = r'Lab'
labels[4] = r'3'
labels[5] = r'4'

plt.axis('off')

nx.draw_networkx_labels(G, pos, labels, font_size=10)

plt.savefig("imagenes/Fig11.eps")
plt.show()

```

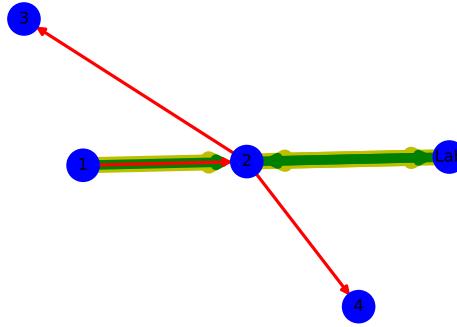


Figura 11: Representación de Multigrafo dirigido cíclico

12. Multigrafo dirigido reflexivo

— Debe existir más de una arista entre un par de nodos, con dirección y además cada nodo debe llamarse a sí mismo.

Un ejemplo en el que el empleo de la teoría de grafos como los que se analizan en esta sección puede emplearse es en el estudio de la influencia de la religión en la sociedad, como un subejemplo de la aplicación en redes sociales [3]. Por ejemplo, si cada nodo es una persona religiosa creyente, ese nodo (persona) constantemente se realiza un autoexamen de conciencia a sí mismo, evaluando cómo ha sido su actitud con respecto a las variables a analizar en este autoexamen. Estas variables pueden ser operacionalizadas como cada una de las doctrinas de cada religión, las cuales varían de una religión a otra, pero son más de una y pueden agruparse por categorías. Estas variables representan las aristas, cada una puede tener un peso determinado según sea el interés del estudio, a su vez estas mismas doctrinas (convertidas en variable) serán las que cada persona creyente se encargará de transmitir a otras personas creyentes o no, pudiera decirse que se transmite el tipo de actitud que se debe tener ante cada una de estas variables de una persona a otra. En este ejemplo se puede evaluar el impacto en un grupo poblacional que puede tener un tipo de creencia u otro, en la medida en que aumente la red. Una simplificación de este tipo de grafo se representa en la Figura 12 en la página 22 donde se muestra la representación gráfica del mismo.

```
import matplotlib.pyplot as plt
import networkx as nx
```

```

G = nx . MultiDiGraph ()

G.add_node(1)
G.add_node(2)
G.add_node(3)
G.add_node(4)
G.add_node(5)

a= {1}
b= {2}
c= {3}
d= {4}
e= {5}

G.add_edge(1,2, weight=3)
G.add_edge(1,2, weight=4)
G.add_edge(1,2, weight=5)
G.add_edge(2,3, weight=3)
G.add_edge(2,3, weight=4)
G.add_edge(2,3, weight=4)
G.add_edge(3,4, weight=3)
G.add_edge(3,4, weight=3)
G.add_edge(3,4, weight=3)
G.add_edge(4,5, weight=3)
G.add_edge(4,5, weight=3)
G.add_edge(4,5, weight=3)

y=[(1,2),(2,3),(3,4),(4,5)]
g=[(1,2),(2,3),(3,4),(4,5)]
r=[(1,2),(2,3),(3,4),(4,5)]

pos = nx.spring_layout(G)

nx.draw_networkx_nodes(G, pos, nodelist=a, node_size=300,
    node_color='black', node_shape='o')
nx.draw_networkx_nodes(G, pos, nodelist=b, node_size=300,
    node_color='y', node_shape='o')
nx.draw_networkx_nodes(G, pos, nodelist=c, node_size=300,
    node_color='b', node_shape='o')
nx.draw_networkx_nodes(G, pos, nodelist=d, node_size=300,
    node_color='r', node_shape='o')
nx.draw_networkx_nodes(G, pos, nodelist=e, node_size=300,
    node_color='g', node_shape='o')

nx.draw_networkx_edges(G, pos, edgelist=y, width=8, alpha=1,
edge_color='y', style='dashed')

nx.draw_networkx_edges(G, pos, edgelist=g, width=6, alpha=0.5,
edge_color='g')

```

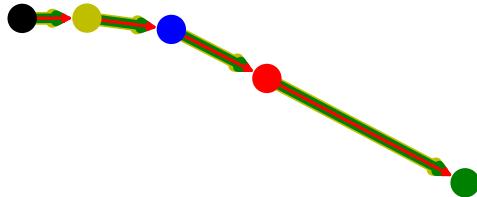


Figura 12: Representación de Multigrafo dirigido reflexivo

```
nx.draw_networkx_edges(G, pos, edgelist=r, width=2, alpha=0.8,
edge_color='r')
```

```
plt.axis('off')
```

```
plt.savefig("imagenes/Fig12.eps")
plt.show()
```

Referencias

- [1] *Complejidad computacional de problemas y el análisis y diseño de algoritmos.* 2017.
- [2] Pieter Swart Aric Hagberg, Dan Schult. *NetworkX Reference, Release 2.3rc1.dev20190113142952*, 2019. *URL = [...]*
- [3] Anwesha Chakraborty, Trina Dutta, Sushmita Mondal, and Asoke Nath. Application of graph theory in social media. **INTERNATIONAL JOURNAL OF COMPUTER SCIENCES AND ENGINEERING**, 6:722–729, 10 2018. doi: 10.26438/ijcse/v6i10.722729.

Tarea No.1: Rectificada

Dayli Machado de Armas (5275)

3 de junio de 2019

1. Gráfo simple no dirigido acíclico

Un grafo simple no dirigido acíclico, es aquel que no posee dirección en sus aristas, ni bucles, y por supuesto no es reflexivo [3].

Este tipo de grafos suele representarse por una secuencia de nodos unidos por aristas, donde de cada nodo generalmente solo sale, o llega una arista [1].

En la práctica este tipo de grafos puede emplearse para representar estructuras moleculares que sigan este comportamiento, por ejemplo, la representación de moléculas. Un ejemplo de estas moléculas sería la representación de alcanos de tipo lineal, donde los átomos serían los nodos y los enlaces entre estos las aristas. Este tipo de moléculas seleccionadas no poseen ciclos, ni bucles en su representación. Ver figura 1 en la página 2 donde se muestra la representación gráfica del mismo.

```
1 import matplotlib.pyplot as plt
2 import networkx as nx
3 H=nx.Graph()
4 H.add_path([0,1,2,3,4,])
5 nx.draw (H)
6
7 plt.savefig("imagenes/Fig01.eps")
```

Practica.py

2. Gráfo simple no dirigido cíclico

Un grafo simple no dirigido cíclico es aquel que no posee dirección en sus aristas, pero si se forma un ciclo que represente una figura cerrada que comience y termine en el mismo vértice, entonces es llamado cíclico [2]. En los grafos no dirigidos, el flujo puede fluir en ambas direcciones.

Este tipo de grafos suele representarse por una secuencia de nodos unidos por aristas, pero estos pueden tener dentro un ciclo, o ser un ciclo en sí mismo. En la práctica, poseen múltiples aplicaciones, por ejemplo en la topología de redes circular, es un ejemplo de aplicación de este tipo de

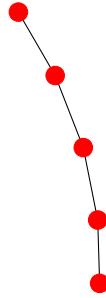


Figura 1: Representación de grafo simple no dirigido acíclico

grafo, otro ejemplo pudiera ser la representación de las autopistas de una región dada, o también la representación de otro tipo de moléculas de alcanos, por ejemplo en las de alcanos que se representan como ciclos, siendo el caso del ciclo pentano por ejemplo con cinco nodos unidos por aristas de manera circular.

El ejemplo a seguir son las llamadas que se realizan interdepartamentales en una muestra de tiempo de la jornada laboral en una empresa determinada, supongamos el departamento de producción, de marketing, comercial, calidad y planificación de la producción, donde cada nodo representa un departamento dentro de la empresa o área de esta, y las aristas las llamadas que se realizan, estas pueden tener un comportamiento que al graficarlo represente un grafo simple no dirigido cíclico. Su utilidad pudiera ser analizar el comportamiento del grafo para minimizar la cantidad de llamadas, o el tiempo de estas por ejemplo. Ver figura 2 en la página 3 donde se muestra la representación gráfica del mismo.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G = nx.Graph()
5
6 G.add_nodes_from([0,1,2,3,4])
7 pos = nx.spring_layout(G)
8
9 G.add_edge(0,1)
10 G.add_edge(0,3)
11 G.add_edge(1,4)
12 G.add_edge(1,2)
13 G.add_edge(3,1)
14 G.add_edge(0,4)
15
16 nx.draw_networkx_nodes(G, pos, node_size=2000, node_color='y', node_shape='o')
17 nx.draw_networkx_edges(G, pos, width=1, alpha=0.5, edge_color='r')
18
19
20 labels = {}
21 labels[0] = r'Markt'
22 labels[1] = r'Prod'
23 labels[2] = r'Cald'
24 labels[3] = r'Comerc'
25 labels[4] = r'Planif'
```

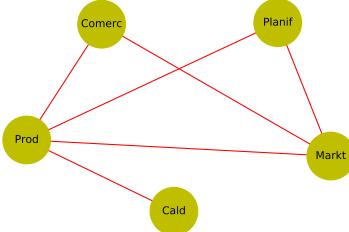


Figura 2: Representación de grafo simple no dirigido cíclico

```

26 plt.axis('off')
27 nx.draw_networkx_labels(G, pos, labels, font_size=10)
28 plt.savefig("imagenes/Fig02.eps")
29
30 plt.show()
31
32

```

grafo2.py

3. Gráfo simple no dirigido reflexivo

Un grafo simple en el cual no se permiten aristas multiples, además será no dirigido y reflexivo, si no poseen dirección en sus aristas y reflexivo si cuenta al menos con un bucle, lo que se refiere a una arista reflexiva en la que coinciden el vértice de origen y el vértice de destino [3].

Este grafo pudiera emplearse para representar el comportamiento del entrecruzamiento entre diferentes grupos raciales, donde cada grupo representa un nodo diferente, pudiendo existir cruzamiento entre ellos, y las relaciones entre los grupos son los hijos que tengan, es una relación no dirigida, a su vez dentro del grupo de la misma raza se pueden relacionar entre ellos mismos y estarían ocurriendo lazos reflexivos. Si se consideran los cuatro grupos representativos de las razas según la clasificación de especialistas serían: Blanco/caucásico, asiático/mongoloide, negroide/negro y australoide, esta clasificación posee dentro cada grupo una clasificación de hasta 30 subgrupos, pero solo se representarán los cuatro principales. Ver figura 3 en la página 4 donde se muestra la representación gráfica del mismo.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G = nx.Graph()
5 G.add_nodes_from([1, 2, 3, 4])
6
7 node_b = {1}
8 node_a = {2}
9 node_n = {3}

```

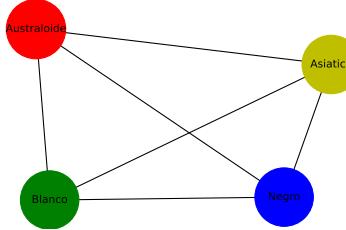


Figura 3: Representación de grafo simple no dirigido reflexivo

```

10| node_au = {4}
11|
12| pos = nx.spring_layout(G)
13| #pos = {1:(5000,8020), 2:(7150,6620), 3:(6675, 5180), 4:(5475,8200)}
14|
15| G.add_edges_from([(1,1), (1,2), (1,3), (1,4)])# Cada nodo es reflexivo pero no me
16|     sale la flechita
17| G.add_edges_from([(2,2), (2,1), (2,3), (2,4)])#igualmente no sale el reflexivo en si
18|     mismo
19| G.add_edges_from([(3,3), (3,1), (3,2), (3,4)])#no sale el reflexivo en si mismo
20| G.add_edges_from([(4,4), (4,1), (4,3), (4,2)])#es reflexivo en si mismo
21|
22| nx.draw_networkx_nodes(G, pos, nodelist=node_b, node_size=3000, node_color='g',
23|     node_shape='o')
24| nx.draw_networkx_nodes(G, pos, nodelist=node_a, node_size=3000, node_color='y',
25|     node_shape='o')
26| nx.draw_networkx_nodes(G, pos, nodelist=node_n, node_size=3000, node_color='b',
27|     node_shape='o')
28| nx.draw_networkx_nodes(G, pos, nodelist=node_au, node_size=3100, node_color='r',
29|     node_shape='o')
30|
31| nx.draw_networkx_edges(G, pos, width=1, alpha=0.8, edge_color='black', )
32|
33| plt.axis('off')
34|
35| nx.draw_networkx_labels(G, pos, labels, font_size=10)
36|
37| #plt.xlim (2000,16000)
38| #plt.autoscale (enable=true, axis='both')
39| plt.savefig("imagenes/Fig03.eps", bbox_inches='tight')
40| #nx.draw(G)
41| plt.show()

```

grafo3.py

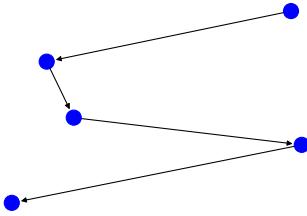


Figura 4: Representación de gráfo simple dirigido acíclico

4. Gráfo simple dirigido acíclico

Un grafo simple dirigido acíclico, es aquel que posee una dirección en sus aristas, y no posee bucles o reflexividad, ni ciclos[3].

En este caso, ejemplos de la vida real son aquellos que posean un origen del que puede salir una o varias aristas por diferentes caminos, sin que entre ellas existan ciclos y tengan un destino final diferente al del origen. Los árboles genealógicos, los organigramas en las empresas, el flujo de procesos industriales que no posean ciclos, que posean un inicio y un fin, pudieran resultar ser ejemplos de la vida real en los que se puede aplicar este tipo de grafos. Ver figura 4 en la página 5 donde se muestra la representación gráfica del mismo.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G = nx . DiGraph ()
5
6 G.add_node(1)
7 G.add_node(2)
8 G.add_node(3)
9 G.add_node(4)
10 G.add_node(5)
11
12 G.add_edge(1 ,2)
13 G.add_edge(2 ,3)
14 G.add_edge(3 ,4)
15 G.add_edge(4 ,5)
16 pos = nx.spring_layout(G)
17 nx.draw_networkx_nodes(G, pos , node_size=200, node_color='b' , node_shape='o')
18 nx.draw_networkx_edges(G, pos , width=1, alpha=0.8, edge_color='black')
19 plt.axis('off')
20
21 plt.savefig("imagenes/Fig04 .eps")
22 plt.show()

```

grafo4.py

5. Gráfo simple dirigido cíclico

Un grafo simple dirigido cíclico es aquel que posee una dirección en sus aristas, y posee una figura cerrada que comienza y termina en el mismo vértice.

En este caso, ejemplos de la vida real son aquellos que posean un origen del que puede salir una o varias aristas por diferentes caminos, y que en un determinado nodo regresen a alguno anterior de modo que se forme uno o más ciclos. El destino final deberá ser diferente al del origen.

Como aplicación real de este tipo de grafo, puede ser el flujo de procesos industriales o artesanales que posean ciclos, por ejemplo: la elaboración artesanal de jugo de naranja, al representar este proceso en un flujograma mediante un diagrama OPERIN, en el que se reflejen sus operaciones. Una vez que se llegue al paso de exprimir las naranjas que se cae en un ciclo volviendo a la operación anterior de seleccionar otra y otra naranja hasta cubrir toda la capacidad del extractor de jugo. En este ejemplo las operaciones son los nodos y los enlaces entre los nodos la transformación que va teniendo la naranja durante el proceso. Pudiera representarse como se muestra en la figura 5 en la página 7 donde se muestra la representación gráfica del mismo.

```
1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G = nx . DiGraph ()
5
6 G.add_node(1)
7 G.add_node(2)
8 G.add_node(3)
9 G.add_node(4)
10 G.add_node(5)
11 G.add_node(6)
12
13 pos = {1:(100, 0), 2:(150,0), 3:(200, 0), 4:(250,0), 5:(300,0), 6:(350,0)}
14
15 G.add_edge(1,2)
16 G.add_edge(2,3)
17 G.add_edge(3,4)
18 G.add_edge(4,5)
19 G.add_edge(5,6)
20
21 G.add_cycle([5,4])
22 #G.add_cycle (5,4)
23
24 nx.draw_networkx_nodes(G, pos , node_size=600, node_color='r' , node_shape='o')
25 nx.draw_networkx_edges(G, pos , width=1, alpha=0.8, edge_color='black')
26
27 labels = {}
28 labels [1] = r '$Op1$'
29 labels [2] = r '$Op2$'
30 labels [3] = r '$Op3$'
31 labels [4] = r '$Op4$'
32 labels [5] = r '$Op5$'
33 labels [6] = r '$Op6$'
34
35 nx.draw_networkx_labels(G, pos , labels , font_size=10)
36
```

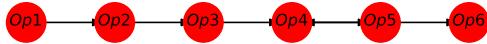


Figura 5: Representación de gráfo simple dirigido cíclico

```

37 plt.axis('off')
38 plt.savefig("imagenes/Fig05.eps")
40 plt.show()

```

grafo5.py

6. Gráfo simple dirigido reflexivo

La diferencia con este grafo y el anterior reflexivo visto, es que en este caso las aristas si tienen sentido y el flujo que se analice debe ir en una dirección. En este caso deberá existir un nodo que se llame a sí mismo, o sea, que posea al menos un bucle.

Un ejemplo en el que pudiera emplearse la teoría de grafos, y que pudiera seguir un comportamiento dirigido reflexivo es al analizar entre personas de grupos sanguíneos diferentes, representar de quien pueden recibir sangre en función del tipo que posean. La reflexividad está dada en los casos en que cada grupo sanguíneo puede recibir de otros grupos y de sí mismo, excepto el grupo O- que solo puede recibir del mismo tipo.

Pudiera representarse como se muestra en la figura 6 en la página 9 donde se muestra la representación gráfica del mismo.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G = nx.DiGraph()
5 G.add_node(1)

```

```

7 G.add_node(2)
8 G.add_node(3)
9 G.add_node(4)
10 G.add_node(5)
11 G.add_node(6)
12 G.add_node(7)
13 G.add_node(8)
14
15 node_a = {1,2}
16 node_b = {3,4}
17 node_ab = {5,6}
18 node_o = {7,8}
19
20 pos = nx.spring_layout(G)
21
22 G.add_edges_from([(1,1), (8,1), (7,1), (2,1)])# Cada nodo es reflexivo pero no me
   sale la flechita
23 G.add_edges_from([(2,2), (8,2)])#igualmente no sale el reflexivo en si mismo
24 G.add_edges_from([(3,3), (8,3), (7,3), (4,3)])#no sale el reflexivo en si mismo
25 G.add_edges_from([(4,4), (8,4)])#es reflexivo en si mismo
26 G.add_edges_from([(5,5), (1,5), (2,5), (3,5), (4,5), (6,5), (7,5), (8,5)])# es
   reflexivo en si mismo
27 G.add_edges_from([(6,6), (4,6), (8,6), (2,6)])
28 G.add_edges_from([(7,7), (8,7)])
29 G.add_edges_from([(8,8)])
30
31 nx.draw_networkx_nodes(G, pos, nodelist=node_a, node_size=800, node_color='g',
   node_shape='o', alpha=0.3)
32 nx.draw_networkx_nodes(G, pos, nodelist=node_b, node_size=800, node_color='y',
   node_shape='o', alpha=0.3)
33 nx.draw_networkx_nodes(G, pos, nodelist=node_ab, node_size=800, node_color='b',
   node_shape='o', alpha=0.3)
34 nx.draw_networkx_nodes(G, pos, nodelist=node_o, node_size=800, node_color='r',
   node_shape='o', alpha=0.8)
35 #nx.draw_networkx_nodes(G, pos, nodelist=node_o, node_size=800, node_color='r',
   node_shape='on', alpha=0.1)
36 nx.draw_networkx_edges(G, pos, width=1, alpha=0.8, edge_color='black', )
37
38 labels = {}
39 labels[1] = r'A+'
40 labels[2] = r'A-'
41 labels[3] = r'B+'
42 labels[4] = r'B-'
43 labels[5] = r'AB+'
44 labels[6] = r'AB-'
45 labels[7] = r'O+'
46 labels[8] = r'O-'
47
48 plt.axis('off')
49 nx.draw_networkx_labels(G, pos, labels, font_size=10)
50 plt.savefig("imagenes/Fig06.eps")
51 plt.show()

```

grafo6.py

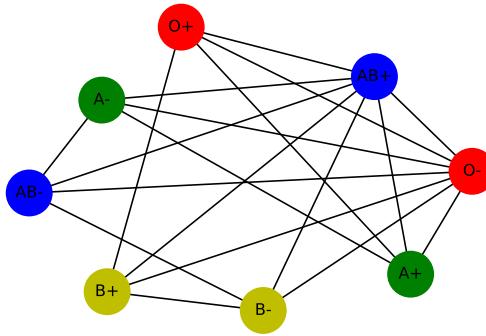


Figura 6: Representación de gráfo simple dirigido reflexivo

7. Multigrafo no dirigido acíclico

Cuando haya más de una arista entre un par de vértices, el grafo se llama multígrafo. Si no se permiten aristas múltiples, el grafo es simple[3]. Es no dirigido cuando las aristas no poseen sentido restringido, sino que el flujo fluye en ambas direcciones, y en este caso no presenta ciclos.

Un ejemplo práctico de multígrafo no dirigido acíclico pudiera decirse que es la representación de determinados enlaces moleculares, como ejemplo pudiera citarse el tipo de elemento que posea un enlace doble éster. Para la representación de los elementos con este tipo de enlace puede emplearse el grafo en cuestión. Específicamente pudiera decirse que la representación de éster sulfúrico pudiera representarse con este comportamiento, como se muestra en la figura 7 en la página 11 donde se muestra la representación gráfica del mismo.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G=nx.MultiGraph()
5
6 G.add_node(1)
7 G.add_node(2)
8 G.add_node(3)
9 G.add_node(4)
10 G.add_node(5)
11 G.add_node(6)
12 G.add_node(7)
13
14 node_o = {4,5,6,7}
15 node_r = {1,2,3}
16
17 G.add_edge(4,3, weight=3)
18 G.add_edge(4,3, weight=5)
19 G.add_edge(3,7, weight=3)

```

```

20 G.add_edge(3,7, weight=5)
21 G.add_edge(1,5)
22 G.add_edge(5,3)
23 G.add_edge(3,6)
24 G.add_edge(6,2)
25
26 blue=[(4,3),(3,7),(1,5),(5,3),(3,6),(6,2)]
27 red=[(4,3),(3,7)]
28 pos = {1:(0,220), 2:(150,220), 3:(75, 180), 4:(75,200), 5:(25,180),6:(125,180),
    7:(75,160)}
29 nx.draw_networkx_nodes(G, pos, nodelist=node_o, node_size=600, node_color='b',
    node_shape='o')
30 nx.draw_networkx_nodes(G, pos, nodelist=node_r, node_size=600, node_color='y',
    node_shape='o')
31 nx.draw_networkx_edges(G, pos, edgelist=blue, width=6, alpha=0.5,
    edge_color='b', style='dashed')
32 nx.draw_networkx_edges(G, pos, edgelist=red, width=6, alpha=0.5,
    edge_color='r')
33
34
35 labels = {}
36 labels[1] = r'R1'
37 labels[2] = r'R2'
38 labels[3] = r'S'
39 labels[4] = r'O'
40 labels[5] = r'O'
41 labels[6] = r'O',
42 labels[7] = r'O'
43
44 plt.axis('off')
45 nx.draw_networkx_labels(G, pos, labels, font_size=10)
46 pos = nx.spring_layout(G)
47 plt.savefig("imagenes/Fig07.eps")
48 plt.show()

```

grafo7.py

8. Multigrafo no dirigido cíclico

Cuando haya más de una arista entre un par de vértices, el grafo se llama multígrafo. En este caso será no dirigido cuando las aristas no poseen sentido restringido, sino que el flujo fluye en ambas direcciones, y deberán existir zonas cerradas, o sea, que regresen al nodo inicial.

Este tipo de grafo es de gran utilidad para representar el comportamiento de redes sociales y neuronales por ejemplo. El ejemplo práctico que se me ocurre mostrar, es cuando tenemos necesidad de comunicarnos con alguien y las vías de comunicación que tenemos en la actualidad, donde cada vía de comunicación seleccionada tendrá un costo (peso), y existen muchas vías para comunicarnos con otras personas, y a su vez para que esa persona se comunique con nosotros, por lo que no es dirigido, pero sí cíclico, debido a los lazos que se pueden formar entre las personas que necesitan comunicarse. El ejemplo se restringe a solo dos vías de comunicación, ya sea por whatsapp, o por llamadas telefónicas, aunque en la práctica existen muchas más. Este grafo se muestra en la figura 8 en la página 12 donde se muestra la representación gráfica del mismo.

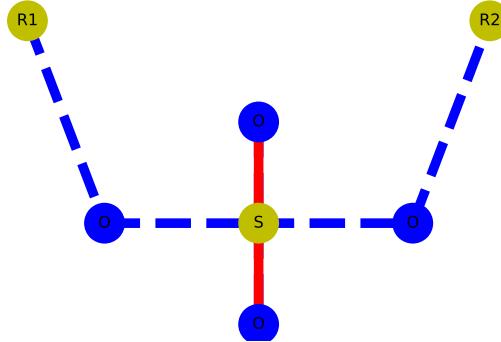


Figura 7: Representación de multigrafo no dirigido acíclico

```

1
2
3 import matplotlib.pyplot as plt
4 import networkx as nx
5
6 G = nx.MultiGraph ()
7
8 G.add_node(1)
9 G.add_node(2)
10 G.add_node(3)
11 G.add_node(4)
12 G.add_node(5)
13 E=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]
14 for e in E:
15     (u,v)=e
16     G.add_edge(u,v,3)
17     G.add_edge(u,v,4)
18 green=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]
19 yellow=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]
20 pos = nx.spring_layout(G)
21 plt.axis('off')
22 nx.draw_networkx_nodes(G, pos ,node_size=400, node_color='black', node_shape='o')
23 nx.draw_networkx_edges(G, pos , edgelist=green , width=3, alpha=0.5,
24 edge_color='g', style='dashed')
25 nx.draw_networkx_edges(G, pos , edgelist=yellow , width=4, alpha=0.5,
26 edge_color='y')
27 plt.savefig("imagenes/Fig08.eps")
28 plt.show()

```

grafo8.py

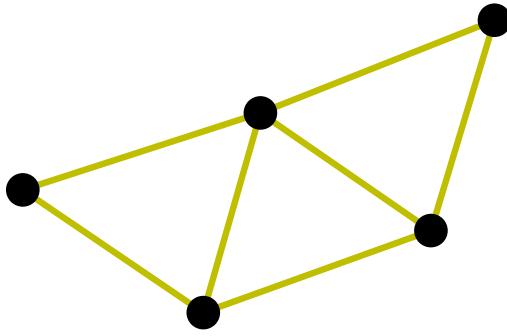


Figura 8: Representación de Multigrafo no dirigido cíclico

9. Multigrafo no dirigido reflexivo

Este caso se refiere a un multigrafo en el que existe más de una arista entre un par de vértices, sin dirección entre estas, pero de un nodo deben salir más de una arista que regresen al mismo nodo, sin pasar por otro.

Una aplicación pudiera ser la relación que existe entre cinco signos zodiacales, donde cada nodo representa a las personas de un signo determinado y de cada uno de ellos puede salir una arista que represente las relaciones positivas entre los diferentes signos y otra las relaciones negativas, y a su vez las personas del mismo signo se relacionarán con sus iguales de manera positiva o negativa, lo que representa la reflexibilidad. Este grafo se muestra, para una muestra de cinco signos cualesquiera, en la figura 9 en la página 13 donde se muestra la representación gráfica del mismo.

```

1 import matplotlib.pyplot as plot
2 import networkx as nx
3
4 G = nx . MultiGraph ()
5
6 G.add_node(1)
7 G.add_node(2)
8 G.add_node(3)
9 G.add_node(4)
10 G.add_node(5)
11
12 E=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]
13 for e in E:
14     (u,v)=e
15     G.add_edge(u,v,3)
16     G.add_edge(u,v,4)
17 node_A = {1}
18 node_S = {2}
```

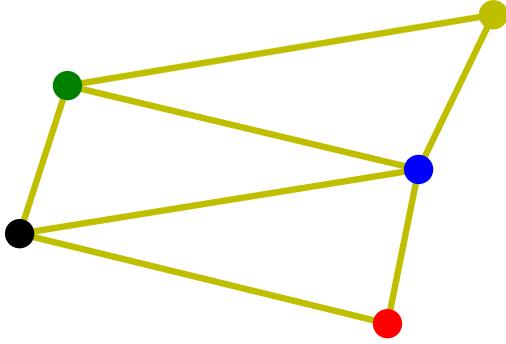


Figura 9: Representación de Multigrafo no dirigido reflexivo

```

20| node_E = {3}
21| node_V = {4}
22| node_C = {5}
23| green=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]
24| yellow=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]
25| pos = nx.spring_layout(G)
26| plt.axis('off')
27| nx.draw_networkx_nodes(G, pos, nodelist=node_A, node_size=300, node_color='g',
28|                         node_shape='o')
29| nx.draw_networkx_nodes(G, pos, nodelist=node_S, node_size=300, node_color='y',
30|                         node_shape='o')
31| nx.draw_networkx_nodes(G, pos, nodelist=node_E, node_size=300, node_color='b',
32|                         node_shape='o')
33| nx.draw_networkx_nodes(G, pos, nodelist=node_V, node_size=300, node_color='r',
34|                         node_shape='o')
35| nx.draw_networkx_nodes(G, pos, nodelist=node_C, node_size=300, node_color='black',
36|                         node_shape='o')
37|
38| plt.savefig("imagenes/Fig09.eps")
39| plt.show()

```

grafo9.py

10. Multigrafo dirigido acíclico

Para este tipo de grafo deben existir más de una arista entre un par de nodos, con dirección y no debe formarse ninguna figura cerrada dentro del grafo.

Este ejemplo aplicado a la práctica pudiera ser la representación de un viajero que desea ir de una ciudad a otra para visitarlas y tiene que elegir entre varias rutas posibles para llegar, o pudiera elegir también entre diferentes medios de transporte para trasladarse entre las ciudades, o una combinación de ambos. En este caso las ciudades serían los nodos, y las rutas o medios de transporte posibles a elegir entre un nodo u otro serían las aristas. Una ruta pudiera ser ir de la ciudad de Matanzas en Cuba, a la Habana, de esta a Monterrey, de Monterrey a Torreón y de ahí a Sinaloa. Este grafo se muestra en la figura 10 en la página 15 donde se muestra la representación gráfica del mismo.

```
1
2
3 import matplotlib.pyplot as plt
4 import networkx as nx
5
6 G = nx.MultiDiGraph()
7
8 G.add_node(1)
9 G.add_node(2)
10 G.add_node(3)
11 G.add_node(4)
12 G.add_node(5)
13
14 G.add_edge(1,2, weight=3)
15 G.add_edge(1,2, weight=4)
16 G.add_edge(2,3, weight=3)
17 G.add_edge(2,3, weight=4)
18 G.add_edge(3,4, weight=3)
19 G.add_edge(3,4, weight=4)
20 G.add_edge(4,5, weight=3)
21 G.add_edge(4,5, weight=4)
22
23 medio=[(1,2),(2,3),(3,4),(4,5)]
24 ruta=[(1,2),(2,3),(3,4),(4,5)]
25
26 pos = nx.spring_layout(G)
27
28 nx.draw_networkx_nodes(G, pos, node_size=500, node_color='b', node_shape='o')
29
30 nx.draw_networkx_edges(G, pos, edgelist=medio, width=6, alpha=0.8,
31 edge_color='black', style='dashed')
32
33 nx.draw_networkx_edges(G, pos, edgelist=ruta, width=4, alpha=0.5,
34 edge_color='r')
35
36 labels = {}
37 labels[1] = r'Mat'
38 labels[2] = r'Hab'
39 labels[3] = r'Monty'
40 labels[4] = r'Torr'
41 labels[5] = r'Sing'
```

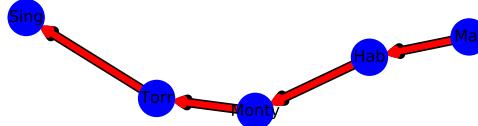


Figura 10: Representación de multigrafo dirigido acíclico

```

43 plt.axis('off')
44 nx.draw_networkx_labels(G, pos, labels, font_size=10)
45
46 plt.savefig("imagenes/Fig10.eps")
47 plt.show()

```

grafo10.py

11. Multigrafo dirigido cíclico

Para ser multigrafo dirigido cíclico debe existir más de una arista entre un par de nodos, con dirección y además formarse al menos una figura cerrada dentro del grafo.

Un ejemplo que representa este tipo de grafo sería la representación del flujo que siguen los pacientes una vez que asisten a la consulta de cuerpo de guardia del ISSSTE, en la que las operaciones del proceso serían los nodos y la clasificación del tipo de paciente y su recorrido dentro de la instalación las aristas. En este caso el cliente al llegar pasa por la consulta de clasificación y ahí le asignan un color en función de la gravedad de su dolencia, se asumen tres colores, rojo, amarillo y verde, que van de mayor gravedad a menor respectivamente y de esta dependerá el tiempo de espera para pasar a la siguiente consulta donde se encuentra el doctor, existe tres consultas con doctores disponibles y de ahí pueden pasar al laboratorio, u a otras de las salas. Luego, del laboratorio pueden pasar nuevamente a la consulta del doctor para revisar los resultados y ocurriría un ciclo, o del doctor directo a la sala, luego al laboratorio y retornar nuevamente al doctor, y ocurriría otro ciclo. Este grafo se muestra en la figura 11 en la página 17.

```

1 import matplotlib.pyplot as plt
2

```

```

3 import networkx as nx
4 G = nx . MultiDiGraph ()
5
6 G.add_node(1)
7 G.add_node(2)
8 G.add_node(3)
9 G.add_node(4)
10 G.add_node(5)
11
12 G.add_edge(1 ,2 , weight=3)
13 G.add_edge(1 ,2 , weight=4)
14 G.add_edge(1 ,2 , weight=5)
15 G.add_edge(2 ,3 , weight=3)
16 G.add_edge(2 ,3 , weight=4)
17 G.add_edge(2 ,4 , weight=3)
18
19 G.add_edge(2 ,5 , weight=3)
20 G.add_edge(3 ,2 , weight=4)
21 G.add_edge(3 ,2 , weight=3)
22
23
24 y=[(1 ,2 ),(2 ,3 ),(3 ,2 )]
25 g=[(1 ,2 ),(2 ,3 ),(3 ,2 )]
26 r=[(1 ,2 ),(2 ,4 ),(2 ,5 )]
27
28 pos = nx.spring_layout(G)
29
30 nx.draw_networkx_nodes(G, pos , node_size=400, node_color='b' , node_shape='o')
31
32 nx.draw_networkx_edges(G, pos , edgelist=y, width=10, alpha=0.2,
33 edge_color='y' , style='dashed')
34
35 nx.draw_networkx_edges(G, pos , edgelist=g, width=6, alpha=0.5,
36 edge_color='g')
37
38 nx.draw_networkx_edges(G, pos , edgelist=r, width=2, alpha=0.8,
39 edge_color='r')
40
41 labels = {}
42 labels [1] = r '1'
43 labels [2] = r '2'
44 labels [3] = r 'Lab'
45 labels [4] = r '3'
46 labels [5] = r '4'
47
48
49 plt.axis( 'off')
50
51 nx.draw_networkx_labels(G, pos , labels , font_size=10)
52
53 plt.savefig("imagenes/Fig11.eps")
54 plt.show()
55

```

grafo11.py

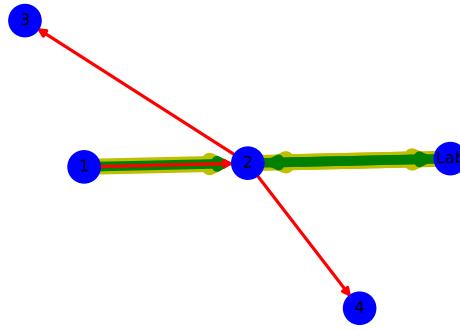


Figura 11: Representación de multigrafo dirigido cíclico

12. Multigrafo dirigido reflexivo

Para ser un multigrafo dirigido reflexivo debe existir más de una arista entre un par de nodos, con dirección y además cada nodo debe llamarse a sí mismo.

Un ejemplo en el que el empleo de la teoría de grafos como los que se analizan en esta sección puede emplearse es en el estudio de la influencia de la religión en la sociedad, como un subejemplo de la aplicación en redes sociales [2]. Por ejemplo, si cada nodo es una persona religiosa creyente, ese nodo (persona) constantemente se realiza un autoexamen de conciencia a sí mismo, evaluando cómo ha sido su actitud con respecto a las variables a analizar en este autoexamen. Estas variables pueden ser operacionalizadas como cada una de las doctrinas de cada religión, las cuales varían de una religión a otra, pero son más de una y pueden agruparse por categorías. Estas variables representan las aristas, cada una puede tener un peso determinado según sea el interés del estudio, a su vez estas mismas doctrinas (convertidas en variable) serán las que cada persona creyente se encargará de transmitir a otras personas creyentes o no, pudiera decirse que se transmite el tipo de actitud que se debe tener ante cada una de estas variables de una persona a otra.

En este ejemplo se puede evaluar el impacto en un grupo poblacional que puede tener un tipo de creencia u otro, en la medida en que aumente la red. Una simplificación de este tipo de grafo se representa en la figura 12 en la página 19 donde se muestra la representación gráfica del mismo.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G = nx . MultiDiGraph ()
5
6 G.add_node(1)
7 G.add_node(2)
8 G.add_node(3)
9 G.add_node(4)
10

```

```

11 G.add_node(5)
12
13 a= {1}
14 b= {2}
15 c= {3}
16 d= {4}
17 e= {5}
18
19 G.add_edge(1,2, weight=3)
20 G.add_edge(1,2, weight=4)
21 G.add_edge(1,2, weight=5)
22 G.add_edge(2,3, weight=3)
23 G.add_edge(2,3, weight=4)
24 G.add_edge(2,3, weight=4)
25 G.add_edge(3,4, weight=3)
26 G.add_edge(3,4, weight=3)
27 G.add_edge(3,4, weight=3)
28 G.add_edge(4,5, weight=3)
29 G.add_edge(4,5, weight=3)
30 G.add_edge(4,5, weight=3)
31
32
33 y=[(1,2),(2,3),(3,4),(4,5)]
34 g=[(1,2),(2,3),(3,4),(4,5)]
35 r=[(1,2),(2,3),(3,4),(4,5)]
36
37 pos = nx.spring_layout(G)
38
39 nx.draw_networkx_nodes(G, pos, nodelist=a, node_size=300, node_color='black',
40     node_shape='o')
41 nx.draw_networkx_nodes(G, pos, nodelist=b, node_size=300, node_color='y', node_shape
42     ='o')
43 nx.draw_networkx_nodes(G, pos, nodelist=c, node_size=300, node_color='b', node_shape
44     ='o')
45 nx.draw_networkx_nodes(G, pos, nodelist=d, node_size=300, node_color='r', node_shape
46     ='o')
47 nx.draw_networkx_nodes(G, pos, nodelist=e, node_size=300, node_color='g', node_shape
48     ='o')
49
50 nx.draw_networkx_edges(G, pos, edgelist=y, width=8, alpha=1,
51     edge_color='y', style='dashed')
52
53 nx.draw_networkx_edges(G, pos, edgelist=g, width=6, alpha=0.5,
54     edge_color='g')
55
56 nx.draw_networkx_edges(G, pos, edgelist=r, width=2, alpha=0.8,
57     edge_color='r')
58
59 plt.axis('off')
60
61 plt.savefig("imagenes/Fig12.eps")
62 plt.show()

```

grafo12.py

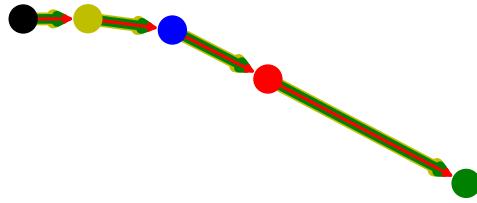


Figura 12: Representación de multigrafo dirigido reflexivo

Referencias

- [1] Pieter Swart Aric Hagberg, Dan Schult. *NetworkX Reference, Release 2.3rc1.dev20190113142952*, 2019.
- [2] Anwesha Chakraborty, Trina Dutta, Sushmita Mondal, and Asoke Nath. Application of graph theory in social media. *International Journal of Computer Sciences and Engineering*, 6, 10 2018. doi: 10.26438/ijcse/v6i10.722729.
- [3] Elisa Schaeffer. *Complejidad computacional de problemas y el análisis y diseño de algoritmos*. 2017.



Tarea No.2: Flujo en Redes

Dayli Machado (5275)

26 de febrero de 2019

1. Grafo simple no dirigido acíclico usando *bipartite layout*

Un grafo simple no dirigido acíclico, es aquel que no posee dirección en sus aristas, ni bucles, y por supuesto no es reflexivo [5]. Este tipo de grafos suele representarse por una secuencia de nodos unidos por aristas, donde de cada nodo generalmente solo sale o llega una arista [1].

Este tipo de grafos puede emplearse para representar estructuras moleculares que sigan este comportamiento, un ejemplo sería la representación de alcanos de tipo lineal, donde los átomos se representan por los nodos y los enlaces entre estos por las aristas. El tipo de algoritmo de acomodamiento *layout* que se empleó para lograr esta representación fue el *bipartite layout* el cual permite posicionar los nodos en dos líneas rectas y trazar ejes entre ellos [4], a partir de tener dos conjuntos de nodos diferentes se puede graficar la relación entre ellos. Se observa en la línea de código número 28 la forma de representar esta función empleada para la representación del grafo. Ver figura 1 en la página 2.

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Sun Feb 24 17:34:52 2019
4 @author: Dayli
5 """
6 import matplotlib.pyplot as plt
7 import networkx as nx
8
9 G = nx.Graph()
10
11 G.add_node(2)
12 G.add_node(4)
13
14 G.add_node(1)
15 G.add_node(3)
16 G.add_node(5)
17
18 G.add_edges_from([(1,2), (3,4)])
19 G.add_edges_from([(2,3), (4,5)])
20
21 labels = {}
22 labels[1] = r'H3C'
23 labels[2] = r'H2C'
24 labels[3] = r'CH2'
25 labels[4] = r'H2C'
26 labels[5] = r'CH3'
27
28 pos = nx.bipartite_layout(G, {1,3,5}, align='horizontal', scale=0.05)
29
30 nx.draw_networkx_nodes(G, pos, node_size=800, node_color='b')
31 nx.draw_networkx_edges(G, pos, width=3)
32 nx.draw_networkx_labels(G, pos, labels, font_size=10)
33 plt.axis('off')
34 plt.savefig("imagenes1/Fig01.eps")
35
36 plt.show()

```

grafo1laybi.py

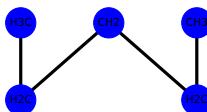


Figura 1: Molécula de alcano lineal usando *bipartite layout*

2. Grafo simple no dirigido cíclico usando *circular layout*

Un grafo simple no dirigido cíclico es aquel que no posee dirección en sus aristas, pero si se forma un ciclo que represente una figura cerrada que comience y termine en el mismo vértice, entonces es llamado cíclico [2]. En los grafos no dirigidos el flujo puede fluir en ambas direcciones.

Este tipo de grafos suele representarse por una secuencia de nodos unidos por aristas, pero estos pueden tener dentro un ciclo, o ser un ciclo en sí mismo. En la práctica pudiera ser la representación de las autopistas de una región dada, o también la representación de otro tipo de moléculas de alcanos que posean áreas cerradas en su representación, o las llamadas que se realizan interdepartamentales en una empresa.

Se toma de ejemplo las llamadas que se realizan entre los departamentos de producción, de marketing, comercial, calidad y planificación de la producción en una empresa, donde cada nodo representa un departamento y las aristas las llamadas que se realizan entre ellos. Para este tipo de grafo se empleó el *layout circular* como algoritmo de acomodamiento, pues permite ubicar los nodos en círculos [4] y de esta forma se visualiza mejor la distribución espacial de los nodos y la relación entre ellos que son las llamadas, pero como en este ejemplo la posición de los nodos puede ser aleatoria, se puede emplear también el *random layout* como otro algoritmo de acomodamiento pues este permite ubicar los nodos uniformemente al azar en el cuadro [4]. En la figura 2 en la página 5 se muestra la representación gráfica del mismo.

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Sun Feb 24 19:25:28 2019
4
5 @author: Dayli
6 """
7
8 import matplotlib.pyplot as plt
9 import networkx as nx
10
11 G = nx.Graph()
12
13 G.add_nodes_from([0,1,2,3,4])
14
15 G.add_edge(0,1)
16 G.add_edge(0,3)
17 G.add_edge(1,4)
18 G.add_edge(1,2)
19 G.add_edge(3,1)
20 G.add_edge(0,4)
21 G.add_edge(2,4)
22
23 labels = []
24 labels[0] = r'Markt'
25 labels[1] = r'Prod'
26 labels[2] = r'Cald'
27 labels[3] = r'Comerc'
28 labels[4] = r'Planif'
29
30 pos = nx.circular_layout(G, scale=0.5)
#pos = nx.spring_layout(G, scale=0.5)
31
32 nx.draw_networkx_nodes(G, pos, node_size=3000, node_color='#756b6b', node_shape='o')
33 nx.draw_networkx_edges(G, pos, width=2, alpha=0.5, edge_color='black')
34 nx.draw_networkx_labels(G, pos, labels, font_size=10, font_color='white')
35
36
37
38
39 plt.axis('off')
40
41 plt.savefig("imagenes1/Fig02.eps")
42
43 plt.show()

```

grafo2laycir.py

3. Grafo simple no dirigido reflexivo usando *circular layout*

Un grafo simple no dirigido reflexivo es aquel en el cual no se permiten aristas múltiples, no posee dirección en sus aristas y cuenta al menos con un bucle, lo que se refiere a una arista reflexiva en la que coinciden el vértice de origen y el vértice de destino [5].

Este grafo puede emplearse para representar el entrecruzamiento de grupos raciales, donde cada

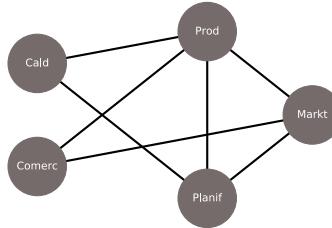


Figura 2: Llamadas interdepartamentales usando *layout circular*

grupo representa un nodo diferente y las relaciones entre los grupos serán los hijos que tengan, siendo una relación no dirigida. Al relacionarse entre la misma raza estarían ocurriendo lazos reflexivos. Si se consideran los cuatro grupos representativos de las razas humanas según la clasificación de especialistas serían: Blanco/caucásico, asiático/mongoloide, negroide/negro y australoide; esta clasificación posee dentro de cada grupo una clasificación de hasta 30 subgrupos, pero solo se representarán los cuatro principales. El *layout* que mejor refleja este tipo de grafo es el circular, la justificación es la misma explicada en la sección anterior. El código para representar este grafo fue tomado y adaptado del sitio [3], el mismo permitió insertar las imágenes como nodos. Ver figura 3 en la página 7.

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Sun Feb 24 23:03:42 2019
4
5 @author: Dayli
6 """
7 import networkx as nx
8 import matplotlib.pyplot as plt
9 import matplotlib.image as mpimg
10
11 # image from http://matplotlib.sourceforge.net/users/image_tutorial.html
12 path = "C:/Users/lapi7/Desktop/Para tarea 2/razas/"
13 img1=mpimg.imread("1.jpg")
14 img2=mpimg.imread("2.jpg")
15 img3=mpimg.imread("3.jpg")
16 img4=mpimg.imread("4.jpg")
17
18 G=nx.Graph()
19
20 G.add_nodes_from([1,2,3,4])
21
22 G.add_edges_from([(1,1), (1,2), (1,3), (1,4)])# Cada nodo es reflexivo
23 G.add_edges_from([(2,2), (2,1), (2,3), (2,4)])#
24 G.add_edges_from([(3,3), (3,1), (3,2), (3,4)])#
25 G.add_edges_from([(4,4), (4,1), (4,3), (4,2)])#
26
27 G.node[1]["image"]=img1
28 G.node[2]["image"]=img2
29 G.node[3]["image"]=img3
30 G.node[4]["image"]=img4
31
32 pos=nx.circular_layout(G, scale=0.5)
33
34 fig=plt.figure(figsize=(13,13))
35 ax=plt.subplot(1,1,1)
36 ax.set_aspect("equal")
37 nx.draw_networkx_edges(G, pos, ax=ax, width=2)
38
39 plt.xlim(-0.5,1.5)
40 plt.ylim(-0.5,1.5)
41
42 plt.axis("off")
43
44 trans=ax.transData.transform
45 trans2=fig.transFigure.inverted().transform
46
47 piesize=0.2 # this is the image size
48 p2=piesize/2.0
49 for n in G:
50     xx,yy=trans(pos[n]) # figure coordinates
51     xa,ya=trans2((xx,yy)) # axes coordinates
52     a = plt.axes([xa-p2,ya-p2, piesize , piesize])
53     a.set_aspect("equal")
54     a.imshow(G.node[n]["image"])
55     a.axis("off")
56
57 plt.savefig("imagenes1/Fig03.eps", bbox_inches='tight')

```

```
58 #nx.draw(G)
59 plt.show()
```

grafo3laycir.py

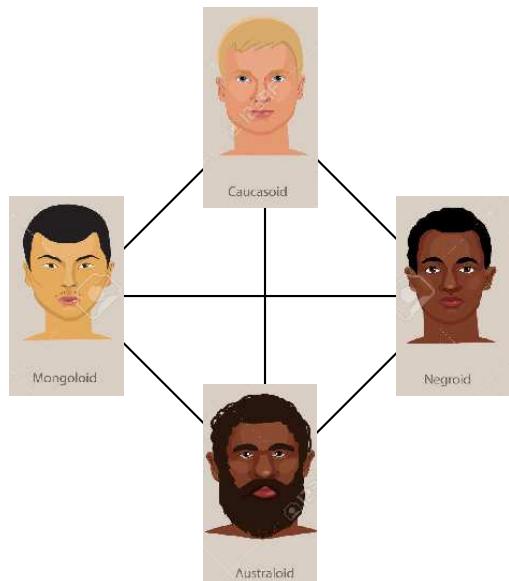


Figura 3: Grupos raciales usando *circular layout*

4. Gráfo simple dirigido acíclico usando *spring layout*

Un grafo simple dirigido acíclico, es aquel que posee una dirección en sus aristas, y no posee bucles o reflexividad, ni ciclos [5].

En este caso, ejemplos de la vida real son aquellos que poseen un origen del que puede salir una o varias aristas por diferentes caminos sin que entre ellas existan ciclos y posean un destino final diferente al del origen.

Los árboles genealógicos, los organigramas en las empresas, el flujo de procesos industriales que no posean ciclos, con un inicio y un fin, pudieran resultar ser ejemplos de la vida real en los que se puede aplicar este tipo de grafos.

Como algoritmo de acomodo para representar este grafo se usó el *spring layout*, el cual permite representar de manera sencilla las aristas rectas que salen de un nodo a otro y fijar la posición haciendo uso de sus parámetros [4] siendo el mismo algoritmo empleado en la tarea anterior para este tipo de grafo, sin embargo en este ejercicio se profundizó en el uso de sus parámetros.

Para la representación inicial se fija la posición de los nodos y el centro del grafo, además se estableció una distancia estándar entre ellos, también se emplea el degradado de color para indicar la transformación que recibe un objeto dado en una línea de producción. Ver figura 4 en la página 10.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3 G = nx.DiGraph()
4
5 G.add_node(1)
6 G.add_node(2)
7 G.add_node(3)
8 G.add_node(4)
9 G.add_node(5)
10
11 nod1={1}
12 nod2={2}
13 nod3={3}
14 nod4={4}
15 nod5={5}
16
17
18 G.add_edge(1,2)
19 G.add_edge(2,3)
20 G.add_edge(3,4)
21 G.add_edge(4,5)
22
23 positions = { 5:(0, 10), 4:(0, 20), 3:(0,30), 2:(0,40), 1:(0,50) }
24 #fixeds={1,2}
25 pos = nx.spring_layout(G, k=3, fixed=None, pos=positions, center=(0,0), iterations
   =50)
26
27 nx.draw_networkx_nodes(G, pos, nodelist=nod1, node_size=400, node_color='#580cf0',
   node_shape='o')
28 nx.draw_networkx_nodes(G, pos, nodelist=nod2, node_size=400, node_color='#6332c5',
   node_shape='o')
29 nx.draw_networkx_nodes(G, pos, nodelist=nod3, node_size=400, node_color='#6240a7',
   node_shape='o')
30 nx.draw_networkx_nodes(G, pos, nodelist=nod4, node_size=400, node_color='#574184',
   node_shape='o')
31 nx.draw_networkx_nodes(G, pos, nodelist=nod5, node_size=400, node_color='#433958',
   node_shape='o')
32
33
34 nx.draw_networkx_edges(G, pos, width=2, alpha=0.8, edge_color='black')
35 plt.axis('on')
36 #plt.axis('off')
37 plt.savefig("imagenes1/Fig04.eps")
38 plt.show()
39
40 #plt.show()

```

grafo4lyspring.py

5. Gráfo simple dirigido cíclico usando *bipartite layout*

Un grafo simple dirigido cíclico es aquel que posee una dirección en sus aristas y una figura cerrada que comienza y termina en el mismo vértice.

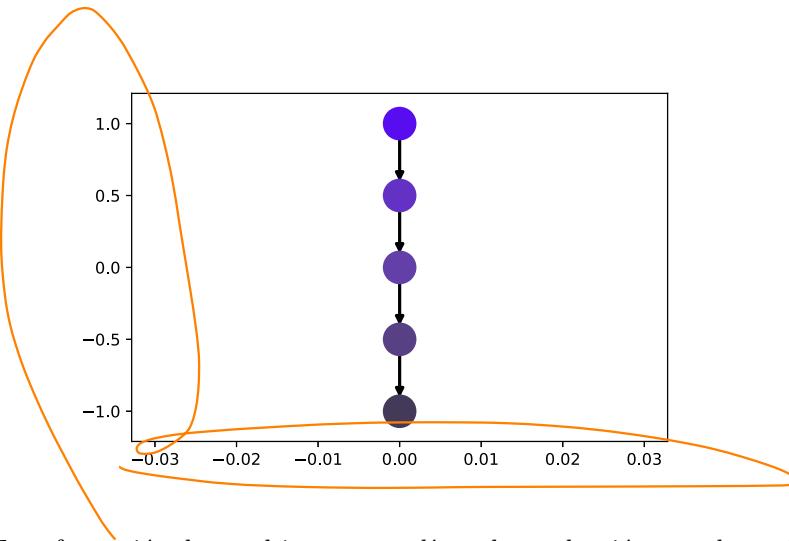


Figura 4: Transformación de un objeto en una línea de producción usando *spring layout*

Para este caso ejemplos de la vida real son aquellos que poseen un origen del que puede salir una o varias aristas por diferentes caminos, y que en un determinado nodo regresan a alguno anterior de modo que se forme uno o más ciclos. El destino final suele ser diferente al del origen.

Como aplicación real de este tipo de grafo se explica el flujo de procesos industriales o artesanales que posean ciclos, por ejemplo: la elaboración artesanal de jugo de naranja, al representar este proceso en un flujograma mediante un diagrama OPERIN, en el que se reflejen sus operaciones. Una vez que se llegue al paso de exprimir las naranjas, se cae en un ciclo volviendo a la operación anterior de seleccionar otra y otra naranja hasta cubrir toda la capacidad del extractor de jugo. En esta segunda tarea el algoritmo de acomodo empleado que se usó para representarlo fue el *bipartite layout* pues resulta más sencillo y directo para graficar los nodos que salen de la línea de producción y que forman el ciclo con los que se mantienen en ella, se emplea el parámetro *top* para separar el conjunto de nodos que no están en la línea principal de producción. Con este algoritmo se aprecia una mejor representación del grafo, aunque es válido aclarar que este ejemplo no se comporta exactamente como un grafo bipartito, pero este fue el algoritmo de acomodo que mejor lo representó. En este ejemplo las operaciones serían los nodos y los enlaces serían la transformación que va teniendo la naranja. Pudiera representarse como se muestra en la figura 5 en la página 12.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G = nx . DiGraph ()
5
6 G.add_node(1)
7 G.add_node(2)
8 G.add_node(3)
9 G.add_node(6)
10
11 G.add_node(4)
12 G.add_node(5)
13
14 G.add_edge(1 ,2)
15 G.add_edge(2 ,3)
16 G.add_edge(3 ,4)
17 G.add_edge(4 ,5)
18 G.add_edge(5 ,6)
19 G.add_edge(3 ,6)
20
21 G.add_cycle ([5 ,3])
22
23 labels = {}
24 labels [1] = r '$Op1$'
25 labels [2] = r '$Op2$'
26 labels [3] = r '$Op3$'
27 labels [4] = r '$Op4$'
28 labels [5] = r '$Op5$'
29 labels [6] = r '$Op6$'
30
31 pos = nx.bipartite_layout(G, {4,5}, align='horizontal', scale=0.7)
32
33 nx.draw_networkx_nodes(G, pos, node_size=500, node_color='r', node_shape='o')
34 nx.draw_networkx_edges(G, pos, width=2, alpha=0.8, edge_color='black')
35 nx.draw_networkx_labels(G, pos, labels, font_size=10)
36
37 plt.axis('off')
38
39 plt.savefig("imagenes1/Fig05.eps")
40
41 plt.show()

```

grafo5laybi.py

6. Gráfo simple dirigido reflexivo usando *circular layout*

La diferencia con este grafo y el anterior reflexivo visto, es que en este caso las aristas sí tienen sentido y el flujo que se analice debe ir en una dirección. En este caso deberá existir un nodo que se llame a sí mismo, o sea, que posea al menos un bucle.

Un ejemplo en el que se puede emplear la teoría de grafos de este tipo es para representar la disposición de personas con grupos sanguíneos diferentes a recibir sangre compatible o no. La

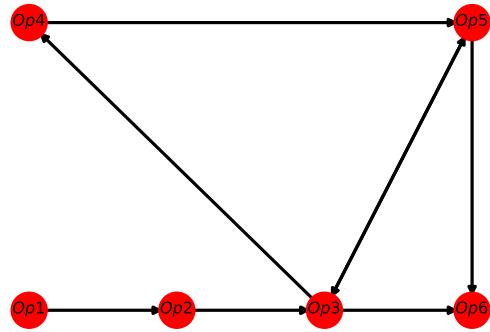


Figura 5: Flujo cíclico de una línea de producción artesanal de jugo de naranja

reflexividad está dada en los casos en que cada grupo sanguíneo puede recibir de otros grupos y de sí mismo, excepto el grupo O- que sólo puede recibir del mismo tipo. Se cambia el algoritmo de acomodo anterior por el *circular layout* haciendo uso de sus parámetros, y se evidencia más estabilidad en las iteraciones de la figura que mantienen la misma posición deseada cada vez que se itera, sin tener que entrarla manualmente. Este se representa como ~~en la figura 6 en la página 14.~~

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G = nx.DiGraph()
5 G.add_node(1) ↑
6 G.add_node(2)
7 G.add_node(3)
8 G.add_node(4)
9 G.add_node(5)
10 G.add_node(6)
11 G.add_node(7)
12 G.add_node(8)
13
14 node_a = {1,2}
15 node_b = {3,4}
16 node_ab = {5,6}
17 node_o = {7,8}
18 #Se cambiaron los layout y el mejor resultado fue con el circular
19 #pos = nx.spring_layout(G)
20 pos = nx.circular_layout(G, dim=2, scale=1, center=(0,0))
21 #pos = nx.shell_layout(G)
22 #pos = nx.rescale_layout(G)
23
24 G.add_edges_from([(1,1), (8,1), (7,1), (2,1)])# Cada nodo es reflexivo pero no me
25     sale la flechita
26 G.add_edges_from([(2,2), (8,2)])#igualmente no sale el reflexivo en si mismo
27 G.add_edges_from([(3,3), (8,3), (7,3), (4,3)])#no sale el reflexivo en si mismo
28 G.add_edges_from([(4,4), (8,4)])#es reflexivo en si mismo
29 G.add_edges_from([(5,5), (1,5), (2,5), (3,5), (4,5), (6,5), (7,5), (8,5)])# es
30     reflexivo en si mismo
31 G.add_edges_from([(6,6), (4,6), (8,6), (2,6)])
32 G.add_edges_from([(7,7), (8,7)])
33 G.add_edges_from([(8,8)])
34
35 nx.draw_networkx_nodes(G, pos, nodelist=node_a, node_size=800, node_color='g',
36     node_shape='o', alpha=0.3)
37 nx.draw_networkx_nodes(G, pos, nodelist=node_b, node_size=800, node_color='y',
38     node_shape='o', alpha=0.3)
39 nx.draw_networkx_nodes(G, pos, nodelist=node_ab, node_size=800, node_color='b',
40     node_shape='o', alpha=0.3)
41 nx.draw_networkx_nodes(G, pos, nodelist=node_o, node_size=800, node_color='r',
42     node_shape='o', alpha=0.8)
43 #nx.draw_networkx_nodes(G, pos, nodelist=node_o, node_size=800, node_color='r',
44     node_shape='on', alpha=0.1)
45 labels = []
46 labels[1] = r'A+'
47 labels[2] = r'A-'
48 labels[3] = r'B+'
49 labels[4] = r'B-'
50 labels[5] = r'AB+'
51 labels[6] = r'AB-'
52 labels[7] = r'O+'
53 labels[8] = r'O-'
54
55 nx.draw_networkx_labels(G, pos, labels, font_size=10)
56
57 #pos1 = pos
58 #

```

```

51 #for key, value in G.edges:
52 #    print("(", pos[key][0] - 0.5, ", ", pos[key][1] - 0.5, ")\\n")
53 #    pos1[key][0] = pos1[key][0] + 0.1
54 #    pos1[key][1] = pos1[key][1]
55 #
56
57 nx.draw_networkx_edges(G, pos, width=2, alpha=1, edge_color='black', scale=0.5 )
58 #print(pos)
59 plt.axis('on')
60 plt.savefig("imagenes1/Fig06.eps")
61 plt.show()

```

grafo6laycir.py

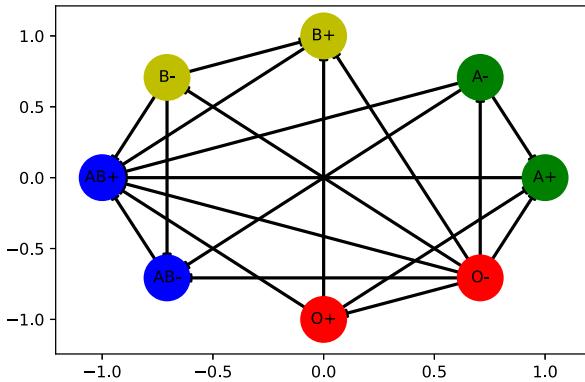


Figura 6: Compatibilidad entre grupos sanguíneos

7. Multigrafo no dirigido acíclico usando *spring layout*

Un multigrafo no dirigido acíclico es cuando existen más de una arista entre un par de vértices. Si no se permiten aristas múltiples, el grafo es simple [5]. Será no dirigido cuando las aristas no posean sentido restringido, sino que el flujo fluye en ambas direcciones, y en este caso no presenta ciclos.

Un ejemplo práctico de multígrafo no dirigido acíclico es la representación de determinados enlaces moleculares, como por ejemplo: un enlace de doble éster, como es el caso del éster sulfúrico. Para la representación de los elementos con este tipo de grafo se puede emplear el algoritmo de acomodo *spring layout*, es el mismo que se usó en el caso anterior pero esta vez se hace uso de sus parámetros sin fijar la posición [4], de este modo se obtiene una representación más ajustada a la realidad, también pudiera emplearse el ~~llamada~~ *Hawai layout* pero al compararlos e iterar el *spring layout* devuelve variantes más atractivas. El grafo usando *spring layout* se muestra en la figura 7 en la página 16.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G=nx.MultiGraph()
5
6 G.add_node(1)
7 G.add_node(2)
8 G.add_node(3)
9 G.add_node(4)
10 G.add_node(5)
11 G.add_node(6)
12 G.add_node(7)
13
14 node_o = {4,5,6,7}
15 node_r = {1,2,3}
16
17 G.add_edge(4,3, weight=3)
18 G.add_edge(4,3, weight=5)
19 G.add_edge(3,7, weight=3)
20 G.add_edge(3,7, weight=5)
21 G.add_edge(1,5)
22 G.add_edge(5,3)
23 G.add_edge(3,6)
24 G.add_edge(6,2)
25
26 blue=[(4,3),(3,7),(1,5),(5,3),(3,6),(6,2)]
27 red=[(4,3),(3,7)]
28
29 labels = {}
30 labels[1] = r'R1'
31 labels[2] = r'R2'
32 labels[3] = r'S'
33 labels[4] = r'O'
34 labels[5] = r'O'
35 labels[6] = r'O'
36 labels[7] = r'O'
37
38 pos=nx.spring_layout(G, k=0.1, iterations=300, threshold=0.0001, weight='weight',
39 scale=1)
40 nx.draw_networkx_nodes(G, pos, nodelist=node_o, node_size=600, node_color='b',
41 node_shape='o')
42 nx.draw_networkx_nodes(G, pos, nodelist=node_r, node_size=600, node_color='y',
43 node_shape='s')
44
45 nx.draw_networkx_edges(G, pos, edgelist=blue, width=6, alpha=0.5, edge_color='b',
46 style='dashed')
47 nx.draw_networkx_edges(G, pos, edgelist=red, width=6, alpha=0.5, edge_color='r')
48
49 plt.axis('on')
50 #pos = nx.spring_layout(G)
51 #pos = nx.spring_layout(G, pos, center, iteration=50)
52 #pos = kamada_kawai_layout (G)
53 #pos= nx.spectral_layout(G) Me superpone los nodos

```

```

54 plt.savefig("imagenes1/Fig07.eps")
55 plt.show()

```

grafo7lyspring.py

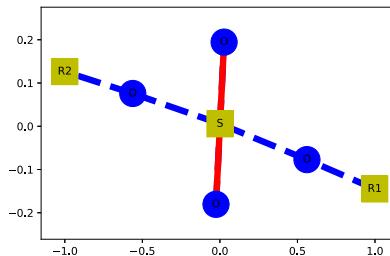


Figura 7: Mólecula de éster sulfúrico

8. Multigrafo no dirigido cíclico usando ~~Kamada Kawai layout~~

Cuando existan más de una arista entre un par de vértices, el grafo se llama multígrafo. En este caso será no dirigido cuando las aristas no poseen sentido restringido, sino que el flujo fluye en ambas direcciones, y deberán existir zonas cerradas, o sea, que regresen al nodo inicial.

Este tipo de grafo es de gran utilidad para representar el comportamiento de redes sociales y neuronales [2]. El ejemplo práctico mostrado es cuando tenemos necesidad de comunicarnos con alguien por las vías de comunicación que tenemos en la actualidad, donde cada vía de comunicación seleccionada tendrá un costo (peso), y existen muchas vías para comunicarnos con otras personas, y a su vez para otras personas se comuniquen con nosotros, por lo que no es dirigido, pero sí cíclico, debido a los lazos que se pueden formar entre las personas que necesitan comunicarse. El algoritmo de acomodo empleado para este ejemplo según sus prestaciones fue el ~~Kamada Kawai layout~~ pues este tiene en cuenta para la representación el peso de la longitud del camino [4] y devuelve grafos más atractivos para el ejemplo en cuestión. El ejemplo se restringe a solo dos vías de comunicación, ya sea por whatsapp, o por llamadas telefónicas, aunque en la práctica existen muchas más. Este grafo se muestra en la figura 8 en la página 18.

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Feb 25 20:09:39 2019
4
5 @author: lapi7
6 """
7
8 import matplotlib.pyplot as plot
9 import networkx as nx
10
11 G = nx.MultiGraph ()
12
13 G.add_node(1)
14 G.add_node(2)
15 G.add_node(3)
16 G.add_node(4)
17 G.add_node(5)
18
19 G.add_edge(1,2, weight=3)
20 G.add_edge(1,2, weight=4)
21 G.add_edge(2,3, weight=3)
22 G.add_edge(2,3, weight=4)
23 G.add_edge(3,4, weight=3)
24 G.add_edge(3,4, weight=4)
25 G.add_edge(4,5, weight=3)
26 G.add_edge(4,5, weight=4)
27 G.add_edge(5,1, weight=3)
28 G.add_edge(5,1, weight=4)
29 G.add_edge(1,3, weight=3)
30 G.add_edge(1,3, weight=4)
31 G.add_edge(5,3, weight=3)
32 G.add_edge(5,3, weight=4)
33
34 green=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]
35 yellow=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]
36
37 pos = nx.kamada_kawai_layout(G, dist=None, pos=None, weight='weight', scale=0.5,
38 center=None, dim=2)
39 plt.axis('off')
40
41 nx.draw_networkx_nodes(G, pos, node_size=400, node_color='r', node_shape='o')
42
43 nx.draw_networkx_edges(G, pos, edgelist=green, width=3, alpha=0.5,
44 edge_color='g', style='dashed')
45 nx.draw_networkx_edges(G, pos, edgelist=yellow, width=4, alpha=0.5,
46 edge_color='y')
47
48 plt.savefig("imagenes1/Fig08.eps")
49
50 plt.show()

```

grafo8lykk.py

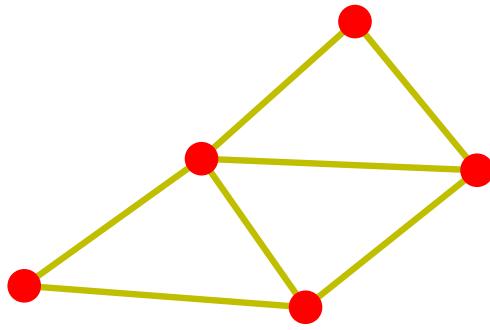


Figura 8: Vías de comunicación entre las personas

9. Multigrafo no dirigido reflexivo usando *random layout*

Este caso se refiere a un multígrafo en el que existe más de una arista entre un par de vértices, sin dirección entre estas, pero de un nodo deben salir más de una arista que regresen al mismo nodo, sin pasar por otro.

Una aplicación pudiera ser la relación que existe entre cinco signos zodiacales, donde cada nodo representa a las personas de un signo determinado y de cada uno de ellos puede salir una arista que represente las relaciones positivas entre los diferentes signos y otra las relaciones negativas, y a su vez las personas del mismo signo se relacionarán con sus iguales de manera positiva o negativa, lo que representa la reflexibilidad. Este es un grafo que no es exigente para su representación y aunque un algoritmo de acomodo circular puede ser apropiado, se propone usar uno aleatorio que funciona igual en este ejemplo y consume menos. Este grafo se muestra para cinco signos cualesquiera del zodiaco, en la Figura 9 en la página 20 donde se muestra la representación gratificante del mismo.

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Feb 11 16:09:42 2019
4
5 @author: lapi7
6 """
7
8 import matplotlib.pyplot as plot
9 import networkx as nx
10 G = nx. MultiGraph ()
11
12 G.add_node(1)
13 G.add_node(2)
14 G.add_node(3)
15 G.add_node(4)
16 G.add_node(5)
17
18 G.add_edge(1,2, weight=4)
19 G.add_edge(2,3, weight=3)
20 G.add_edge(2,3, weight=4)
21 G.add_edge(3,4, weight=3)
22 G.add_edge(3,4, weight=4)
23 G.add_edge(4,5, weight=3)
24 G.add_edge(4,5, weight=4)
25 G.add_edge(5,1, weight=3)
26 G.add_edge(5,1, weight=4)
27 G.add_edge(1,3, weight=3)
28 G.add_edge(1,3, weight=4)
29 G.add_edge(5,3, weight=3)
30 G.add_edge(5,3, weight=4)
31
32 node_A = {1}
33 node_S = {2}
34 node_E = {3}
35 node_V = {4}
36 node_C = {5}
37
38
39 green=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]
40 yellow=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]
41
42 pos = nx.random_layout(G, dim=2, center=None) # Se puede usar el random en este
43 caso
44 #pos = nx.circular_layout(G, dim=2, center=None)
45
46 plt.axis('off')
47
48 nx.draw_networkx_nodes(G, pos, nodelist=node_A, node_size=300, node_color='g',
49 node_shape='o')
50 nx.draw_networkx_nodes(G, pos, nodelist=node_S, node_size=300, node_color='y',
51 node_shape='o')
52 nx.draw_networkx_nodes(G, pos, nodelist=node_E, node_size=300, node_color='b',
53 node_shape='o')
54 nx.draw_networkx_nodes(G, pos, nodelist=node_V, node_size=300, node_color='r',
55 node_shape='o')
56 nx.draw_networkx_nodes(G, pos, nodelist=node_C, node_size=300, node_color='black',

```

```

53     node_shape='o')
54
55 nx.draw_networkx_edges(G, pos, edgelist=green, width=3, alpha=0.5,
56 edge_color='g', style='dashed')
57 nx.draw_networkx_edges(G, pos, edgelist=yellow, width=4, alpha=0.5,
58 edge_color='y')
59 plt.savefig("imagenes1/Fig09.eps")
60
61 plt.show()

```

grafo9lyrandom.py

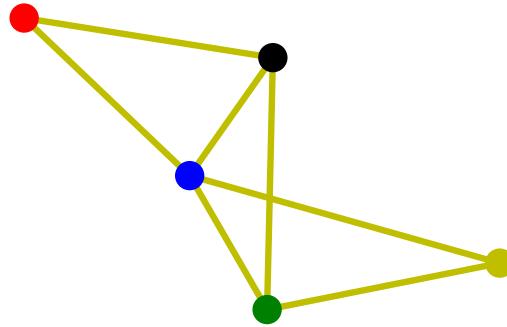


Figura 9: Relación entre personas del mismo signo zodiacal

10. Multigrafo dirigido acíclico usando *spectral layout*

Para definir un multigrafo dirigido acíclico deben existir más de una arista entre un par de nodos, con dirección y no formar ninguna figura cerrada dentro del grafo.

Este ejemplo aplicado a la práctica puede ser la representación de un viajero que desea ir de una ciudad a otra para visitarlas y tiene que elegir entre varias rutas posibles para llegar, o pudiera elegir también entre diferentes medios de transporte para trasladarse entre las ciudades, o una combinación de ambos. En este caso las ciudades serían los nodos, y las rutas o medios de transporte posibles a elegir entre un nodo u otro serían las aristas. Una ruta pudiera ser ir de la ciudad de Matanzas en Cuba, a la Habana, de esta a Monterrey, de Monterrey a Torreón y de ahí a Sinaloa. Para devolver el acomodo de estos nodos puede emplearse el algoritmo *spectral layout* que no se ha empleado en las secciones anteriores, este devuelve una forma sinusoidal pues refleja el algoritmo basado en la utilización de los vectores del gráfico laplaciano [4], lo cual no afecta a la visualización del grafo ejemplo. Asimismo se puede emplear el ~~KamadaKawai layout~~ con resultados gráficos más

direccional y apropiados para el ejemplo, pero ya se usó en secciones anteriores. El grafo se muestra en la figura 10 en la página 23.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G = nx.MultiDiGraph()
5
6 G.add_node(1)
7 G.add_node(2)
8 G.add_node(3)
9 G.add_node(4)
10 G.add_node(5)
11
12 G.add_edge(1,2, weight=3)
13 G.add_edge(1,2, weight=4)
14 G.add_edge(2,3, weight=3)
15 G.add_edge(2,3, weight=4)
16 G.add_edge(3,4, weight=3)
17 G.add_edge(3,4, weight=4)
18 G.add_edge(4,5, weight=3)
19 G.add_edge(4,5, weight=4)
20
21
22 medio=[(1,2),(2,3),(3,4),(4,5)]
23 ruta=[(1,2),(2,3),(3,4),(4,5)]
24
25 #pos = nx.spring_layout(G)
26 #pos=nx.kamada_kawai_layout(G, dist=None, pos=None, weight='weight', scale=0.5,
27 #center=None, dim=2)
28
29 pos=nx.spectral_layout(G, weight='weight', scale=1, center=None, dim=2)
30
31 nx.draw_networkx_nodes(G, pos, node_size=800, node_color="#ecf815", node_shape='o')
32
33 nx.draw_networkx_edges(G, pos, edgelist=medio, width=6, alpha=0.8,
34 edge_color='black', style='dashed')
35
36 nx.draw_networkx_edges(G, pos, edgelist=ruta, width=4, alpha=0.5,
37 edge_color='r')
38
39 labels = []
40 labels[1] = r'Mat'
41 labels[2] = r'Hab'
42 labels[3] = r'Monty'
43 labels[4] = r'Torr'
44 labels[5] = r'Sing'
45
46 plt.axis('off')
47
48 nx.draw_networkx_labels(G, pos, labels, font_size=10)
49
50 plt.savefig("imagenes1/Fig10.eps")
51 plt.show()

```

grafo10lyspectral.py

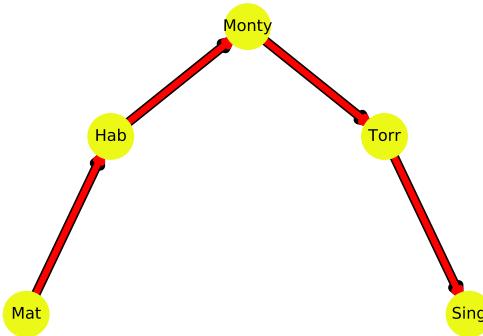


Figura 10: Ruta para viajar de Matanzas a Sinaloa

11. Multigrafo dirigido cíclico usando *spring layout*

En este tipo de grafos dirigidos cíclicos debe existir más de una arista entre un par de nodos, con dirección y además formarse al menos una figura cerrada dentro del grafo.

Un ejemplo que representa este tipo de grafo sería la representación del flujo que siguen los pacientes una vez que asisten a la consulta de cuerpo de guardia del ISSSTE, en la que las operaciones del proceso serían los nodos y la clasificación del tipo de paciente y su recorrido dentro de la instalación las aristas. En este caso el cliente al llegar pasa por la consulta de clasificación y ahí le asignan un color en función de la gravedad de su dolencia, se asumen tres colores, rojo, amarillo y verde, que van de mayor gravedad a menor respectivamente y de esta dependerá el tiempo de espera para pasar a la siguiente consulta donde se encuentra el doctor, existe tres consultas con doctores disponibles y de ahí pueden pasar al laboratorio, u a otras de las salas. Luego, del laboratorio pueden pasar nuevamente a la consulta del doctor para revisar los resultados y ocurriría un ciclo, o del doctor directo a la sala, luego al laboratorio y retornar nuevamente al doctor, y ocurriría otro ciclo. El algoritmo de acomodo que mejor se ajustó fue el *spring layout*. ~~Este grafo se muestra en la figura 11 en la página 25 donde se muestra la representación gráfica del mismo.~~

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G = nx . MultiDiGraph ()
5
6 G.add_node(1)
7 G.add_node(2)
8 G.add_node(3)
9 G.add_node(4)
10 G.add_node(5)
11
12 G.add_edge(1 ,2 , weight=3)
13 G.add_edge(1 ,2 , weight=4)
14 G.add_edge(1 ,2 , weight=5)
15 G.add_edge(2 ,3 , weight=3)
16 G.add_edge(2 ,3 , weight=4)
17 G.add_edge(2 ,4 , weight=3)
18
19 G.add_edge(2 ,5 , weight=3)
20 G.add_edge(3 ,2 , weight=4)
21 G.add_edge(3 ,2 , weight=3)
22
23 y=[(1 ,2) ,(2 ,3) ,(3 ,2)]
24 g=[(1 ,2) ,(2 ,3) ,(3 ,2)]
25 r=[(1 ,2) ,(2 ,4) ,(2 ,5)]
26
27 pos= nx.spring_layout(G, k=1, iterations=50, threshold=0.0001, weight='weight',
28 scale=1)
29
30 nx.draw_networkx_nodes(G, pos , node_size=400, node_color='b' , node_shape='o')
31
32 nx.draw_networkx_edges(G, pos , edgelist=y, width=10, alpha=0.2,
33 edge_color='y' , style='dashed')
34
35 nx.draw_networkx_edges(G, pos , edgelist=g, width=6, alpha=0.5,
36 edge_color='g')
37
38 nx.draw_networkx_edges(G, pos , edgelist=r, width=2, alpha=0.8,
39 edge_color='r')
40
41 labels = {}
42 labels [1] = r '1'
43 labels [2] = r '2'
44 labels [3] = r 'Lab'
45 labels [4] = r '3'
46 labels [5] = r '4'
47
48 plt.axis('off')
49
50 nx.draw_networkx_labels(G, pos , labels , font_size=10)
51
52 plt.savefig("imagenes1/Fig11.eps")
53 plt.show()

```

grafo11yspring.py

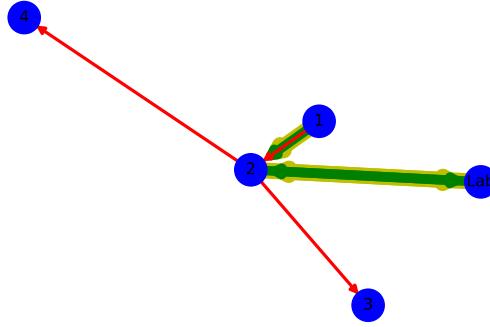


Figura 11: Recorrido del paciente en la sala de urgencias del ISSSTE

12. Multigrafo dirigido reflexivo usando *shell layout*

En los multigrafos dirigidos debe existir más de una arista entre un par de nodos, con dirección y además cada nodo debe llamarse a sí mismo.

Un ejemplo en el que el empleo de la teoría de grafos como los que se analizan en esta sección puede emplearse es en el estudio de la influencia de la religión en la sociedad, como un subejemplo de la aplicación en redes sociales [2]. Por ejemplo, si cada nodo es una persona religiosa creyente, ese nodo (persona) constantemente se realiza un autoexamen de conciencia a sí mismo, evaluando cómo ha sido su actitud con respecto a las variables a analizar en este autoexamen. Estas variables pueden ser operacionalizadas como cada una de las doctrinas de cada religión, las cuales varían de una religión a otra, pero son más de una y pueden agruparse por categorías. Estas variables representan las aristas, cada una puede tener un peso determinado según sea el interés del estudio, a su vez estas mismas doctrinas (convertidas en variable) serán las que cada persona creyente se encargará de transmitir a otras personas creyentes o no, pudiera decirse que se transmite el tipo de actitud que se debe tener ante cada una de estas variables de una persona a otra. En este ejemplo se puede evaluar el impacto en un grupo poblacional que puede tener un tipo de creencia u otro en la medida en que aumente la red. Para este ejemplo se reservó el algoritmo de acomodamiento *shell layout* es un algoritmo que acomoda los nodos en círculos concéntricos [4] este transmite la idea de expansión que es justamente lo que se desea reflejar en el ejemplo propuesto, si se llevara a gran escala. Una simplificación de este tipo de grafo se representa en la figura 12 en la página 27.

```

2 import matplotlib.pyplot as plt
3 import networkx as nx
4
5 G = nx.MultiDiGraph()
6
7 G.add_node(1)
8 G.add_node(2)
9 G.add_node(3)
10 G.add_node(4)
11 G.add_node(5)
12
13 a=[1]
14 b=[2]
15 c=[3]
16 d=[4]
17 e=[5]
18
19 G.add_edge(1,2, weight=3)
20 G.add_edge(1,2, weight=4)
21 G.add_edge(1,2, weight=5)
22 G.add_edge(2,3, weight=3)
23 G.add_edge(2,3, weight=4)
24 G.add_edge(2,3, weight=4)
25 G.add_edge(3,4, weight=3)
26 G.add_edge(3,4, weight=3)
27 G.add_edge(3,4, weight=3)
28 G.add_edge(4,5, weight=3)
29 G.add_edge(4,5, weight=3)
30 G.add_edge(4,5, weight=3)
31
32
33 y=[(1,2),(2,3),(3,4),(4,5)]
34 g=[(1,2),(2,3),(3,4),(4,5)]
35 r=[(1,2),(2,3),(3,4),(4,5)]
36
37 #pos = nx.spring_layout(G)
38 pos=nx.shell_layout(G, nlist = None , scale = 1 , center = None , dim = 2 )
39
40 nx.draw_networkx_nodes(G, pos, nodelist=a, node_size=300, node_color='black',
41 node_shape='o')
42 nx.draw_networkx_nodes(G, pos, nodelist=b, node_size=300, node_color='y',
43 node_shape='o')
44 nx.draw_networkx_nodes(G, pos, nodelist=c, node_size=300, node_color='b',
45 node_shape='o')
46 nx.draw_networkx_nodes(G, pos, nodelist=d, node_size=300, node_color='r',
47 node_shape='o')
48 nx.draw_networkx_nodes(G, pos, nodelist=e, node_size=300, node_color='g',
49 node_shape='o')
50
51 nx.draw_networkx_edges(G, pos, edgelist=y, width=8, alpha=1,
52 edge_color='y', style='dashed')
53
54 nx.draw_networkx_edges(G, pos, edgelist=g, width=6, alpha=0.5,
55 edge_color='g')
56
57 nx.draw_networkx_edges(G, pos, edgelist=r, width=2, alpha=0.8,

```

```

53 edge_color='r')
54
55 plt.axis('off')
56
57 plt.savefig("imagenes1/Fig12.eps")
58 plt.show()

```

grafo12lyshell.py

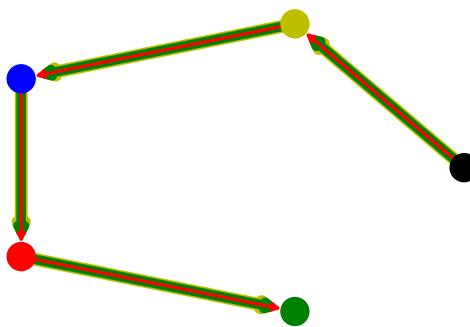


Figura 12: Expansión de doctrinas religiosas

Referencias

- [1] Pieter Swart Aric Hagberg, Dan Schult. *NetworkX Reference, Release 2.3rc1.dev20190113142952*, 2019. *VW*
- [2] Anwesha Chakraborty, Trina Dutta, Sushmita Mondal, and Asoke Nath. Application of graph theory in social media. *International Journal of Computer Sciences and Engineering*, 6:722–729, 10 2018. doi: 10.26438/ijcse/v6i10.722729.
- [3] Aric Hagberg. *title* <http://https://groups.google.com/forum/#!topic/networkx-discuss/qhHjh0CGP8A>. Accessed: 2019-02-25.
- [4] Desarrolladores NetworkX. *https://networkx.github.io/documentation/stable/reference/drawing.html*. Accessed: 2019-02-23.
- [5] Elisa Schaeffer. *Complejidad computacional de problemas y el análisis y diseño de algoritmos*. 2017. *VW*

Tarea No.2: Rectificada

Dayli Machado (5275)

3 de junio de 2019

1. Grafo simple no dirigido acíclico usando *bipartite layout*

Un grafo simple no dirigido acíclico, es aquel que no posee dirección en sus aristas, ni bucles, y no es reflexivo [5]. Este tipo de grafos suele representarse por una secuencia de nodos unidos por aristas, donde de cada nodo generalmente solo sale o llega una arista [1].

Este tipo de grafos puede emplearse para representar estructuras moleculares que sigan este comportamiento, un ejemplo sería la representación de alkanos de tipo lineal, donde los átomos se representan por los nodos y los enlaces entre estos por las aristas. El tipo de algoritmo de acomodamiento (*layout*) que se utilizó para lograr esta representación fue el *bipartite layout* el cual permite posicionar los nodos en dos líneas rectas y trazar ejes entre ellos [4], a partir de tener dos conjuntos de nodos diferentes se puede graficar la relación entre ellos. Se observa en la línea de código número 28 la forma de representar esta función empleada para la representación del grafo. Ver figura 1 en la página 2.

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Sun Feb 24 17:34:52 2019
4 @author: Dayli
5 """
6 import matplotlib.pyplot as plt
7 import networkx as nx
8
9 G = nx.Graph()
10
11 G.add_node(2)
12 G.add_node(4)
13
14 G.add_node(1)
15 G.add_node(3)
16 G.add_node(5)
17
18 G.add_edges_from([(1,2), (3,4)])
19 G.add_edges_from([(2,3), (4,5)])
20
21 labels = {}
22 labels[1] = r'H3C'
23 labels[2] = r'H2C'
24 labels[3] = r'CH2'
25 labels[4] = r'H2C'
26 labels[5] = r'CH3'
27
28 pos = nx.bipartite_layout(G, {1,3,5}, align='horizontal', scale=0.05)
29
30 nx.draw_networkx_nodes(G, pos, node_size=800, node_color='b')
31 nx.draw_networkx_edges(G, pos, width=3)
32 nx.draw_networkx_labels(G, pos, labels, font_size=10)
33 plt.axis('off')
34 plt.savefig("imagenes1/Fig01.eps")
35
36 plt.show()

```

grafoallaybi.py

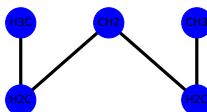


Figura 1: Molécula de alcano lineal usando *bipartite layout*

2. Grafo simple no dirigido cíclico usando *circular layout*

Un grafo simple no dirigido cíclico es aquel que no posee dirección en sus aristas, pero si se forma un ciclo que represente una figura cerrada que comience y termine en el mismo vértice, entonces es llamado cíclico [2]. En los grafos no dirigidos el flujo puede fluir en ambas direcciones.

Este tipo de grafos suele representarse por una secuencia de nodos unidos por aristas, pero estos pueden tener dentro un ciclo, o ser un ciclo en sí mismo. En la práctica pudiera ser la representación de las autopistas de una región dada, o también la representación de otro tipo de moléculas de alcanos que posean áreas cerradas en su representación, o las llamadas que se realizan interdepartamentales en una empresa.

Se toma de ejemplo las llamadas que se realizan entre los departamentos de producción, de marketing, comercial, calidad y planificación de la producción en una empresa, donde cada nodo representa un departamento y las aristas las llamadas que se realizan entre ellos. Para este tipo de grafo se emplea el *layout circular* como algoritmo de acomodamiento, pues permite ubicar los nodos en círculos [4]. De esta forma se visualiza mejor la distribución espacial de los nodos y la relación entre ellos que son las llamadas, pero como en este ejemplo la posición de los nodos puede ser aleatoria, se puede emplear también el *random layout* como otro algoritmo de acomodamiento pues este permite ubicar los nodos uniformemente al azar en el cuadro [4]. En la figura 2 en la página 4 se muestra la representación gráfica del mismo.

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Sun Feb 24 19:25:28 2019
4
5 @author: Dayli
6 """
7
8 import matplotlib.pyplot as plt
9 import networkx as nx
10
11 G=nx.Graph()
12
13 G.add_nodes_from([0,1,2,3,4])
14
15 G.add_edge(0,1)
16 G.add_edge(0,3)
17 G.add_edge(1,4)
18 G.add_edge(1,2)
19 G.add_edge(3,1)
20 G.add_edge(0,4)
21 G.add_edge(2,4)
22
23 labels = []
24 labels[0] = r'Markt'
25 labels[1] = r'Prod'
26 labels[2] = r'Cald'
27 labels[3] = r'Comerc'
28 labels[4] = r'Planif'
29
30 pos = nx.circular_layout(G, scale=0.5)
#pos = nx.spring_layout(G, scale=0.5)
31
32 nx.draw_networkx_nodes(G, pos, node_size=3000, node_color='#756b6b', node_shape='o')
33 nx.draw_networkx_edges(G, pos, width=2, alpha=0.5, edge_color='black')
34 nx.draw_networkx_labels(G, pos, labels, font_size=10, font_color='white')
35 plt.axis('off')
36 plt.savefig("imagenes1/Fig02.eps")
37 plt.show()

```

grafo2laycir.py

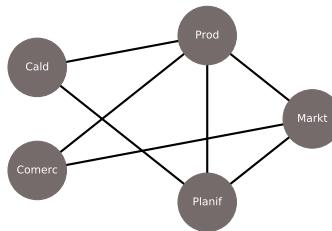


Figura 2: Llamadas interdepartamentales usando acomodo circular

3. Grafo simple no dirigido reflexivo usando *circular layout*

Un grafo simple no dirigido reflexivo es aquel en el cual no se permiten aristas múltiples, no posee dirección en sus aristas y cuenta al menos con un bucle, lo que se refiere a una arista reflexiva en la que coinciden el vértice de origen y el de destino [5].

Este grafo puede emplearse para representar el entrecruzamiento de grupos raciales, donde cada grupo representa un nodo diferente y las relaciones entre los grupos serán los hijos que tengan, siendo una relación no dirigida. Al relacionarse entre la misma raza estarían ocurriendo lazos reflexivos. Si se consideran los cuatro grupos representativos de las razas humanas según la clasificación de especialistas serían: blanco/caucásico, asiático/mongoloide, negroide/negro y australoide; esta clasificación posee dentro de cada grupo una clasificación de hasta 30 subgrupos, pero solo se representarán los cuatro principales. El *layout* que mejor refleja este tipo de grafo es el circular, la justificación es la misma explicada en la sección anterior. El código para representar este grafo fue tomado y adaptado del sitio [3], el mismo permitió insertar las imágenes como nodos. Ver figura 3 en la página 7.

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Sun Feb 24 23:03:42 2019
4
5 @author: Dayli
6 """
7 import networkx as nx
8 import matplotlib.pyplot as plt
9 import matplotlib.image as mpimg
10
11 # image from http://matplotlib.sourceforge.net/users/image_tutorial.html
12 path = "C:/Users/lapi7/Desktop/Para tarea 2/razas/"
13 img1=mpimg.imread("1.jpg")
14 img2=mpimg.imread("2.jpg")
15 img3=mpimg.imread("3.jpg")
16 img4=mpimg.imread("4.jpg")
17
18 G=nx.Graph()
19
20 G.add_nodes_from([1,2,3,4])
21
22 G.add_edges_from([(1,1), (1,2), (1,3), (1,4)])# Cada nodo es reflexivo
23 G.add_edges_from([(2,2), (2,1), (2,3), (2,4)])#
24 G.add_edges_from([(3,3), (3,1), (3,2), (3,4)])#
25 G.add_edges_from([(4,4), (4,1), (4,3), (4,2)])#
26
27 G.node[1]["image"]=img1
28 G.node[2]["image"]=img2
29 G.node[3]["image"]=img3
30 G.node[4]["image"]=img4
31
32 pos=nx.circular_layout(G, scale=0.5)
33
34 fig=plt.figure(figsize=(13,13))
35 ax=plt.subplot(1,1,1)
36 ax.set_aspect("equal")
37 nx.draw_networkx_edges(G, pos, ax=ax, width=2)
38
39 plt.xlim(-0.5,1.5)
40 plt.ylim(-0.5,1.5)
41
42 plt.axis("off")
43
44 trans=ax.transData.transform
45 trans2=fig.transFigure.inverted().transform
46
47 piesize=0.2 # this is the image size
48 p2=piesize/2.0
49 for n in G:
50     xx,yy=trans(pos[n]) # figure coordinates
51     xa,ya=trans2((xx,yy)) # axes coordinates
52     a = plt.axes([xa-p2,ya-p2, piesize , piesize])
53     a.set_aspect("equal")
54     a.imshow(G.node[n]["image"])
55     a.axis("off")
56
57 plt.savefig("imagenes1/Fig03.eps", bbox_inches='tight')

```

```
58 #nx.draw(G)
59 plt.show()
```

grafo3laycir.py

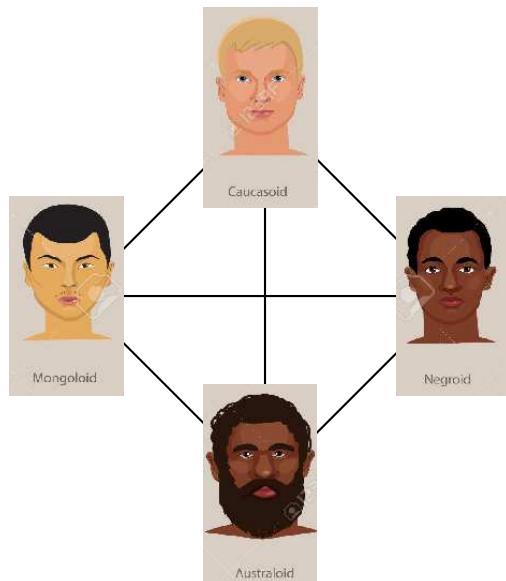


Figura 3: Grupos raciales usando *circular layout*

4. Gráfo simple dirigido acíclico usando *spring layout*

Un grafo simple dirigido acíclico, es aquel que posee una dirección en sus aristas, y no posee bucles o reflexividad, ni ciclos [5].

En este caso, ejemplos de la vida real son aquellos que poseen un origen del que puede salir una o varias aristas por diferentes caminos sin que entre ellas existan ciclos y posean un destino final diferente al del origen.

Los árboles genealógicos, los organigramas en las empresas, el flujo de procesos industriales que no posean ciclos, con un inicio y un fin, pudieran resultar ser ejemplos de la vida real en los que se puede aplicar este tipo de grafos.

Como algoritmo de acomodo para representar este grafo se usó el *spring layout*, el cual permite representar de manera sencilla las aristas rectas que salen de un nodo a otro y fijar la posición haciendo uso de sus parámetros [4] siendo el mismo algoritmo empleado en la tarea anterior para este tipo de grafo, sin embargo en este ejercicio se profundizó en el uso de sus parámetros.

Para la representación inicial se fija la posición de los nodos y el centro del grafo, además se estableció una distancia estándar entre ellos, también se emplea el degradado de color para indicar la transformación que recibe un objeto dado en una línea de producción. Ver figura 4 en la página 10.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G=nx.DiGraph ()
5
6 G.add_node(1)
7 G.add_node(2)
8 G.add_node(3)
9 G.add_node(4)
10 G.add_node(5)
11
12 nod1={1}
13 nod2={2}
14 nod3={3}
15 nod4={4}
16 nod5={5}
17
18 G.add_edge(1 ,2)
19 G.add_edge(2 ,3)
20 G.add_edge(3 ,4)
21 G.add_edge(4 ,5)
22
23 positions = { 5:(0 , 10) , 4:(0 , 20) , 3:(0 ,30) , 2:(0 ,40) , 1:(0 ,50) }
24 #fixeds={1,2}
25 pos = nx.spring_layout(G, k=3, fixed=None, pos=positions , center=(0,0) , iterations
   =50)
26
27 nx.draw_networkx_nodes(G, pos , nodelist=nod1 , node_size=400,node_color='#580cf0' ,
   node_shape='o')
28 nx.draw_networkx_nodes(G, pos , nodelist=nod2 , node_size=400,node_color='#6332c5' ,
   node_shape='o')
29 nx.draw_networkx_nodes(G, pos , nodelist=nod3 , node_size=400,node_color='#6240a7' ,
   node_shape='o')
30 nx.draw_networkx_nodes(G, pos , nodelist=nod4 , node_size=400,node_color='#574184' ,
   node_shape='o')
31 nx.draw_networkx_nodes(G, pos , nodelist=nod5 , node_size=400,node_color='#433958' ,
   node_shape='o')
32
33
34 nx.draw_networkx_edges(G, pos , width=2, alpha=0.8, edge_color='black')
35 plt.axis('off')
36 #plt.axis('off')
37 plt.savefig("imagenes1/Fig04 .eps")
38 plt.show()
39
40 #plt.show()

```

grafo4lyspring.py

5. Gráfo simple dirigido cíclico usando *bipartite layout*

Un grafo simple dirigido cíclico es aquel que posee una dirección en sus aristas y una figura cerrada que comienza y termina en el mismo vértice.



Figura 4: Transformación de un objeto en una línea de producción usando *spring layout*

Para este caso ejemplos de la vida real son aquellos que poseen un origen del que puede salir una o varias aristas por diferentes caminos, y que en un determinado nodo regresan a alguno anterior de modo que se forme uno o más ciclos. El destino final suele ser diferente al del origen.

Como aplicación real de este tipo de grafo se explica el flujo de procesos industriales o artesanales que posean ciclos, por ejemplo: la elaboración artesanal de jugo de naranja, al representar este proceso en un flujograma mediante un diagrama OPERIN, en el que se reflejen sus operaciones. Una vez que se llegue al paso de exprimir las naranjas, se cae en un ciclo volviendo a la operación anterior de seleccionar otra y otra naranja hasta cubrir toda la capacidad del extractor de jugo. En esta segunda tarea el algoritmo de acomodo empleado que se usó para representarlo fue el *bipartite layout* pues resulta más sencillo y directo para graficar los nodos que salen de la línea de producción y que forman el ciclo con los que se mantienen en ella, se emplea el parámetro *top* para separar el conjunto de nodos que no están en la línea principal de producción. Con este algoritmo se aprecia una mejor representación del grafo, aunque es válido aclarar que este ejemplo no se comporta exactamente como un grafo bipartito, pero este fue el algoritmo de acomodo que mejor lo representó. En este ejemplo las operaciones son los nodos y los enlaces la transformación que va teniendo la naranja como se muestra en la figura 5 en la página 12.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G=nx.DiGraph ()
5
6 G.add_node(1)
7 G.add_node(2)
8 G.add_node(3)
9
10 G.add_node(6)
11 G.add_node(4)
12 G.add_node(5)
13
14 G.add_edge(1 ,2)
15 G.add_edge(2 ,3)
16 G.add_edge(3 ,4)
17
18 G.add_edge(4 ,5)
19 G.add_edge(5 ,6)
20 G.add_edge(3 ,6)
21
22 G.add_cycle ([5 ,3])
23
24 labels = {}
25 labels [1] = r '$Op1$'
26 labels [2] = r '$Op2$'
27 labels [3] = r '$Op3$'
28 labels [4] = r '$Op4$'
29 labels [5] = r '$Op5$'
30 labels [6] = r '$Op6$'
31
32 pos = nx.bipartite_layout(G, {4,5}, align='horizontal', scale=0.7)
33
34 nx.draw_networkx_nodes(G, pos, node_size=500, node_color='r', node_shape='o')
35 nx.draw_networkx_edges(G, pos, width=2, alpha=0.8, edge_color='black')
36 nx.draw_networkx_labels(G, pos, labels, font_size=10)
37
38 plt.axis('off')
39 plt.savefig("imagenes1/Fig05.eps")
40 plt.show()

```

grafo5laybi.py

6. Gráfo simple dirigido reflexivo usando *circular layout*

La diferencia con este grafo y el anterior reflexivo visto, es que en este caso las aristas sí tienen sentido y el flujo que se analice debe ir en una dirección. En este caso deberá existir un nodo que se llame a sí mismo, o sea, que posea al menos un bucle.

Un ejemplo en el que se puede emplear la teoría de grafos de este tipo es para representar la disposición de personas con grupos sanguíneos diferentes a recibir sangre compatible o no. La reflexividad está dada en los casos en que cada grupo sanguíneo puede recibir de otros grupos y

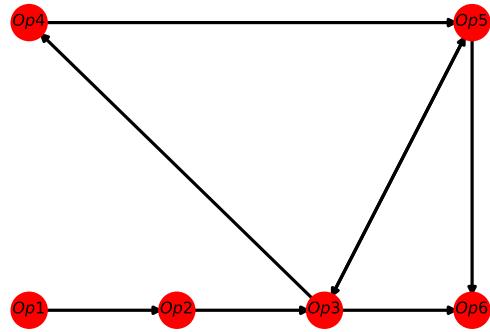


Figura 5: Flujo cíclico de una línea de producción artesanal de jugo de naranja

de sí mismo, excepto el grupo O- que sólo puede recibir del mismo tipo. Se cambia el algoritmo de acomodo anterior por el *circular layout* haciendo uso de sus parámetros, y se evidencia más estabilidad en las iteraciones de la figura que mantienen la misma posición deseada cada vez que se itera, sin tener que entrarla manualmente. Este se muestra en la figura 6 en la página 14.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G = nx.DiGraph()
5 G.add_node(1)
6 G.add_node(2)
7 G.add_node(3)
8 G.add_node(4)
9 G.add_node(5)
10 G.add_node(6)
11 G.add_node(7)
12 G.add_node(8)
13
14 node_a = {1,2}
15 node_b = {3,4}
16 node_ab = {5,6}
17 node_o = {7,8}
18 #Se cambiaron los layout y el mejor resultado fue con el circular
19 #pos = nx.spring_layout(G)
20 pos = nx.circular_layout(G, dim=2, scale=1, center=(0,0))
21 #pos = nx.shell_layout(G)
22 #pos = nx.rescale_layout(G)
23
24 G.add_edges_from([(1,1), (8,1), (7,1), (2,1)])# Cada nodo es reflexivo pero no me
25     sale la flechita
26 G.add_edges_from([(2,2), (8,2)])#igualmente no sale el reflexivo en si mismo
27 G.add_edges_from([(3,3), (8,3), (7,3), (4,3)])#no sale el reflexivo en si mismo
28 G.add_edges_from([(4,4), (8,4)])#es reflexivo en si mismo
29 G.add_edges_from([(5,5), (1,5), (2,5), (3,5), (4,5), (6,5), (7,5), (8,5)])# es
30     reflexivo en si mismo
31 G.add_edges_from([(6,6), (4,6), (8,6), (2,6)])
32 G.add_edges_from([(7,7), (8,7)])
33 G.add_edges_from([(8,8)])
34
35 nx.draw_networkx_nodes(G, pos, nodelist=node_a, node_size=800, node_color='g',
36     node_shape='o', alpha=0.3)
37 nx.draw_networkx_nodes(G, pos, nodelist=node_b, node_size=800, node_color='y',
38     node_shape='o', alpha=0.3)
39 nx.draw_networkx_nodes(G, pos, nodelist=node_ab, node_size=800, node_color='b',
40     node_shape='o', alpha=0.3)
41 nx.draw_networkx_nodes(G, pos, nodelist=node_o, node_size=800, node_color='r',
42     node_shape='o', alpha=0.8)
43 #nx.draw_networkx_nodes(G, pos, nodelist=node_o, node_size=800, node_color='r',
44     node_shape='on', alpha=0.1)
45 labels = []
46 labels[1] = r'A+'
47 labels[2] = r'A-'
48 labels[3] = r'B+'
49 labels[4] = r'B-'
50 labels[5] = r'AB+'
51 labels[6] = r'AB-'
52 labels[7] = r'O+'
53 labels[8] = r'O-'
54
55 nx.draw_networkx_labels(G, pos, labels, font_size=10)
56
57 #pos1 = pos
58 #

```

```

51 #for key, value in G.edges:
52 #    print("(", pos[key][0] - 0.5, ", ", pos[key][1] - 0.5, ")\n")
53 #    pos1[key][0] = pos1[key][0] + 0.1
54 #    pos1[key][1] = pos1[key][1]
55 #
56
57 nx.draw_networkx_edges(G, pos, width=2, alpha=1, edge_color='black', scale=0.5 )
58 #print(pos)
59 plt.axis('on')
60 plt.savefig("imagenes1/Fig06.eps")
61 plt.show()

```

grafo6laycir.py

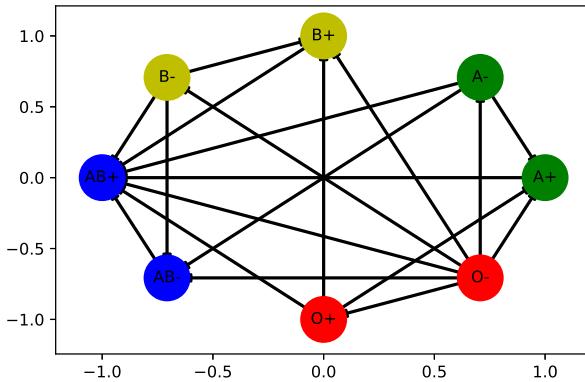


Figura 6: Compatibilidad entre grupos sanguíneos

7. Multigrafo no dirigido acíclico usando *spring layout*

Un multigrafo no dirigido acíclico es cuando existe más de una arista entre un par de vértices. Si no se permiten aristas múltiples, el grafo es simple [5]. Será no dirigido cuando las aristas no posean sentido restringido, sino que el flujo fluye en ambas direcciones, y en este caso no presenta ciclos.

Un ejemplo práctico de multígrafo no dirigido acíclico es la representación de determinados enlaces moleculares, como por ejemplo: un enlace de doble éster, como es el caso del éster sulfúrico. Para la representación de los elementos con este tipo de grafo se puede emplear el algoritmo de acomodo *spring layout*, es el mismo que se usó en el caso anterior pero esta vez se hace uso de sus parámetros sin fijar la posición [4], de este modo se obtiene una representación más ajustada a la realidad, también pudiera emplearse el *Kamada Kawai layout* pero al compararlos e iterar el *spring layout* devuelve variantes más atractivas. El grafo usando *spring layout* se muestra en la figura 7 en la página 16.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G=nx.MultiGraph()
5
6 G.add_node(1)
7 G.add_node(2)
8 G.add_node(3)
9 G.add_node(4)
10 G.add_node(5)
11 G.add_node(6)
12 G.add_node(7)
13
14 node_o = {4,5,6,7}
15 node_r = {1,2,3}
16
17 G.add_edge(4,3, weight=3)
18 G.add_edge(4,3, weight=5)
19 G.add_edge(3,7, weight=3)
20 G.add_edge(3,7, weight=5)
21 G.add_edge(1,5)
22 G.add_edge(5,3)
23 G.add_edge(3,6)
24 G.add_edge(6,2)
25
26 blue=[(4,3),(3,7),(1,5),(5,3),(3,6),(6,2)]
27 red=[(4,3),(3,7)]
28
29 labels = {}
30 labels[1] = r'R1'
31 labels[2] = r'R2'
32 labels[3] = r'S'
33 labels[4] = r'O'
34 labels[5] = r'O'
35 labels[6] = r'O'
36 labels[7] = r'O'
37
38 pos=nx.spring_layout(G, k=0.1, iterations=300, threshold=0.0001, weight='weight',
39 scale=1)
40 nx.draw_networkx_nodes(G, pos, nodelist=node_o, node_size=600, node_color='b',
41 node_shape='o')
42 nx.draw_networkx_nodes(G, pos, nodelist=node_r, node_size=600, node_color='y',
43 node_shape='s')
44
45 nx.draw_networkx_edges(G, pos, edgelist=blue, width=6, alpha=0.5, edge_color='b',
46 style='dashed')
47 nx.draw_networkx_edges(G, pos, edgelist=red, width=6, alpha=0.5, edge_color='r')
48
49 nx.draw_networkx_labels(G, pos, labels, font_size=10)
50
51 plt.axis('on')
52 plt.savefig("imagenes1/Fig07.eps")
53 plt.show()

```

grafo7lyspring.py

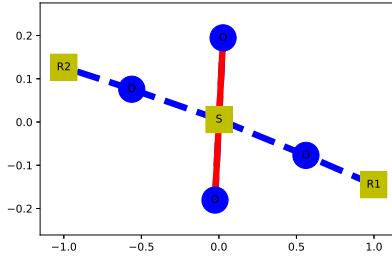


Figura 7: Mólecula de éster sulfúrico

8. Multigrafo no dirigido cíclico usando *Kamada Kawai layout*

Cuando exista más de una arista entre un par de vértices, el grafo se llama multígrafo. En este caso será no dirigido cuando las aristas no poseen sentido restringido, sino que el flujo fluye en ambas direcciones, y deberán existir zonas cerradas, o sea, que regresen al vértice inicial.

Este tipo de grafo es de gran utilidad para representar el comportamiento de redes sociales y neuronales [2]. El ejemplo práctico mostrado es cuando tenemos necesidad de comunicarnos con alguien por las vías de comunicación que tenemos en la actualidad, donde cada vía de comunicación seleccionada tendrá un costo (peso), y existen muchas vías para comunicarnos con otras personas, y a su vez para otras personas se comuniquen con nosotros, por lo que no es dirigido, pero sí cíclico, debido a los lazos que se pueden formar entre las personas que necesitan comunicarse. El algoritmo de acomodo empleado para este ejemplo según sus prestaciones fue el *Kamada Kawai layout* pues este tiene en cuenta para la representación el peso de la longitud del camino [4] y devuelve grafos más atractivos para el ejemplo en cuestión. El ejemplo se restringe a solo dos vías de comunicación, ya sea por *whatsapp*, o por llamadas telefónicas, aunque en la práctica existen muchas más. Este grafo se muestra en la figura 8 en la página 18.

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Feb 25 20:09:39 2019
4
5 @author: lapi7
6 """
7
8 import matplotlib.pyplot as plot
9 import networkx as nx
10
11 G = nx.MultiGraph()
12
13 G.add_node(1)
14 G.add_node(2)
15 G.add_node(3)
16 G.add_node(4)
17 G.add_node(5)
18
19 G.add_edge(1,2, weight=3)
20 G.add_edge(1,2, weight=4)
21 G.add_edge(2,3, weight=3)
22 G.add_edge(2,3, weight=4)
23 G.add_edge(3,4, weight=3)
24 G.add_edge(3,4, weight=4)
25 G.add_edge(4,5, weight=3)
26 G.add_edge(4,5, weight=4)
27 G.add_edge(5,1, weight=3)
28 G.add_edge(5,1, weight=4)
29 G.add_edge(1,3, weight=3)
30 G.add_edge(1,3, weight=4)
31 G.add_edge(5,3, weight=3)
32 G.add_edge(5,3, weight=4)
33
34 green=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]
35 yellow=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]
36
37 pos = nx.kamada_kawai_layout(G, dist=None, pos=None, weight='weight', scale=0.5,
38 center=None, dim=2)
39 plt.axis('off')
40
41 nx.draw_networkx_nodes(G, pos, node_size=400, node_color='r', node_shape='o')
42
43 nx.draw_networkx_edges(G, pos, edgelist=green, width=3, alpha=0.5,
44 edge_color='g', style='dashed')
45 nx.draw_networkx_edges(G, pos, edgelist=yellow, width=4, alpha=0.5,
46 edge_color='y')
47
48 plt.savefig("imagenes1/Fig08.eps")
49
50 plt.show()

```

grafo8lykk.py

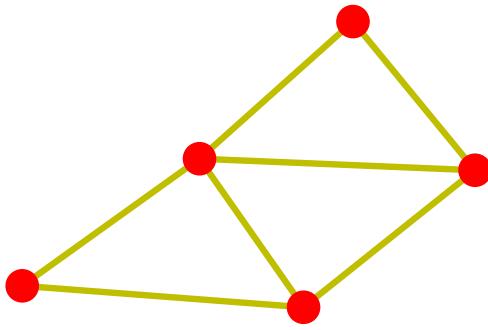


Figura 8: Vías de comunicación entre las personas

9. Multigrafo no dirigido reflexivo usando *random layout*

Este caso se refiere a un multígrafo en el que existe más de una arista entre un par de vértices, sin dirección entre estas, pero de un nodo deben salir más de una arista que regresen al mismo nodo, sin pasar por otro.

Una aplicación pudiera ser la relación que existe entre cinco signos zodiacales, donde cada nodo representa a las personas de un signo determinado y de cada uno de ellos puede salir una arista que represente las relaciones positivas entre los diferentes signos y otra las relaciones negativas, y a su vez las personas del mismo signo se relacionarán con sus iguales de manera positiva o negativa, lo que representa la reflexibilidad. Este es un grafo que no es exigente para su representación y aunque un algoritmo de acomodo circular puede ser apropiado, se propone usar uno aleatorio que funciona igual en este ejemplo y consume menos. Este grafo se muestra para cinco signos cualesquiera del zodiaco en la figura 9 en la página 20.

```

1 # -*- coding: utf-8 -*-
2 """
3 Created on Mon Feb 11 16:09:42 2019
4
5 @author: lapi7
6 """
7
8 import matplotlib.pyplot as plot
9 import networkx as nx
10
11 G=nx.MultiGraph()
12
13 G.add_node(1)
14 G.add_node(2)
15 G.add_node(3)
16 G.add_node(4)
17 G.add_node(5)
18
19 G.add_edge(1,2, weight=4)
20 G.add_edge(2,3, weight=3)
21 G.add_edge(2,3, weight=4)
22 G.add_edge(3,4, weight=3)
23 G.add_edge(3,4, weight=4)
24 G.add_edge(4,5, weight=3)
25 G.add_edge(4,5, weight=4)
26 G.add_edge(5,1, weight=3)
27 G.add_edge(5,1, weight=4)
28 G.add_edge(1,3, weight=3)
29 G.add_edge(1,3, weight=4)
30 G.add_edge(5,3, weight=3)
31 G.add_edge(5,3, weight=4)
32
33 node_A = {1}
34 node_S = {2}
35 node_E = {3}
36 node_V = {4}
37 node_C = {5}
38
39
40 green=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]
41 yellow=[(1,2),(2,3),(3,4),(4,5),(5,1),(1,3),(5,3)]
42
43 pos = nx.random_layout(G, dim=2, center=None) # Se puede usar el random en este
44 caso
45 #pos = nx.circular_layout(G, dim=2, center=None)
46 plt.axis('off')
47
48 nx.draw_networkx_nodes(G, pos, nodelist=node_A, node_size=300, node_color='g',
49 node_shape='o')
50 nx.draw_networkx_nodes(G, pos, nodelist=node_S, node_size=300, node_color='y',
51 node_shape='o')
52 nx.draw_networkx_nodes(G, pos, nodelist=node_E, node_size=300, node_color='b',
53 node_shape='o')
54 nx.draw_networkx_nodes(G, pos, nodelist=node_V, node_size=300, node_color='r',
55 node_shape='o')
56 nx.draw_networkx_nodes(G, pos, nodelist=node_C, node_size=300, node_color='black',

```

```

53     node_shape='o')
54
55 nx.draw_networkx_edges(G, pos, edgelist=green, width=3, alpha=0.5,
56 edge_color='g', style='dashed')
57 nx.draw_networkx_edges(G, pos, edgelist=yellow, width=4, alpha=0.5,
58 edge_color='y')
59 plt.savefig("imagenes1/Fig09.eps")
60
61 plt.show()

```

grafo9lyrandom.py

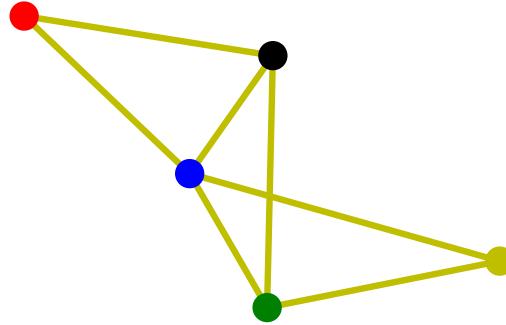


Figura 9: Relación entre personas del mismo signo zodiacal

10. Multigrafo dirigido acíclico usando *spectral layout*

Para definir un multigrafo dirigido acíclico debe existir más de una arista entre un par de nodos, con dirección y no formar ninguna figura cerrada dentro del grafo.

Este ejemplo aplicado a la práctica puede ser la representación de un viajero que desea ir de una ciudad a otra para visitarlas y tiene que elegir entre varias rutas posibles para llegar, o pudiera elegir también entre diferentes medios de transporte para trasladarse entre las ciudades, o una combinación de ambos. En este caso las ciudades serían los nodos, y las rutas o medios de transporte posibles a elegir entre un nodo u otro serían las aristas. Una ruta pudiera ser ir de la ciudad de Matanzas en Cuba, a la Habana, de esta a Monterrey, de Monterrey a Torreón y de ahí a Sinaloa. Para devolver el acomodo de estos nodos puede emplearse el algoritmo *spectral layout* que no se ha empleado en las secciones anteriores, este devuelve una forma sinusoidal pues refleja el algoritmo basado en la utilización de los vectores del gráfico laplaciano [4], lo cual no afecta a la visualización del grafo ejemplo. Asimismo se puede emplear el *Kamada Kawai layout* con resultados gráficos

más direccionalizados y apropiados para el ejemplo, pero ya se usó en secciones anteriores. El grafo se muestra en la figura 10 en la página 22.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G=nx.MultiDiGraph()
5
6 G.add_node(1)
7 G.add_node(2)
8 G.add_node(3)
9 G.add_node(4)
10 G.add_node(5)
11
12 G.add_edge(1,2, weight=3)
13 G.add_edge(1,2, weight=4)
14 G.add_edge(2,3, weight=3)
15 G.add_edge(2,3, weight=4)
16 G.add_edge(3,4, weight=3)
17 G.add_edge(3,4, weight=4)
18 G.add_edge(4,5, weight=3)
19 G.add_edge(4,5, weight=4)
20
21 medio=[(1,2),(2,3),(3,4),(4,5)]
22 ruta=[(1,2),(2,3),(3,4),(4,5)]
23
24 #pos = nx.spring_layout(G)
25 #pos=nx.kamada_kawai_layout(G, dist=None, pos=None, weight='weight', scale=0.5,
26   center=None, dim=2)
27
28 pos=nx.spectral_layout(G, weight='weight', scale=1, center=None, dim=2)
29
30 nx.draw_networkx_nodes(G, pos, node_size=800, node_color="#ecf815", node_shape='o')
31
32 nx.draw_networkx_edges(G, pos, edgelist=medio, width=6, alpha=0.8,
33   edge_color='black', style='dashed')
34
35 nx.draw_networkx_edges(G, pos, edgelist=ruta, width=4, alpha=0.5,
36   edge_color='r')
37
38 labels = {}
39 labels[1] = r'Mat'
40 labels[2] = r'Hab'
41 labels[3] = r'Monty'
42 labels[4] = r'Torr'
43 labels[5] = r'Sing'
44
45 plt.axis('off')
46
47 nx.draw_networkx_labels(G, pos, labels, font_size=10)
48
49 plt.savefig("imagenes1/Fig10.eps")
50 plt.show()

```

grafo10lyspectral.py

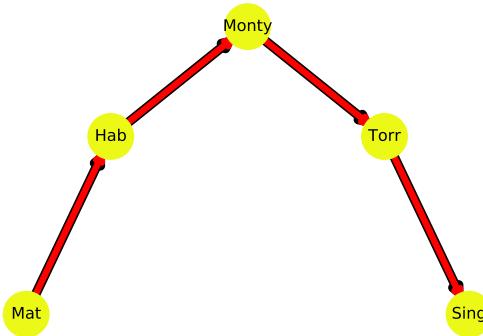


Figura 10: Ruta para viajar de Matanzas a Sinaloa

11. Multigrafo dirigido cíclico usando *spring layout*

En este tipo de grafos dirigidos cíclicos debe existir más de una arista entre un par de nodos, con dirección y además formarse al menos una figura cerrada dentro del grafo.

Un ejemplo que representa este tipo de grafo sería la representación del flujo que siguen los pacientes una vez que asisten a la consulta de cuerpo de guardia del ISSSTE, en la que las operaciones del proceso serían los nodos y la clasificación del tipo de paciente y su recorrido dentro de la instalación las aristas. En este caso el cliente al llegar pasa por la consulta de clasificación y ahí le asignan un color en función de la gravedad de su dolencia, se asumen tres colores, rojo, amarillo y verde, que van de mayor gravedad a menor respectivamente y de esta dependerá el tiempo de espera para pasar a la siguiente consulta donde se encuentra el doctor, existe tres consultas con doctores disponibles y de ahí pueden pasar al laboratorio, u a otras de las salas. Luego, del laboratorio pueden pasar nuevamente a la consulta del doctor para revisar los resultados y ocurriría un ciclo, o del doctor directo a la sala, luego al laboratorio y retornar nuevamente al doctor, y ocurriría otro ciclo. El algoritmo de acomodo que mejor se ajustó fue el *spring layout*. Este grafo se muestra en la figura 11 en la página 24 donde se muestra la representación gráfica del mismo.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G=nx.MultiDiGraph()
5
6 G.add_node(1)
7 G.add_node(2)
8 G.add_node(3)
9 G.add_node(4)
10 G.add_node(5)
11
12 G.add_edge(1,2, weight=3)
13 G.add_edge(1,2, weight=4)
14 G.add_edge(1,2, weight=5)
15 G.add_edge(2,3, weight=3)
16 G.add_edge(2,3, weight=4)
17 G.add_edge(2,4, weight=3)
18
19 G.add_edge(2,5, weight=3)
20 G.add_edge(3,2, weight=4)
21 G.add_edge(3,2, weight=3)
22
23 y=[(1,2),(2,3),(3,2)]
24 g=[(1,2),(2,3),(3,2)]
25 r=[(1,2),(2,4),(2,5)]
26
27 pos= nx.spring_layout(G, k=1, iterations=50, threshold=0.0001, weight='weight',
28 scale=1)
29
30 nx.draw_networkx_nodes(G, pos, node_size=400, node_color='b', node_shape='o')
31
32 nx.draw_networkx_edges(G, pos, edgelist=y, width=10, alpha=0.2,
33 edge_color='y', style='dashed')
34
35 nx.draw_networkx_edges(G, pos, edgelist=g, width=6, alpha=0.5,
36 edge_color='g')
37
38 nx.draw_networkx_edges(G, pos, edgelist=r, width=2, alpha=0.8,
39 edge_color='r')
40
41 labels = {}
42 labels[1] = r'1'
43 labels[2] = r'2'
44 labels[3] = r'Lab'
45 labels[4] = r'3'
46 labels[5] = r'4'
47
48 plt.axis('off')
49
50 nx.draw_networkx_labels(G, pos, labels, font_size=10)
51
52 plt.savefig("imagenes1/Fig11.eps")
53 plt.show()

```

grafo11yspring.py

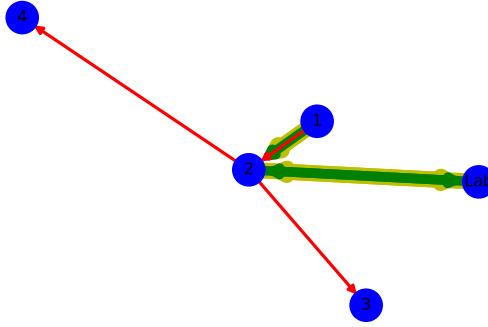


Figura 11: Recorrido del paciente en la sala de urgencias del ISSSTE

12. Multigrafo dirigido reflexivo usando *shell layout*

En los multigráficos dirigidos debe existir más de una arista entre un par de nodos, con dirección y además cada nodo debe llamarse a sí mismo.

Un ejemplo en el que el empleo de la teoría de grafos como los que se analizan en esta sección puede emplearse es en el estudio de la influencia de la religión en la sociedad, como un subejemplo de la aplicación en redes sociales.[2]. Por ejemplo, si cada nodo es una persona religiosa creyente, ese nodo (persona) constantemente se realiza un autoexamen de conciencia a sí mismo, evaluando cómo ha sido su actitud con respecto a las variables a analizar en este autoexamen. Estas variables pueden ser operacionalizadas como cada una de las doctrinas de cada religión, las cuales varían de una religión a otra, pero son más de una y pueden agruparse por categorías. Estas variables representan las aristas, cada una puede tener un peso determinado según sea el interés del estudio, a su vez estas mismas doctrinas (convertidas en variable) serán las que cada persona creyente se encargará de transmitir a otras personas creyentes o no, pudiera decirse que se transmite el tipo de actitud que se debe tener ante cada una de estas variables de una persona a otra. En este ejemplo se puede evaluar el impacto en un grupo poblacional que puede tener un tipo de creencia u otro en la medida en que aumente la red. Para este ejemplo se reservó el algoritmo de acomodamiento *shell layout* es un algoritmo que acomoda los nodos en círculos concéntricos [4] este transmite la idea de expansión que es justamente lo que se desea reflejar en el ejemplo propuesto, si se llevara a gran escala. Una simplificación de este tipo de grafo se representa en la figura 12 en la página 26.

```

1 import matplotlib.pyplot as plt
2 import networkx as nx
3
4 G=nx.MultiDiGraph()
5
6 G.add_node(1)
7 G.add_node(2)
8 G.add_node(3)
9 G.add_node(4)
10 G.add_node(5)
11
12 G.add_edge(1,2, weight=3)
13 G.add_edge(1,2, weight=4)
14 G.add_edge(1,2, weight=5)
15 G.add_edge(2,3, weight=3)
16 G.add_edge(2,3, weight=4)
17 G.add_edge(2,3, weight=4)
18 G.add_edge(3,4, weight=3)
19 G.add_edge(3,4, weight=3)
20 G.add_edge(3,4, weight=3)
21 G.add_edge(4,5, weight=3)
22 G.add_edge(4,5, weight=3)
23 G.add_edge(4,5, weight=3)
24
25 y=[(1,2),(2,3),(3,4),(4,5)]
26 g=[(1,2),(2,3),(3,4),(4,5)]
27 r=[(1,2),(2,3),(3,4),(4,5)]
28
29 pos=nx.shell_layout(G, nlist = None , scale = 1 , center = None , dim = 2 )
30 list=[]
31 list=G.nodes()
32 listacolor=['black', 'y', 'b', 'r', 'g']
33
34 for i in list:
35     nx.draw_networkx_nodes(G, pos, node=i, node_size=300, node_color=listacolor,
36     node_shape='o')
37 nx.draw_networkx_edges(G, pos, edgelist=y, width=8, alpha=1,
38 edge_color='y', style='dashed')
39 nx.draw_networkx_edges(G, pos, edgelist=g, width=6, alpha=0.5,
40 edge_color='g')
41 nx.draw_networkx_edges(G, pos, edgelist=r, width=2, alpha=0.8,
42 edge_color='r')
43 plt.axis('off')
44 plt.savefig("imagenes1/Fig13.eps")
45 plt.show()

```

12rectificado.py

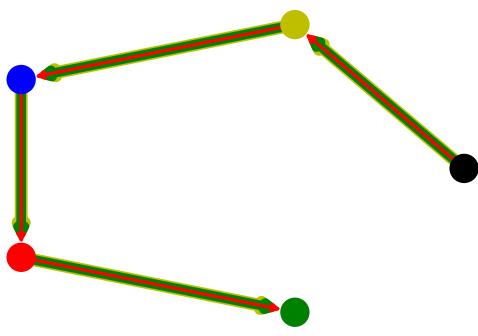


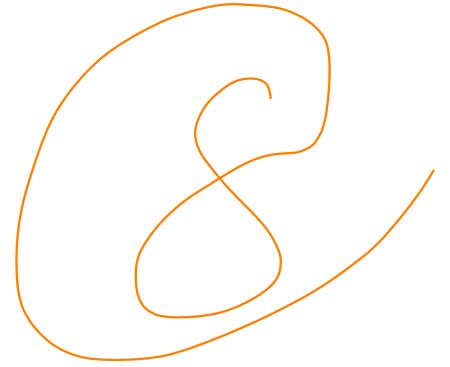
Figura 12: Expansión de doctrinas religiosas

Referencias

- [1] Pieter Swart Aric Hagberg, Dan Schult. *NetworkX Reference, Release 2.3rc1.dev20190113142952*, 2019. URL <https://networkx.github.io/documentation/stable/reference/index.html>.
- [2] Anwesha Chakraborty, Trina Dutta, Sushmita Mondal, and Asoke Nath. Application of graph theory in social media. *International Journal of Computer Sciences and Engineering*, 6:722–729, 10 2018. doi: 10.26438/ijcse/v6i10.722729.
- [3] Aric Hagberg. Forum de networkx. <http://https://groups.google.com/forum/#!topic/networkx-discuss/qhHjh0CGP8A>. Accessed: 2019-02-25.
- [4] Desarrolladores NetworkX. <https://networkx.github.io/documentation/stable/reference/drawing.html>. Accessed: 2019-02-23.
- [5] Elisa Schaeffer. Complejidad computacional de problemas y el análisis y diseño de algoritmos, 2017. URL <https://elisa.dyndns-web.com/teaching/aa/pdf/aa.pdf>.

Tarea No.3: Flujo en Redes

Dayli Machado (5275)



19 de marzo de 2019

1. Grafos y algoritmos empleados

Para la realización de esta tarea se crearon nuevos grafos pues con los que se contaba de tareas anteriores no se podía realizar la medición del experimento y obtener valores medibles pues el número máximo de nodos con los que se contaba era de cinco en todos los grafos. A continuación se muestran las líneas de código que fueron empleadas para generar los grafos con los que se trabajó durante la medición de los algoritmos seleccionados.

```
1 def Generar(nombre, num_archivos, cant_nodos_min, probabilidad):
2     G = nx.Graph()
3     for r in range(num_archivos):
4         print("Archivo", r)
5         for i in range(cant_nodos_min):
6             G.add_node(i)
7             for j in range(i + 1, cant_nodos_min):
8                 crear_vertice = rnd.randint(0, 100) < probabilidad
9                 print(i, j, crear_vertice)
10                if crear_vertice:
11                    G.add_edge(i, j, distancia=rnd.randint(10,15))
12                df = nx.to_pandas_adjacency(G, dtype=int, weight='distancia')
13                df.to_csv(nombre+str(r)+".csv", index = None, header=None )
14    return 0
15
16 #Generar("grafo", 5, 200, 25)
17
18 def Clicgenerar(nombre, num_archivos, cant_nodos_min, probabilidad):
19     G = nx.Graph()
20     for r in range(num_archivos):
21         print("Archivo", r)
22         for i in range(cant_nodos_min):
23             G.add_node(i)
24             for j in range(i + 1, cant_nodos_min):
25                 crear_vertice = rnd.randint(0, 100) < probabilidad
26                 print(i, j, crear_vertice)
27                 if crear_vertice:
28                     G.add_edge(i, j, distancia=rnd.randint(10,15))
29                 df = nx.to_pandas_adjacency(G, dtype=int, weight='distancia')
30                 df.to_csv(nombre+str(r)+".csv", index = None, header=None )
31     return 0
```

```

32 #Clicgenerar(" clicgrafo", 5, 40, 10)
33
34 def Digenerar(nombre, num_archivos, cant_nodos_min, probabilidad):
35     G = nx.DiGraph()
36     for r in range(num_archivos):
37         print("Archivo", r)
38         for i in range(cant_nodos_min):
39             G.add_node(i)
40             for j in range(i + 1, cant_nodos_min):
41                 crear_vertice = rnd.randint(0, 100) < probabilidad
42                 print(i, j, crear_vertice)
43                 if crear_vertice:
44                     G.add_edge(i,j, distancia=rnd.randint(10,15))
45             df = nx.to_pandas_adjacency(G, dtype=int, weight='distancia')
46             df.to_csv(nombre+str(r)+".csv", index = None, header=None )
47
48     return 0
49
50 #Digenerar(" digrafo", 5, 400, 25)

```

codigomadre.py

Se generaron cinco grafos: dirigidos y no dirigidos, ponderados, con la cantidad de nodos y arcos aleatorias, así como otro grupo de cinco grafos con una cantidad de nodos y arcos menor para algoritmos que así lo requerían.

cuatro

Para la selección del tipo de algoritmo se tuvo en cuenta que combinaran con los grafos generados y permitieran obtener valores medibles de ejecución en cada uno. Fueron probados todos y de ellos los mejores resultados de medición arrojaron fueron los siguientes:

- Betweenness centrality (en lo adelante BC)
- Max clique (en lo adelante MC)
- Greedy color (en lo adelante GC)
- Maximum flow (en lo adelante MF)
- Minimum spanning tree (en lo adelante MS)

1.1. Breve descripción de los algoritmos medidos y su código

El algoritmo BC nos brinda la medida de centralidad de un grafo basado en la trayectoria más corta, este devuelve un diccionario de nodos con la medida de centralidad [1]. La centralidad de la interrelación de un nodo es la suma de la fracción de las rutas más cortas de todos los pares que pasan por él [2]. Puede usarse en grafos dirigidos y no dirigidos.

El algoritmo MC devuelve el subgrafo más grande que encuentre, se recomienda su uso en grafos no dirigidos, es uno de los algoritmos que más tiempo demora en su ejecución.

Por su parte el algoritmo GC, lo que hace es colorear un nodo usando diferentes estrategias de coloración que se le pasan como un parámetro dentro de una función. Devuelve un diccionario con claves que representan nodos y valores que representan la coloración correspondiente [3]. Puede emplearse en grafos dirigidos y no dirigidos.

El algoritmo MF determina la ruta a través de la cual puede pasar el máximo flujo, de ahí que uno de los parámetros que requiere es la capacidad, y un su defecto la asume como infinita [4]. Se recomienda emplearlo en grafos dirigidos.

El algoritmo MS tree devuelve la conexión entre los nodos de modo que la unión entre ellos es un árbol no dirigido ponderado cuya suma de pesos de cada arco es la menor posible [5].

A continuación se muestra el fragmento de código desarrollado para la medición del tiempo por cada algoritmo:

```

1 def betweenness_centrality(name):
2     dr = pd.read_csv(name, header=None)
3     F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
4     tiempo=[]
5     for i in range(30):
6         tiempo_inicial = dt.datetime.now()
7         d= nx.betweenness_centrality(F) # aca estoy guardando el resultado del
8         #algoritmo y como no lo uso paranada me da
9         tiempo_final = dt.datetime.now()
10        tiempo_ejecucion = (tiempo_final - tiempo_inicial).total_seconds()
11        tiempo.append(tiempo_ejecucion)
12
13    media=nup.mean(tiempo)
14    desv=nup.std(tiempo)
15    mediana=nup.median(tiempo)
16    salvar=[]
17    salvar.append(media)
18    salvar.append(desv)
19    salvar.append(mediana)
20    df = pd.DataFrame(salvar)
21    df.to_csv("bet_"+name, index = None, header=None )
22    print("terminado betweenness")
23
23 def max_clique(name):
24     dr = pd.read_csv(name, header=None)
25     F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
26     tiempo=[]
27     for i in range(30):
28         tiempo_inicial = dt.datetime.now()
29         d= nx.make_max_clique_graph(F,create_using=None) # aca estoy guardando el
30         #resultado del algoritmo y como no lo uso paranada me da
31         tiempo_final = dt.datetime.now()
32         tiempo_ejecucion = (tiempo_final - tiempo_inicial).total_seconds()
33         tiempo.append(tiempo_ejecucion)
34
34    media=nup.mean(tiempo)
35    desv=nup.std(tiempo)
36    mediana=nup.median(tiempo)
37    salvar=[]
38    salvar.append(media)
39    salvar.append(desv)
```

```

40     salvar.append(mediana)
41     df = pd.DataFrame(salvar)
42     df.to_csv("clique_"+name, index = None, header=None )
43     print("terminado maxclique")
44
45
46 def greedy_color(name):
47     dr = pd.read_csv(name,header=None)
48     F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
49     tiempo=[]
50     for i in range(300):
51         tiempo_inicial = dt.datetime.now()
52         d= nx.greedy_color(F)
53         tiempo_final = dt.datetime.now()
54         tiempo_ejecucion = (tiempo_final - tiempo_inicial).total_seconds()
55         tiempo.append(tiempo_ejecucion)
56
57     media=nup.mean(tiempo)
58     desv=nup.std(tiempo)
59     mediana=nup.median(tiempo)
60     salvar=[]
61     salvar.append(media)
62     salvar.append(desv)
63     salvar.append(mediana)
64     df = pd.DataFrame(salvar)
65     df.to_csv("greed_."+name, index = None, header=None )
66     print("Terminado greedy")
67
68 def maximum_flow (name):
69     dr = pd.read_csv(name,header=None)
70     F = nx.from_pandas_adjacency(dr, create_using= nx.DiGraph())
71     tiempo=[]
72     for i in range(30):
73         tiempo_inicial = dt.datetime.now()
74         d= nx.maximum_flow (F, 4, 2) # aca estoy guardando el resultado del
75         # algoritmo y como no lo uso paranada me da
76         tiempo_final = dt.datetime.now()
77         tiempo_ejecucion = (tiempo_final - tiempo_inicial).total_seconds()
78         tiempo.append(tiempo_ejecucion)
79
80     media=nup.mean(tiempo)
81     desv=nup.std(tiempo)
82     mediana=nup.median(tiempo)
83     salvar=[]
84     salvar.append(media)
85     salvar.append(desv)
86     salvar.append(mediana)
87     df = pd.DataFrame(salvar)
88     df.to_csv("maxflow_."+name, index = None, header=None )
89     print ("terminado maximun flow")
90
91 def minimum_spanning_tree(name):
92     dr = pd.read_csv(name,header=None)
93     F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
94     tiempo=[]
95     for i in range(30):
96         tiempo_inicial = dt.datetime.now()
97         d= nx.minimum_spanning_tree(F) # aca estoy guardando el resultado del

```

```

97     algoritmo y como no lo uso paranada me da
98         tiempo_final = dt.datetime.now()
99         tiempo_ejecucion = (tiempo_final - tiempo_inicial).total_seconds()
100        tiempo.append(tiempo_ejecucion)
101
102    media=nup.mean(tiempo)
103    desv=nup.std(tiempo)
104    mediana=nup.median(tiempo)
105    salvar=[]
106    salvar.append(media)
107    salvar.append(desv)
108    salvar.append(mediana)
109    df = pd.DataFrame(salvar)
110    df.to_csv("spanning_tree_"+name, index = None, header=None )
111    print ("terminado spanning-tree ")

```

codigomadre.py

CORRECTOS = False

if corretos:

```

1  #for r in range(5):
2      betweenness_centrality("grafo"+str(r)+".csv") # OK
3      max_clique("clicgrafo"+str(r)+".csv")# OK
4      ## all_shortest_paths("grafo"+str(r)+".csv") #No funciona
5      ## topological("digrafo"+str(r)+".csv") # No funciona
6      ## greedy_color("digrafo"+str(r)+".csv")#OK
7      maximum_flow("digrafo"+str(r)+".csv") #OK
8      ## min_weighted_vertex_cover("grafo"+str(r)+".csv")#no esta en networkx
9      minimum_spanning_tree ("grafo"+str(r)+".csv") # OK
10     ## strongly_connected_components ("digrafo"+str(r)+".csv")#No funciona
11     ## treewidth_min_degree ("grafo"+str(r)+".csv")# no esta mi networkx
12     ## min_maximal_matching ("grafo"+str(r)+".csv") no esta en mi networkx

```

codigomadre.py

from sys import argc

2. Resultados

if 'debug' in argc:

Se realizó la medición de los tiempos de los algoritmos teniendo en cuenta la media y la desviación estándar en cada caso.

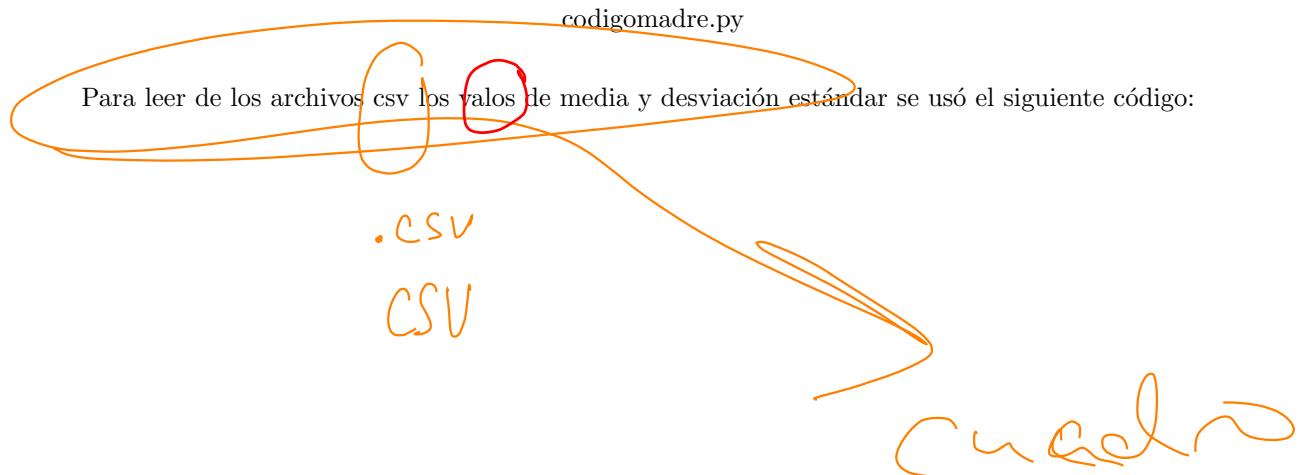
Para obtener la cantidad de nodos y arcos de cada grafo empleado se desarrolló el siguiente código:

```

1 def GrafosElementos(cant):
2     grafos={
3         "clicgrafoN":[] ,
4         "digrafoN":[] ,
5         "grafoN":[] ,
6         "clicgrafoE":[] ,
7         "digrafoE":[] ,
8         "grafoE":[] ,
9     }
10    for i in range(cant):
11        dr = pd.read_csv("clicgrafo"+str(i)+".csv",header=None)
12        F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
13        grafos[ "clicgrafoN"].append(F.number_of_nodes())
14
15        dr = pd.read_csv("digrafo"+str(i)+".csv",header=None)
16        F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
17        grafos[ "digrafoN"].append(F.number_of_nodes())
18
19        dr = pd.read_csv("grafo"+str(i)+".csv",header=None)
20        F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
21        grafos[ "grafoN"].append(F.number_of_nodes())
22
23        dr = pd.read_csv("clicgrafo"+str(i)+".csv",header=None)
24        F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
25        grafos[ "clicgrafoE"].append(F.number_of_edges())
26
27        dr = pd.read_csv("digrafo"+str(i)+".csv",header=None)
28        F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
29        grafos[ "digrafoE"].append(F.number_of_edges())
30
31        dr = pd.read_csv("grafo"+str(i)+".csv",header=None)
32        F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
33        grafos[ "grafoE"].append(F.number_of_edges())
34
35    df = pd.DataFrame(grafos)
36    df.to_csv("cantNE.csv")
37
38 #GrafosElementos(5)

```

Para leer de los archivos csv los valores de media y desviación estándar se usó el siguiente código:



```

1 def lee_valores(cant):
2
3     algorit={}
4
5         "Media_A1bc":[] ,
6         "Media_A2mc":[] ,
7         "Media_A3gc":[] ,
8         "Media_A4mf":[] ,
9         "Media_A5mst":[] ,
10        "Desvi_A1bc":[] ,
11        "Desvi_A2mc":[] ,
12        "Desvi_A3gc":[] ,
13        "Desvi_A4mf":[] ,
14        "Desvi_A5mst":[]
15    }
16
17    for i in range(cant):
18
19        dr = pd.read_csv("bet_grafo"+str(i)+".csv",header=None, )
20        # print(dr[0][0])
21        algorit [ "Media_A1bc" ].append(dr[0][0])
22
23        dr = pd.read_csv("clique_clicgrafo"+str(i)+".csv",header=None)
24
25        algorit [ "Media_A2mc" ].append(dr[0][0])
26
27        dr = pd.read_csv("greed_digrafo"+str(i)+".csv",header=None)
28        algorit [ "Media_A3gc" ].append(dr[0][0])
29
30        dr = pd.read_csv("maxflow_digrafo"+str(i)+".csv",header=None)
31        algorit [ "Media_A4mf" ].append(dr[0][0])
32
33        dr = pd.read_csv("spanning_tree_grafo"+str(i)+".csv",header=None)
34        algorit [ "Media_A5mst" ].append(dr[0][0])
35
36        dr = pd.read_csv("bet_grafo"+str(i)+".csv",header=None)
37        # print(dr[0][1])
38        algorit [ "Desvi_A1bc" ].append(dr[0][1])
39
40        dr = pd.read_csv("clique_clicgrafo"+str(i)+".csv",header=None)
41        algorit [ "Desvi_A2mc" ].append(dr[0][1])
42
43        dr = pd.read_csv("greed_digrafo"+str(i)+".csv",header=None)
44        algorit [ "Desvi_A3gc" ].append(dr[0][1])
45
46        dr = pd.read_csv("maxflow_digrafo"+str(i)+".csv",header=None)
47        algorit [ "Desvi_A4mf" ].append(dr[0][1])
48
49        dr = pd.read_csv("spanning_tree_grafo"+str(i)+".csv",header=None)
50        algorit [ "Desvi_A5mst" ].append(dr[0][1])
51
52        df = pd.DataFrame(algorit)
53        df.to_csv("algori_med.csv" )
54
55    lee_valores(5)

```

codigomadre.py

Luego se realizaron los histogramas que reflejan la variación del valor de la media por algoritmo empleado. Seguidamente se muestra un fragmento del código y los histogramas por algoritmos se muestran en la figura 1 de la página 9. En la gráfica se observa que en la mayoría de los algoritmos existe tendencia a ubicarse hacia los valores extremos, excepto en el algoritmo GC, que se aprecia una cantidad de valores igual en cada barra.

```
1 import random as rnd
2 import networkx as nx
3 import matplotlib.pyplot as plt
4 import numpy as nup
5 import datetime as dt
6 import pandas as pd
7 import statistics as stats
8
9 data = pd.read_csv("1.csv")
10 plt.figure(figsize=(12, 6))
11
12 plt.subplot(321)
13 x = plt.hist(data["Media_A1bc"], 5, facecolor='blue', alpha=0.5)
14
15 plt.subplot(322)
16 x = plt.hist(data["Media_A2mc"], 5, facecolor='red', alpha=0.5)
17 print(x)
18 plt.subplot(323)
19 x = plt.hist(data["Media_A3gc"], 5, facecolor='green', alpha=0.5)
20
21 plt.subplot(324)
22 x = plt.hist(data["Media_A4mf"], 5, facecolor='yellow', alpha=0.5)
23
24 plt.subplot(325)
25 x = plt.hist(data["Media_A5mst"], 5, facecolor='pink', alpha=0.5)
26
27
28 plt.subplots_adjust(left=0.15)
29 plt.savefig("Imagenes/Fig01.eps")
30 plt.show()
```

Histogramas.py

Para concluir se graficaron en un diagrama de dispersión el tiempo medio que se demora cada algoritmo en función de la cantidad de nodos y arcos respectivamente.

En la figura 2 de la página 10, se muestra la relación entre la media de cada algoritmo y la cantidad de nodos que se emplearon por grafo. De esta se puede apreciar que de los algoritmos que se corrieron con 200 nodos que fueron el MS y el BC, el que presenta valores más elevados de la media en cada una de las veces que se corrió el algoritmo fue el BC. De los que se corrieron para 400 nodos que fueron los algoritmos GC y MF, el que posee mayores valores de la media fue el MF. Finalmente se corrió con una cantidad significativamente menor el algoritmo MC, pues es el que más tiempo consumía en la medición, arrojando lógicamente menores valores en la media, sin embargo a pesar de tener solo 50 nodos al compararlo con el algoritmo MS que se corrió con 400 nodos, los valores en

centrals

trim

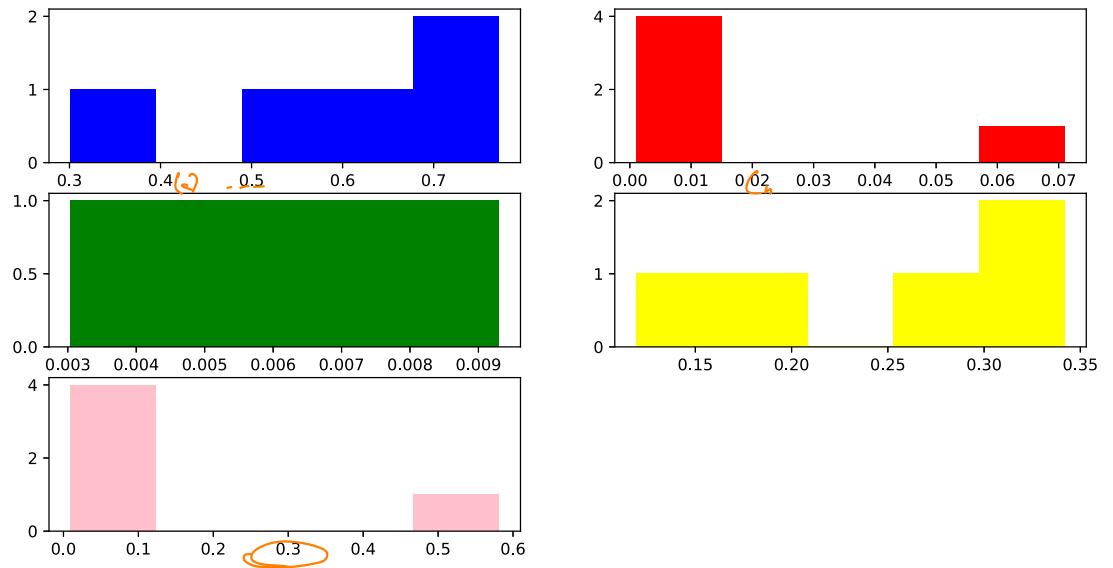


Figura 1: Histogramas de Algoritmos (cant. valores vs media): azul-BC, rojo-MC, verde-GC, amarillo-MF y rosado-MS

la media que arroja son similares, lo que demuestra que el MC es mucho más lento en su tiempo de ejecución que el MS. De lo anterior se destaca como oportunidad de mejora homogenizar la cantidad de nodos que se empleen en los grafos para tener una comparación más cercana a la realidad.

A continuación se muestra el código donde aparece identificado la figura y color a cuál algoritmo corresponde.

```

1 def Scatter_Nodos(nombre):
2     colors = ["y", "r", "b", "black", "g"]
3     data = pd.read_csv(nombre)
4     markers=["d", "s", "*", "8", "x"]
5
6     x=data["Media_A1bc"]
7     y=data["grafoN"]
8     area=300
9     plt.scatter(x, y, s=area, c=colors, alpha=0.5, marker="d")
10    print(y)
11
12    x=data["Media_A2mc"]
13    y=data["clicgrafoN"]
14    area=300
15    plt.scatter(x, y, s=area, c=colors, alpha=0.5, marker="s")
16    print(y)
17

```

```

18 x=data[ "Media_A3gc" ]
19 y=data[ "digrafoN" ]
20 area=300
21 plt.scatter(x, y, s=area, c=colors, alpha=0.5, marker="*")
22 print(y)
23
24 x=data[ "Media_A4mf" ]
25 y=data[ "digrafoN" ]
26 area=300
27 plt.scatter(x, y, s=area, c=colors, alpha=0.5, marker="8")
28 print(y)
29
30 x=data[ "Media_A5mst" ]
31 y=data[ "grafoN" ]
32 area=300
33 plt.scatter(x, y, s=area, c=colors, alpha=0.5, marker="x")
34 print(y)
35
36 plt.savefig("Imagenes/Fig2.eps")
37 plt.show()
38
39 Scatter_Nodos("1.csv")
40

```

Scatterplot.py

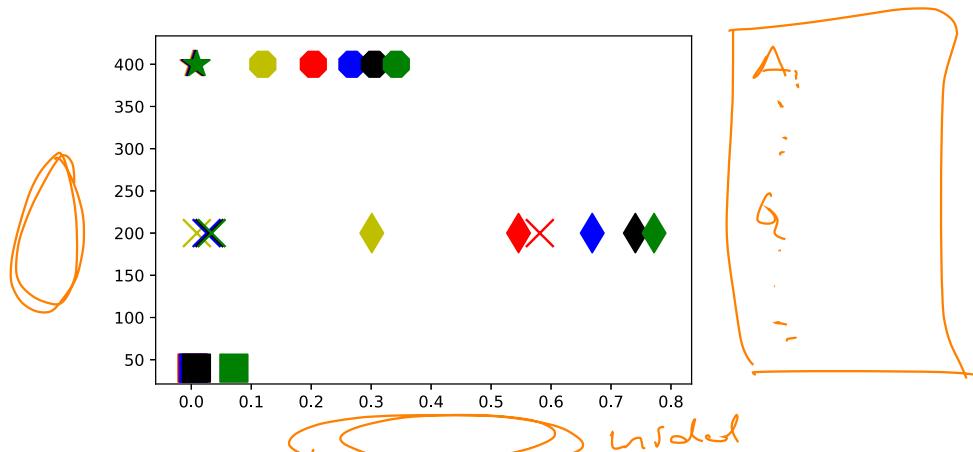


Figura 2: Diagrama de dispersión (cant. nodos vs media-algoritmo)

En la figura 3 de la página 12 se muestra la relación entre la media de cada algoritmo y la cantidad de arcos que se emplearon por grafo. En esta se aprecia que como tendencia general en los algoritmos medidos, al aumentar el número de arcos aumenta la media, el que más aumenta en su media es el algoritmo BC, en el algoritmo GC el valor de la media permanece casi constante a pesar del aumento de arcos, lo cual tiene sentido por el modo en que opera el mismo, el algoritmo MS actúa de manera similar. Para el MC es casi despreciable la cantidad de arcos, debido a la demora del mismo se decidió disminuirlos.

Seguidamente se muestra el fragmento de código:

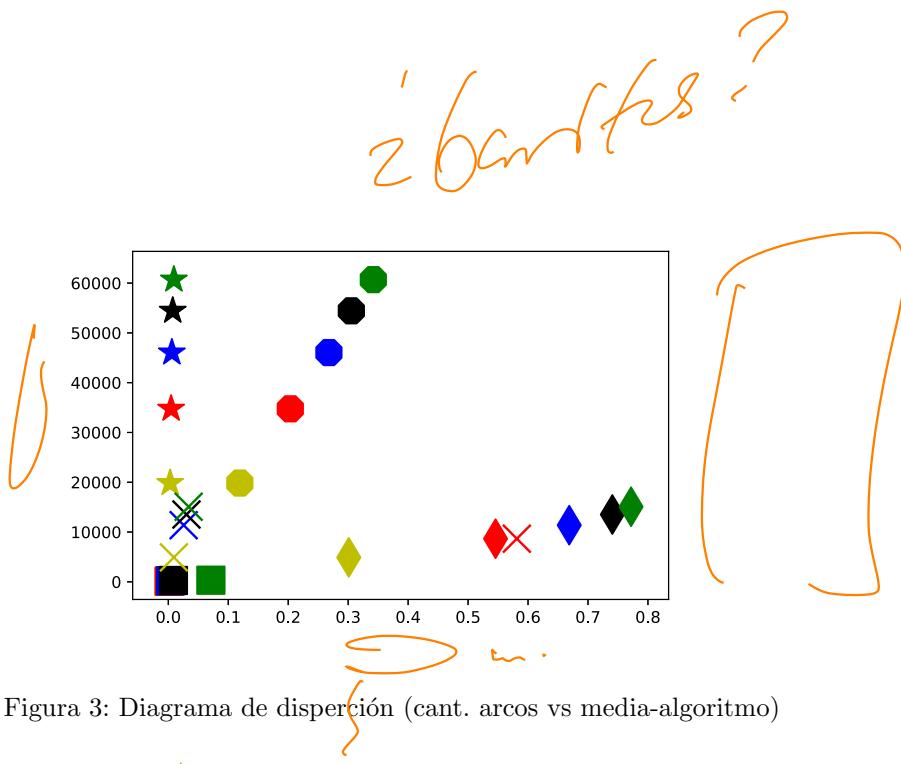
```
1 def Scatter_Edges (nombre):
2     colors = [ "y" , "r" , "b" , "black" , "g" ]
3     data = pd.read_csv(nombre)
4     markers=[ "d" , "s" , "*" , "8" , "x" ]
5
6     x=data[ "Media_A1bc" ]
7     y=data[ "grafoE" ]
8     area=300
9     plt.scatter(x, y, s=area, c=colors, alpha=0.5, marker="d")
10    print(y)
11
12    x=data[ "Media_A2mc" ]
13    y=data[ "clicgrafoE" ]
14    area=300
15    plt.scatter(x, y, s=area, c=colors, alpha=0.5, marker="s")
16    print(y)
17
18    x=data[ "Media_A3gc" ]
19    y=data[ "digrafoE" ]
20    area=300
21    plt.scatter(x, y, s=area, c=colors, alpha=0.5, marker="*")
22    print(y)
23
24    x=data[ "Media_A4mf" ]
25    y=data[ "digrafoE" ]
26    area=300
27    plt.scatter(x, y, s=area, c=colors, alpha=0.5, marker="8")
28    print(y)
29
30    x=data[ "Media_A5mst" ]
31    y=data[ "grafoE" ]
32    area=300
33    plt.scatter(x, y, s=area, c=colors, alpha=0.5, marker="x")
34    print(y)
35
36
37    plt.savefig("Imagenes/Fig3.eps")
38    plt.show()
39
40 Scatter_Edges("1.csv")
```

Scatterplot.py

Referencias

- [1] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.centralization.betweenness_centrality.html, . Accessed: 2019-03-10.
- [2] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/_modules/networkx/algorithms/centrality/betweenness.html.betweenness_centrality. Accessed: 2019-03-18.

year = 2019,¹¹



- [3] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/_modules/networkx/algorithms/coloring/greedy_coloring.html#greedy_color, . Accessed: 2019-03-17.
- [4] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.flow.maximum_flow.html, . Accessed: 2019-03-17.
- [5] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.mst.minimum_spanning_tree.html, . Accessed: 2019-03-17.

Tarea No.3: Rectificada

Dayli Machado (5275)

3 de junio de 2019

1. Grafos y algoritmos empleados

Para la realización de esta tarea se crearon nuevos grafos pues con los que se contaba de tareas anteriores no se podía realizar la medición del experimento y obtener valores medibles pues el número máximo de nodos con los que se contaba era de cinco en todos los grafos. A continuación se muestran las líneas de código que fueron empleadas para generar los grafos con los que se trabajó durante la medición de los algoritmos seleccionados.

```
1 def Generar(nombre, num_archivos, cant_nodos_min, probabilidad):
2     G = nx.Graph()
3     for r in range(num_archivos):
4         print("Archivo", r)
5         for i in range(cant_nodos_min):
6             G.add_node(i)
7             for j in range(i + 1, cant_nodos_min):
8                 crear_vertice = rnd.randint(0, 100) < probabilidad
9                 print(i, j, crear_vertice)
10                if crear_vertice:
11                    G.add_edge(i, j, distancia=rnd.randint(10,15))
12                df = nx.to_pandas_adjacency(G, dtype=int, weight='distancia')
13                df.to_csv(nombre+str(r)+".csv", index = None, header=None )
14        return 0
15
16 #Generar("grafo", 5, 200, 25)
17
18 def Clicgenerar(nombre, num_archivos, cant_nodos_min, probabilidad):
19     G = nx.Graph()
20     for r in range(num_archivos):
21         print("Archivo", r)
22         for i in range(cant_nodos_min):
23             G.add_node(i)
24             for j in range(i + 1, cant_nodos_min):
25                 crear_vertice = rnd.randint(0, 100) < probabilidad
26                 print(i, j, crear_vertice)
27                 if crear_vertice:
28                     G.add_edge(i, j, distancia=rnd.randint(10,15))
29                 df = nx.to_pandas_adjacency(G, dtype=int, weight='distancia')
30                 df.to_csv(nombre+str(r)+".csv", index = None, header=None )
31         return 0
32
33 #Clicgenerar("clicgrafo", 5, 40, 10)
34
35 def Digenesar(nombre, num_archivos, cant_nodos_min, probabilidad):
36     G = nx.DiGraph()
37     for r in range(num_archivos):
38         print("Archivo", r)
```

```

39     for i in range(cant_nodos_min):
40         G.add_node(i)
41         for j in range(i + 1, cant_nodos_min):
42             crear_vertice = rnd.randint(0, 100) < probabilidad
43             print(i, j, crear_vertice)
44             if crear_vertice:
45                 G.add_edge(i, j, distancia=rnd.randint(10,15))
46             df = nx.to_pandas_adjacency(G, dtype=int, weight='distancia')
47             df.to_csv(nombre+str(r)+".csv", index = None, header=None )
48     return 0
49
50 #Digenerar("digrafo", 5, 400, 25)

```

codigomadre.py

Se generaron cinco grafos: dirigidos y no dirigidos, ponderados, con la cantidad de nodos y arcos aleatorias, así como otro grupo de cinco grafos con una cantidad de nodos y arcos menor para algoritmos que así lo requerían.

Para la selección del tipo de algoritmo se tuvo en cuenta que combinaran con los grafos generados y permitieran obtener valores medibles de ejecución en cada uno. Fueron probados todos y de ellos los que mejores resultados de medición arrojaron fueron los siguientes:

- Centralidad de intermediación (en lo adelante BC según sus siglas en inglés *Betweenness centrality*)
- *Max clique* (en lo adelante MC)
- *Greedy* color (en lo adelante GC)
- Flujo Máximo (en lo adelante MF, por sus siglas en inglés *Maximum flow*)
- Árbol de expansión mínima (en lo adelante MS, por sus siglas en inglés Minimum spanning tree)

1.1. Breve descripción de los algoritmos medidos y su código

El algoritmo BC nos brinda la medida de centralidad de un grafo basado en la trayectoria más corta, este devuelve un diccionario de nodos con la medida de centralidad [de NetworkX(a)]. La centralidad de la interrelación de un nodo es la suma de la fracción de las rutas más cortas de todos los pares que pasan por él [de NetworkX(b)]. Puede usarse en grafos dirigidos y no dirigidos.

El algoritmo MC devuelve el subgrafo más grande que encuentre, se recomienda su uso en grafos no dirigidos, es uno de los algoritmos que más tiempo demora en su ejecución.

Por su parte el algoritmo GC, lo que hace es colorear un nodo usando diferentes estrategias de colo-ración que se le pasan como un parámetro dentro de una función. Devuelve un diccionario con claves que representan nodos y valores que representan la coloración correspondiente [NetworkX(a)]. Puede emplearse en grafos dirigidos y no dirigidos.

El algoritmo MF determina la ruta a través de la cual puede pasar el máximo flujo, de ahí que uno de los parámetros que requiere es la capacidad, y un su defecto la asume como infinita [NetworkX(b)]. Se recomienda emplearlo en grafos dirigidos.

El algoritmo MS tree devuelve la conexión entre los nodos de modo que la unión entre ellos es un árbol no dirigido ponderado cuya suma de pesos de cada arco es la menor posible [NetworkX(c)].

A continuación se muestra el fragmento de código desarrollado para la medición del tiempo por cada algoritmo:

```

1 def betweenness_centrality(name):
2     dr = pd.read_csv(name, header=None)
3     F = nx.from_pandas_adjacency(dr, create_using=nx.Graph())
4     tiempo=[]
5     for i in range(30):
6         tiempo_inicial = dt.datetime.now()
7         d= nx.betweenness_centrality(F) # aca estoy guardando el resultado del
8         #algoritmo y como no lo uso paranada me da
9         tiempo_final = dt.datetime.now()
10        tiempo_ejecucion = (tiempo_final - tiempo_inicial).total_seconds()
11        tiempo.append(tiempo_ejecucion)
12
13    media=nup.mean(tiempo)
14    desv=nup.std(tiempo)
15    mediana=nup.median(tiempo)
16    salvar=[]
17    salvar.append(media)
18    salvar.append(desv)
19    salvar.append(mediana)
20    df = pd.DataFrame(salvar)
21    df.to_csv("bet_"+name, index = None,header=None )
22    print("terminado betweeness")
23
24 def max_clique(name):
25     dr = pd.read_csv(name, header=None)
26     F = nx.from_pandas_adjacency(dr, create_using=nx.Graph())
27     tiempo=[]
28     for i in range(30):
29         tiempo_inicial = dt.datetime.now()
30         d= nx.make_max_clique_graph(F,create_using=None) # aca estoy guardando el
31         #resultado del algoritmo y como no lo uso paranada me da
32         tiempo_final = dt.datetime.now()
33         tiempo_ejecucion = (tiempo_final - tiempo_inicial).total_seconds()
34         tiempo.append(tiempo_ejecucion)
35
36    media=nup.mean(tiempo)
37    desv=nup.std(tiempo)
38    mediana=nup.median(tiempo)
39    salvar=[]
40    salvar.append(media)
41    salvar.append(desv)
42    salvar.append(mediana)
43    df = pd.DataFrame(salvar)
44    df.to_csv("clique_"+name, index = None,header=None )
45    print("terminado maxclique")
46
47 def greedy_color(name):
48     dr = pd.read_csv(name, header=None)
49     F = nx.from_pandas_adjacency(dr, create_using=nx.Graph())
50     tiempo=[]
51     for i in range(300):
52         tiempo_inicial = dt.datetime.now()
53         d= nx.greedy_color(F)
54         tiempo_final = dt.datetime.now()
55         tiempo_ejecucion = (tiempo_final - tiempo_inicial).total_seconds()
56         tiempo.append(tiempo_ejecucion)
57
58    media=nup.mean(tiempo)
59    desv=nup.std(tiempo)
60    mediana=nup.median(tiempo)
61    salvar=[]

```

```

61     salvar.append(media)
62     salvar.append(desv)
63     salvar.append(mediana)
64     df = pd.DataFrame(salvar)
65     df.to_csv("greed_"+name, index = None, header=None )
66     print("Terminado greedy")
67
68 def maximum_flow (name):
69     dr = pd.read_csv(name, header=None)
70     F = nx.from_pandas_adjacency(dr, create_using= nx.DiGraph())
71     tiempo=[]
72     for i in range(30):
73         tiempo_inicial = dt.datetime.now()
74         d= nx.maximum_flow (F, 4, 2) # aca estoy guardando el resultado del algoritmo
75         y como no lo uso paranada me da
76         tiempo_final = dt.datetime.now()
77         tiempo_ejecucion = (tiempo_final - tiempo_inicial).total_seconds()
78         tiempo.append(tiempo_ejecucion)
79
80     media=nup.mean(tiempo)
81     desv=nup.std(tiempo)
82     mediana=nup.median(tiempo)
83     salvar=[]
84     salvar.append(media)
85     salvar.append(desv)
86     salvar.append(mediana)
87     df = pd.DataFrame(salvar)
88     df.to_csv("maxflow_"+name, index = None, header=None )
89     print ("terminado maximun flow")
90
91 def minimum_spanning_tree(name):
92     dr = pd.read_csv(name, header=None)
93     F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
94     tiempo=[]
95     for i in range(30):
96         tiempo_inicial = dt.datetime.now()
97         d= nx.minimum_spanning_tree(F) # aca estoy guardando el resultado del
98         algoritmo y como no lo uso paranada me da
99         tiempo_final = dt.datetime.now()
100        tiempo_ejecucion = (tiempo_final - tiempo_inicial).total_seconds()
101        tiempo.append(tiempo_ejecucion)
102
103    media=nup.mean(tiempo)
104    desv=nup.std(tiempo)
105    mediana=nup.median(tiempo)
106    salvar=[]
107    salvar.append(media)
108    salvar.append(desv)
109    salvar.append(mediana)
110    df = pd.DataFrame(salvar)
111    df.to_csv("spanning_tree_"+name, index = None, header=None )
112    print ("terminado spanning_tree ")

```

codigomadre.py

2. Resultados

Se realizó la medición de los tiempos de los algoritmos teniendo en cuenta la media y la desviación estándar en cada caso.

Para obtener la cantidad de nodos y aristas de cada grafo empleado se desarrolló el siguiente código:

```
1 def GrafosElementos(cant):
2     grafos={}
3         "clicgrafoN":[] ,
4         "digrafoN":[] ,
5         "grafoN":[] ,
6         "clicgrafoE":[] ,
7         "digrafoE":[] ,
8         "grafoE":[] ,
9     }
10    for i in range(cant):
11        dr = pd.read_csv("clicgrafo"+str(i)+".csv",header=None)
12        F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
13        grafos[ "clicgrafoN"].append(F.number_of_nodes())
14
15        dr = pd.read_csv("digrafo"+str(i)+".csv",header=None)
16        F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
17        grafos[ "digrafoN"].append(F.number_of_nodes())
18
19        dr = pd.read_csv("grafo"+str(i)+".csv",header=None)
20        F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
21        grafos[ "grafoN"].append(F.number_of_nodes())
22
23        dr = pd.read_csv("clicgrafo"+str(i)+".csv",header=None)
24        F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
25        grafos[ "clicgrafoE"].append(F.number_of_edges())
26
27        dr = pd.read_csv("digrafo"+str(i)+".csv",header=None)
28        F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
29        grafos[ "digrafoE"].append(F.number_of_edges())
30
31        dr = pd.read_csv("grafo"+str(i)+".csv",header=None)
32        F = nx.from_pandas_adjacency(dr, create_using= nx.Graph())
33        grafos[ "grafoE"].append(F.number_of_edges())
34
35    df = pd.DataFrame(grafos)
36    df.to_csv("cantNE.csv")
37
38 #GrafosElementos(5)
```

codigomadre.py

Para leer de los archivos .csv los valores de media y desviación estándar se usó el siguiente código:

```
1 def lee_valores(cant):
2
3     algorit={
4
5         "Media_A1bc":[] ,
6         "Media_A2mc":[] ,
7         "Media_A3gc":[] ,
8         "Media_A4mf":[] ,
9         "Media_A5mst":[] ,
10        "Desvi_A1bc":[] ,
11        "Desvi_A2mc":[] ,
12        "Desvi_A3gc":[] ,
13        "Desvi_A4mf":[] ,
```

```

14     "Desvi_A5mst": []
15 }
16
17 for i in range(cant):
18
19     dr = pd.read_csv("bet_grafo"+str(i)+".csv", header=None, )
20     print(dr[0][0])
21     algorit [ "Media_A1bc" ].append(dr[0][0])
22
23     dr = pd.read_csv("clique_clicgrafo"+str(i)+".csv", header=None)
24
25     algorit ["Media_A2mc"].append(dr[0][0])
26
27     dr = pd.read_csv("greed_digrafo"+str(i)+".csv", header=None)
28     algorit ["Media_A3gc"].append(dr[0][0])
29
30     dr = pd.read_csv("maxflow_digrafo"+str(i)+".csv", header=None)
31     algorit ["Media_A4mf"].append(dr[0][0])
32
33     dr = pd.read_csv("spanning_tree_grafo"+str(i)+".csv", header=None)
34     algorit ["Media_A5mst"].append(dr[0][0])
35
36     dr = pd.read_csv("bet_grafo"+str(i)+".csv", header=None)
37     print(dr[0][1])
38     algorit [ "Desvi_A1bc" ].append(dr[0][1])
39
40     dr = pd.read_csv("clique_clicgrafo"+str(i)+".csv", header=None)
41     algorit ["Desvi_A2mc"].append(dr[0][1])
42
43     dr = pd.read_csv("greed_digrafo"+str(i)+".csv", header=None)
44     algorit ["Desvi_A3gc"].append(dr[0][1])
45
46     dr = pd.read_csv("maxflow_digrafo"+str(i)+".csv", header=None)
47     algorit ["Desvi_A4mf"].append(dr[0][1])
48
49     dr = pd.read_csv("spanning_tree_grafo"+str(i)+".csv", header=None)
50     algorit ["Desvi_A5mst"].append(dr[0][1])
51
52 df = pd.DataFrame(algorit)
53 df.to_csv("algori_med.csv" )
54
55
56 lee_valores(5)

```

codigomadre.py

Luego se realizaron los histogramas que reflejan la variación del valor de la media por algoritmo empleado. Seguidamente se muestra un fragmento del código y los histogramas por algoritmos se muestran en la figura 1 de la página 8. En la gráfica se observa que en la mayoría de los algoritmos existe tendencia a ubicarse hacia los valores extremos, excepto en el algoritmo GC, que se aprecia una cantidad de valores igual en cada barra.

```

1 import random as rnd
2 import networkx as nx
3 import matplotlib.pyplot as plt
4 import numpy as nup
5 import datetime as dt
6 import pandas as pd
7 import statistics as stats
8
9 data = pd.read_csv("1.csv")
10
11
12

```

```

13 plt.hist(nup.log1p(data["Media_A1bc"]), 5, facecolor='blue', alpha=0.5, edgecolor = 'black', linewidth=1)
14 plt.ylabel('Frecuencia de ocurrencia')
15 plt.xlabel('Tiempo de Ejecucion')
16 plt.savefig("Imagenes/Histograma1.eps",bbox_inches='tight')
17 plt.show()
18
19
20 plt.hist(nup.log1p(data["Media_A2mc"]), 5, facecolor='red', alpha=0.5, edgecolor = 'black', linewidth=1)
21 plt.ylabel('Frecuencia de ocurrencia')
22 plt.xlabel('Tiempo de Ejecucion')
23 plt.savefig("Imagenes/Histograma2.eps",bbox_inches='tight')
24 plt.show()
25
26 plt.hist(nup.log1p(data["Media_A3gc"]), 5, facecolor='green', alpha=0.5, edgecolor = 'black', linewidth=1)
27 plt.ylabel('Frecuencia de ocurrencia')
28 plt.xlabel('Tiempo de Ejecucion')
29 plt.savefig("Imagenes/Histograma3.eps",bbox_inches='tight')
30 plt.show()

```

Histogramas.py

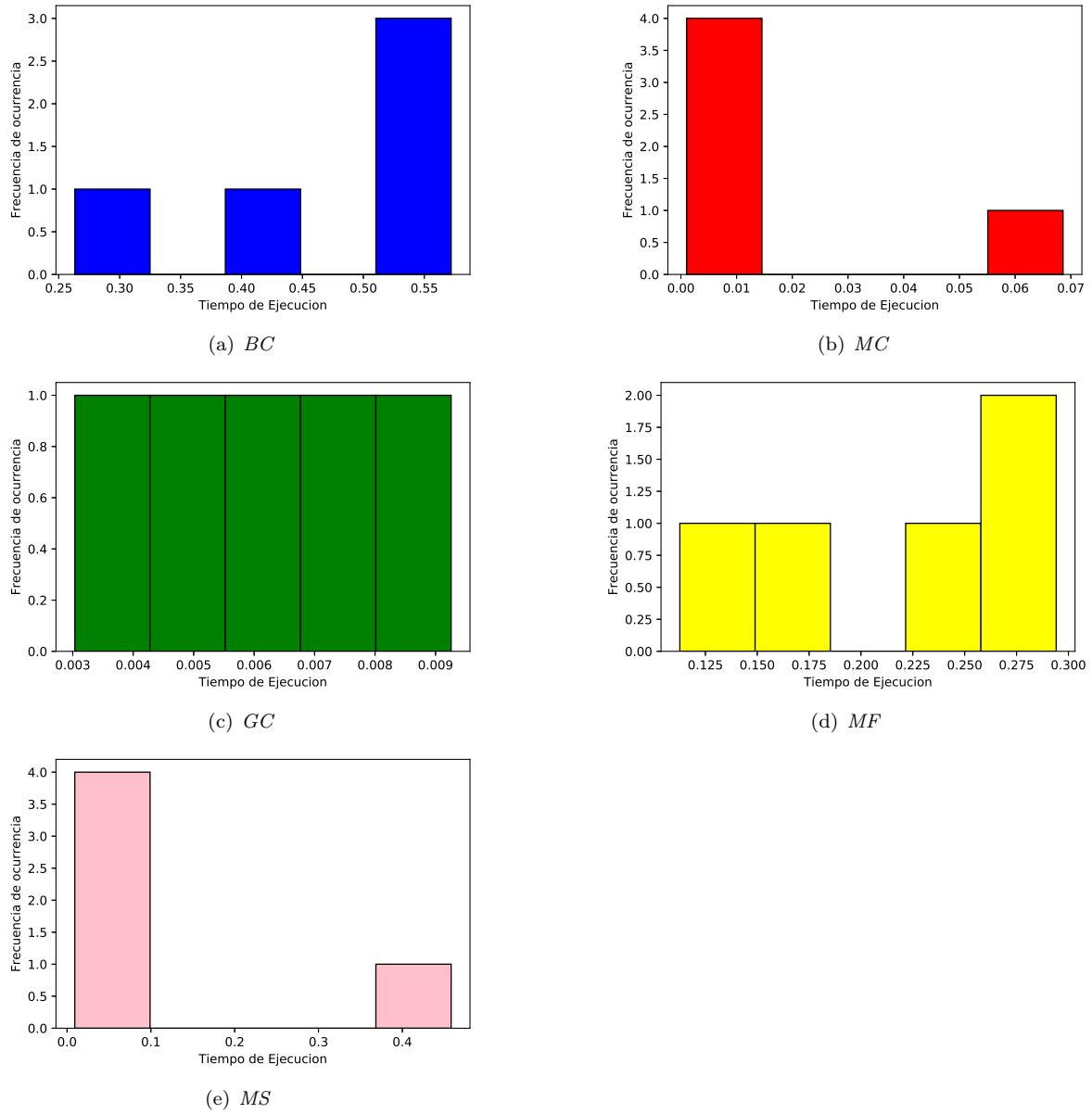


Figura 1: Histograma de cada uno de los cinco algoritmo con los cinco grafos generados.

Para concluir se graficaron en un diagrama de dispersión el tiempo medio que se demora cada algoritmo en función de la cantidad de nodos y aristas respectivamente.

En la figura 2 de la página 10, se muestra la relación entre la media de cada algoritmo y la cantidad de nodos que se emplearon por grafo. De esta se puede apreciar que de los algoritmos que se corrieron con 200 nodos que fueron el MS y el BC, el que presenta valores más elevados de la media en cada una de las veces que se corrió el algoritmo fue el BC. De los que se corrieron para 400 nodos que fueron los algoritmos GC y MF, el que posee mayores valores de la media fue el MF. Finalmente se corrió con una cantidad menor el algoritmo MC, pues es el que más tiempo consumía en la medición, arrojando lógicamente menores valores en la media, sin embargo a pesar de tener solo 50 nodos al compararlo con el algoritmo MS que se corrió con 400 nodos, los valores en la media que arroja son similares, lo que demuestra que el MC es mucho más lento en su tiempo de ejecución que el MS. De lo anterior se destaca como oportunidad de mejora homogenizar la cantidad de nodos que se empleen en los grafos para tener una comparación más cercana a la realidad.

A continuación se muestra el código donde aparece identificado la figura y color a cuál algoritmo corresponde.

```

1 import matplotlib.patches as mpatches
2
3 data = pd.read_csv("1.csv")
4
5 figure, axes = plt.subplots(figsize=(10, 10))
6 x=data["Media_A1bc"]
7 y=data["grafoN"]
8 xerr=data["Desvi_A1bc"]
9
10 axes.errorbar(x, y, xerr=xerr, fmt='D', color="#932525", alpha=1, label="BC")
11 print(y)
12
13 x=data["Media_A2mc"]
14 y=data["clicgrafoN"]
15 xerr=data["Desvi_A2mc"]
16
17 axes.errorbar(x, y, xerr=xerr, fmt='s', color="#129f10", alpha=1, label="MC")
18 print(y)
19
20 x=data["Media_A3gc"]
21 y=data["digrafoN"]
22 xerr=data["Desvi_A3gc"]
23
24 axes.errorbar(x, y, xerr=xerr, fmt='8', color="#093ea8", alpha=1, label="GC")
25 print(y)
26
27 x=data["Media_A4mf"]
28 y=data["digrafoN"]
29 xerr=data["Desvi_A4mf"]
30
31 axes.errorbar(x, y, xerr=xerr, fmt='>', color="#adcb18", alpha=1, label="MF")
32 print(y)
33
34 x=data["Media_A5mst"]
35 y=data["grafoN"]
36 xerr=data["Desvi_A5mst"]
37
38 axes.errorbar(x, y, xerr=xerr, fmt='o', color="#ee9110", alpha=1, label="MST")
39 print(y)

```

Scatterplot.py

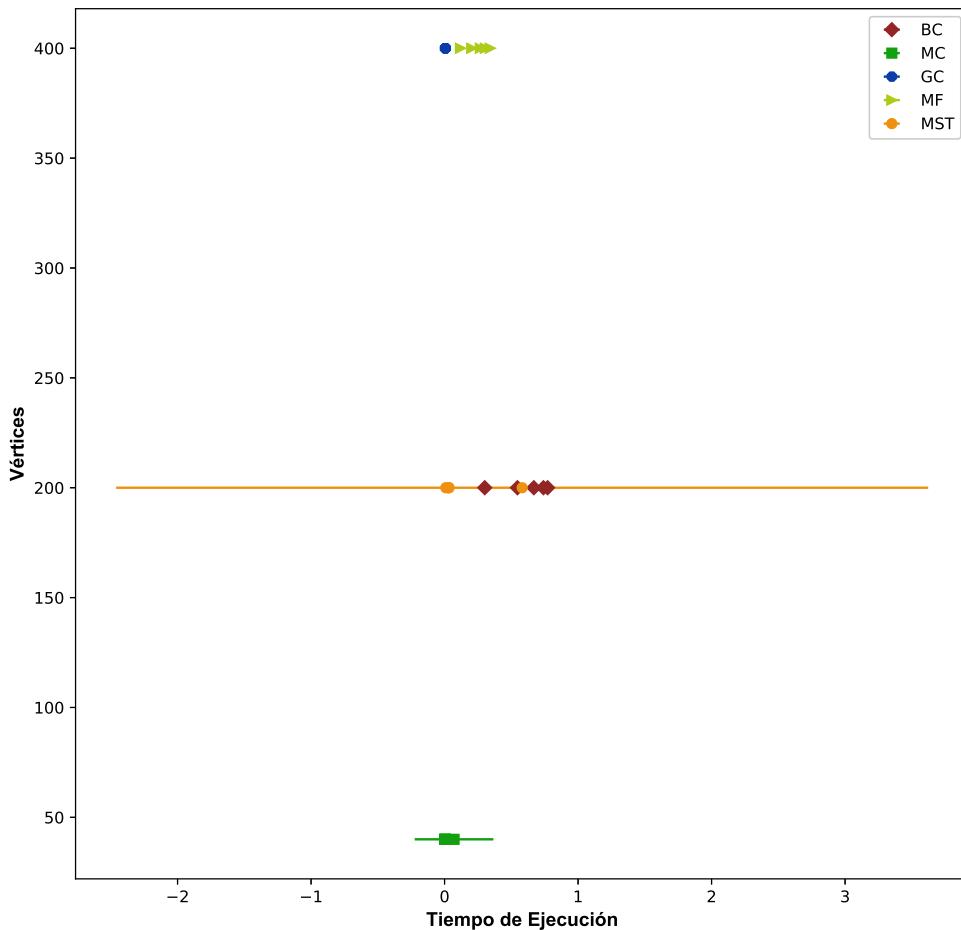


Figura 2: Diagrama de dispersión (cant. nodos vs media-algoritmo)

En la figura 3 de la página 12 se muestra la relación entre la media de cada algoritmo y la cantidad de arcos que se emplearon por grafo. En esta se aprecia que como tendencia general en los algoritmos medidos, al aumentar el número de arcos aumenta la media, el que más aumenta en su media es el algoritmo BC, en el algoritmo GC el valor de la media permanece casi constante a pesar del aumento de arcos, lo cual tiene sentido por el modo en que opera el mismo, el algoritmo MS actua de manera similar. Para el MC es casi despreciable la cantidad de arcos, debido a la demora del mismo se decidió disminuirlos.

Seguidamente se muestra el fragmento de código:

```

1 axes.legend()
2 plt.savefig("Imagenes/Fig2.eps")
3 plt.show()
4

```

```

5
6
7 figure , axes = plt.subplots(figsize=(10, 10))
8
9 x=data[ "Media_A1bc" ]
10 y=data[ "grafoE" ]
11 xerr=data[ "Desvi_A1bc" ]
12 axes.errorbar(x, y, xerr=xerr, fmt='D', color="#932525", alpha=1, label="BC")
13 print(y)
14
15 x=data[ "Media_A2mc" ]
16 y=data[ "clicgrafoE" ]
17 xerr=data[ "Desvi_A2mc" ]
18 axes.errorbar(x, y, xerr=xerr, fmt='s', color="#129f10", alpha=1, label="MC")
19 print(y)
20
21 x=data[ "Media_A3gc" ]
22 y=data[ "digrafoE" ]
23 xerr=data[ "Desvi_A3gc" ]
24 axes.errorbar(x, y, xerr=xerr, fmt='8', color="#093ea8", alpha=1, label="GC")
25 print(y)
26
27 x=data[ "Media_A4mf" ]
28 y=data[ "digrafoE" ]
29 xerr=data[ "Desvi_A4mf" ]
30 axes.errorbar(x, y, xerr=xerr, fmt='>', color="#adcb18", alpha=1, label="MF")
31 print(y)
32
33 x=data[ "Media_A5mst" ]
34 y=data[ "grafoE" ]
35 xerr=data[ "Desvi_A5mst" ]
36 axes.errorbar(x, y, fmt='o', color="#ee9110", alpha=1, label="MST")
37 print(y)
38
39 axes.set_ylabel("Aritas", fontsize=12, fontfamily="arial", fontweight="bold")
40 axes.set_xlabel("Tiempo de Ejecucion", fontsize=12, fontfamily="arial", fontweight="bold")

```

Scatterplot.py

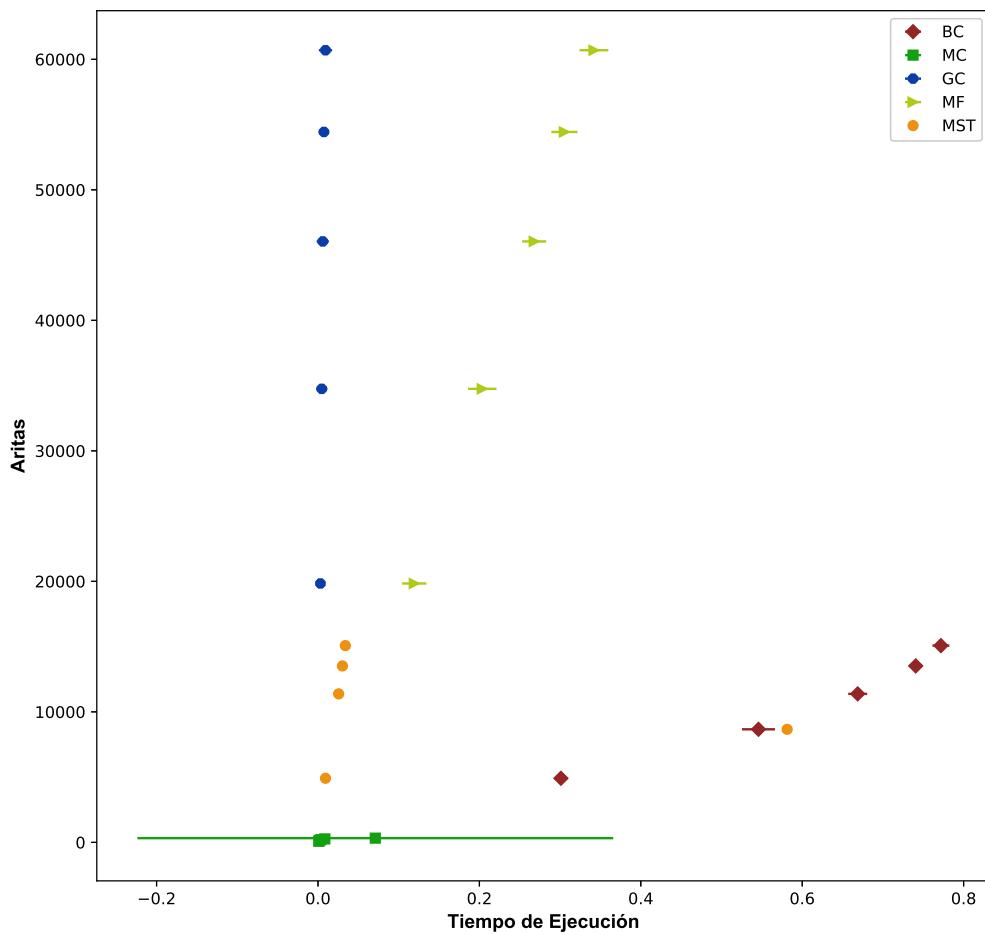
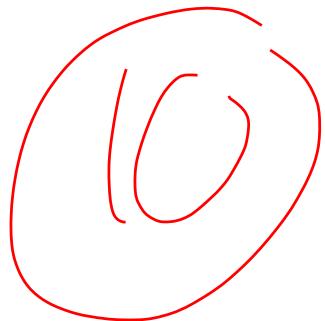


Figura 3: Diagrama de dispersión (cant. arcos vs media-algoritmo)

Referencias

- [de NetworkX(a)] Desarrolladores de NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.centrality.betweenness_centrality.html, a. Accessed: 2019-03-10.
- [de NetworkX(b)] Desarrolladores de NetworkX. https://networkx.github.io/documentation/networkx-1.10/_modules/networkx/algorithms/centrality/betweenness.html. betweenness_centrality, b. Accessed: 2019-03-18.
- [NetworkX(a)] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/_modules/networkx/algorithms/coloring/greedy_coloring.html#greedy_color, a. Accessed: 2019-03-17.
- [NetworkX(b)] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.flow.maximum_flow.html, b. Accessed: 2019-03-17.
- [NetworkX(c)] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.mst.minimum_spanning_tree.html, c. Accessed: 2019-03-17.



Tarea No.4: Flujo en Redes

Dayli Machado (5275)

2 de abril de 2019

1. Objetivo

Determinar mediante un diseño de experimentos empleando el análisis de varianzas de un factor y otras pruebas estadísticas la influencia que puede tener en la variable dependiente *tiempo de ejecución*, el orden y la densidad del grafo así como el generador de grafos y el algoritmo de flujo máximo seleccionado.

2. Generador de grafos, algoritmos de flujo máximo empleados y generación del .csv

De los tipos de generador de grafos se seleccionaron los generadores aleatorios y dentro de estos el grupo de tres modelos de generadores de grafos desarrollados por ~~Watts y Strogatz~~ [5]. Estos permiten de una forma relativamente sencilla y con menor número de parámetros generar los grafos. Una propiedad importante de estos tres generadores de grafos es que se desarrollan bajo la teoría de red de mundos pequeños, bajo esta teoría se crean nodos principales los cuales están alejados entre sí y generalmente se grafican más grandes que los demás, alrededor de estos se crean nodos que sí son vecinos entre sí y se grafican con tamaños más pequeños, de esta manera se garantiza la propiedad de crear grupos dentro del mismo grafo permitiendo que sea relativamente fácil realizar la visita entre todos los nodos. Esta propiedad refleja mejor el comportamiento de fenómenos reales como las redes eléctricas, neuronales y sociales. Otra propiedad de estos generadores de grafos es que la distancia esperada entre dos nodos elegidos al azar crece de manera proporcional al logaritmo de la cantidad de nodos de la red mientras no se trate de los nodos que están más agrupados, propiedad que detectaron los creadores a partir del comportamiento real de diferentes fenómenos [2]. De ahí que los generadores de grafos seleccionados fueron los siguientes:

- grafo de ~~Watts strogatz~~ ~~Newman~~
- grafo de ~~Watts strogatz~~
- grafo de ~~Watts strogatz~~ conectado

Los algoritmos de flujo máximo seleccionados fueron los siguientes:

- *Algoritmo Boykov-Kolmogorov*
- *Algoritmo de flujo máximo*
- *Algoritmo Edmonds-Karp*

El algoritmo de máximo flujo determina la ruta a través de la cual puede pasar el máximo flujo, de ahí que uno de los parámetros que requiere es la capacidad, y un su defecto la asume como infinita [6]. Se recomienda emplearlo en grafos dirigidos pero funciona también para no dirigidos.

El algoritmo de *Boykov-Kolmogorov* encuentra el flujo máximo de un sólo producto este devuelve la red residual resultante después de calcular el flujo máximo, debe tener capacidad en sus pesos sino los toma como infinitos [3]. Se recomienda para grafos dirigidos, aunque puede emplearse también para no dirigidos.

El algoritmo de *Edmonds-Karp* calcula el flujo máximo de un producto y además devuelve la red residual del mismo, ~~debe emplearse con grafos dirigidos aunque también se emplea para no dirigidos~~. Al igual que los anteriores debe asignarse una capacidad sino la ~~toma como infinita~~ [4].

A continuación se muestra el fragmento de código desarrollado para generar los grafos empleando los diferentes algoritmos:

```

1 genera_grafo = {"newman_watts_strogatz_graph": nx.newman_watts_strogatz_graph ,
2                     "watts_strogatz_graph": nx.watts_strogatz_graph ,
3                     "connected_watts_strogatz_graph": nx.
4                         connected_watts_strogatz_graph}
5 algoritmos_flujomax = { "maximum_flow": nx.maximum_flow ,
6                         "boykov_kolmogorov": boykov_kolmogorov ,
7                         "edmonds_karp": edmonds_karp}

```

Tarea4csvfinal.py

```

1 for generador_grafo in genera_grafo:
2     for instancia_grafo_x_nodos in [round(pow(2.6, value + 1)) for value in range(4,
3     8)]: # eleva a la potencia partiendo de la base 2.6

```

Tarea4csvfinal.py

En el siguiente fragmento se muestra como se ha desarrollado el código para asignar el peso normalmente distribuido a los ejes, apoyándose en [1].

```

1 grafo_temp = genera_grafo[generador_grafo](instancia_grafo_x_nodos ,
2                                              round((
3                                                 instancia_grafo_x_nodos * 0.15) / 2),
4                                              0.15,
5                                              seed=None)
6
6      aristas = grafo_temp.number_of_edges()
7      pesos_normalmente_distribuidos = np.random.normal(15, 0.2, aristas)
8
9      increment = 0 # incremento para que itere dentro del for que es el grafo
10     , magia!!!
11     for (u, v) in grafo_temp.edges():
12         grafo_temp.edges[u, v][“capacity”] = pesos_normalmente_distribuidos[
13             increment]
14         increment += 1
15
16     for instancia_grafo in range(1, 6):
17         for algoritmo_flujo in algoritmos_flujomax:
18             tabla_tiempo_ejec = []
19             for medicion in range(1, 6):
20                 hora_inicio = dt.datetime.now()
21                 obj = algoritmos_flujomax[algoritmo_flujo](grafo_temp,
22                     fuente, sumidero, capacity=“capacity”)
23                     hora.fin = dt.datetime.now()
24                     tiempo_consumido_segundos = (hora.fin - hora_inicio).
25                         total_seconds()
26                     tabla_tiempo_ejec.append(tiempo_consumido_segundos)
27                     media = stats.mean(tabla_tiempo_ejec)

```

Tarea4csvfinal.py

Después se genera el .csv con la siguiente estructura:

```

1     estructura_CSV[ "grafo" ].append( "vertices" + str(
2         instancia_grafo_x_nodos ) + "aristas" + str(aristas) )
3     estructura_CSV[ "algoritmo_flujo" ].append(algoritmo_flujo)
4     estructura_CSV[ "generador" ].append(generador_grafo)
5     estructura_CSV[ "vertices" ].append(instancia_grafo_x_nodos)
6     estructura_CSV[ "aristas" ].append(aristas)
7     estructura_CSV[ "fuente" ].append(fuente)
8     estructura_CSV[ "sumidero" ].append(sumidero)
9     estructura_CSV[ "densidad" ].append(nx.density(grafo_temp))
10    estructura_CSV[ "media" ].append(round(media, 5))
11    estructura_CSV[ "mediana" ].append(round(stats.median(
12        tabla_tiempo_ejec), 5))
12    estructura_CSV[ "varianza" ].append(round(stats.pvariance(
13        tabla_tiempo_ejec, mu=media), 5))
13    estructura_CSV[ "desviacion" ].append(round(stats.pstdev(
14        tabla_tiempo_ejec, mu=media), 5))

```

Tarea4csvfinal.py

Con los datos generados se pasa a realizar el análisis estadístico de los mismos.

3. Análisis de varianza (ANOVA), prueba de *Tukey* y relación general entre factores

Para realizar el análisis del comportamiento de la variable dependiente *tiempo de ejecución* con respecto a cada factor a analizar se realizó un análisis de varianza (ANOVA) para cada factor.

En el caso del análisis de la densidad vs *tiempo de ejecución* se convirtieron los valores de densidad a rangos de valores genéricos, para ello se realizó un histograma para dividir los rangos y llevarlos a una escala cualitativa en correspondencia con el arreglo obtenido delbins. El arreglo se muestra en el cuadro siguiente y el histograma en la figura 1 de la página 5

Visualmente del histograma es complejo identificar los rangos correctos para establecer la escala, por lo que se trabajó con el arreglo de rangos que devuelve el histograma siguiente:

Cuadro 1: Arreglo para los rangos de densidad

0.0656	0.0713	0.077	0.0827
--------	--------	-------	--------

A continuación se muestran los cuadros que resumen el resultado del ANOVA para cada factor:

Al analizar los resultados individualmente se observa que en todos los casos el valor del p-valor es menor que el valor de alfa de 0,05. Por lo que en todos los casos se rechaza la hipótesis nula y se acepta la hipótesis alternativa, la que afirma que existen diferencias entre las medias globales de cada factor y las medias de cada grupo por factor para los cuatro casos analizados. Lo anterior se refleja en los siguientes diagramas de cajas realizados para cada caso que se reflejan en la figura 2 de la página 7. De estos se reafirma la idea de que existe diferencia entre las medias de cada grupo por



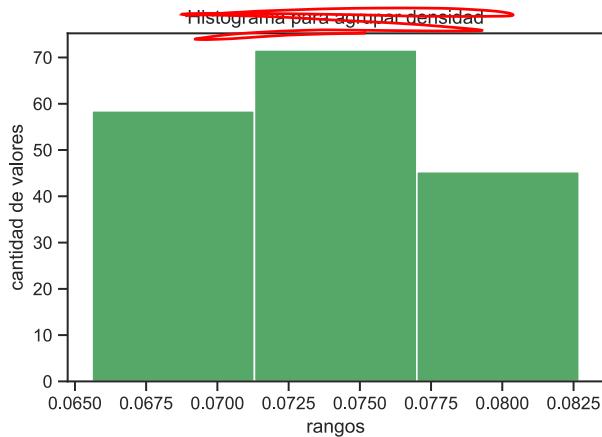


Figura 1: Histograma para determinar escala de densidad

Cuadro 2: ANOVA del tiempo de ejecución vs algoritmo de flujo

Source	SS	DF	MS	F	p-unc	np2
algoritmo_flujo	21.6	2	0.88	3.468	0.03127659	0.004
Within	456.098	1797	0.254	-	-	-

Cuadro 3: ANOVA del tiempo de ejecución vs densidad del grafo

Source	SS	DF	MS	F	p-unc	np2
convlogdensidad	61.501	2	30.784	139.629	4.31E-57	0.135
Within	396.49	1797	0.22	-	-	-

Cuadro 4: ANOVA del tiempo de ejecución vs generador

Source	SS	DF	MS	F	p-unc	np2
generador	3.095	2	1.547	6.116	0.00225312	0.007
Within	454.624	1797	0.253	-	-	-

Cuadro 5: ANOVA del tiempo de ejecución vs vértices

Source	SS	DF	MS	F	p-unc	np2
vertices	433.025	3	144.408	10571.46	0	0.946
Within	245.534	1796	0.014	-	-	-

factor por lo que sí se puede decir que existe una influencia de cada factor en la variable dependiente *tiempo de ejecución*. Al analizar cada factor se pudiera decir que para el 95 % de las veces como nivel de confianza y un margen de error de 0,05, al realizar el experimento para evaluar cuanto influye el algoritmo de flujo máximo seleccionado en el tiempo promedio de ejecución, el algoritmo que más influye es el *Boykov Kolmogorov*. En el caso del generador de grafos, aparentemente el que más influye en el *tiempo de ejecución* es el Watts Strogatz Newman, para el factor cantidad de vértices, la mayor influencia en el tiempo de ejecución está en el caso que poseen mayor cantidad de vértices, y según la densidad, influye más el rango medio de densidad.

Para tener más certeza sobre cuál de las categorías por factor es la que más influye en la variable dependiente, es recomendable aplicar después del análisis ANOVA una prueba de rango múltiple, en este caso se aplicó la prueba de *TUKEY*, para comprobar la hipótesis por pares en cada grupo de factor. A continuación se muestran los cuadros con los resultados para cada grupo por factor.

Cuadro 6: *Tukey* para factor: Algoritmo de Flujo

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
boykov_kolmogorov	edmonds_karp	-0.0621	-0.1304	0.0061	<i>False</i>
boykov_kolmogorov	maximum_flow	-0.0699	-0.1381	-0.0016	<i>True</i>
edmonds_karp	maximum_flow	-0.0077	-0.0759	0.0605	<i>False</i>

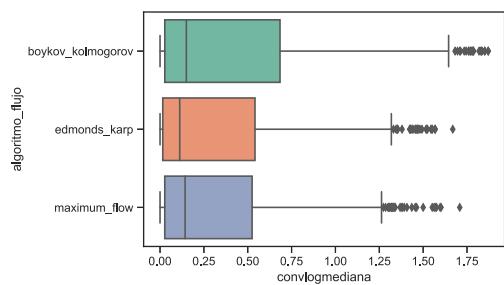
Cuadro 7: *Tukey* para factor: Densidad

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alto	bajo	-0.3101	-0.3851	-0.235	<i>True</i>
alto	medio	0.2192	0.1623	0.276	<i>True</i>
bajo	medio	0.5292	0.4538	0.6047	<i>True</i>

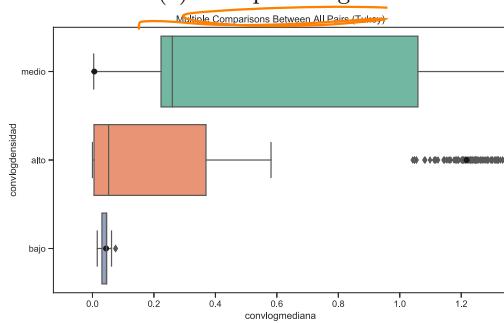
Cuadro 8: *Tukey* para factor: Generador grafo

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
connected_watts_strogatz_graph	newman_watts_strogatz_graph	0.0948	0.0267	0.163	<i>True</i>
connected_watts_strogatz_graph	watts_strogatz_graph	0.0159	-0.0522	0.084	<i>False</i>
newman_watts_strogatz_graph	watts_strogatz_graph	-0.0789	-0.147	-0.0108	<i>True</i>

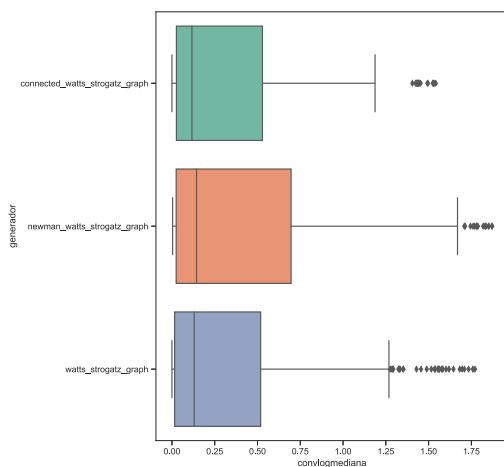
CWS NWS
WS



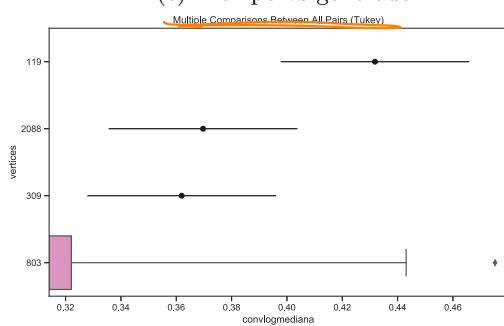
(a) Tiempo vs algoritmo



(b) Tiempo vs densidad del grafo



(c) Tiempo vs generador



(d) Tiempo vs vértices

Figura 2: Diagramas de cajas por factor vs tiempo de ejecución

Cuadro 9: *Tukey* para factor: Cantidad de vértices

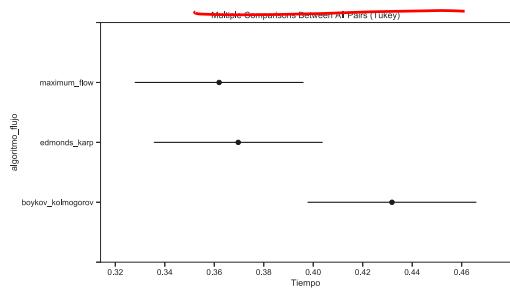
<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
119	2088	1.2113	1.1913	1.2314	<i>True</i>
119	309	0.0384	0.0183	0.0584	<i>True</i>
119	803	0.2773	0.2573	0.2974	<i>True</i>
2088	309	-1.1729	-1.193	-1.1529	<i>True</i>
2088	803	-0.934	-0.954	-0.9139	<i>True</i>
309	803	0.239	0.2189	0.259	<i>True</i>

De estas tablas se analizan aquellos pares cuyos valores de diferencia de medias sean mayores y se devuelve cual es del par el que más influye. El mismo resultado se puede apreciar mejor en la figura 3 de la página 9.

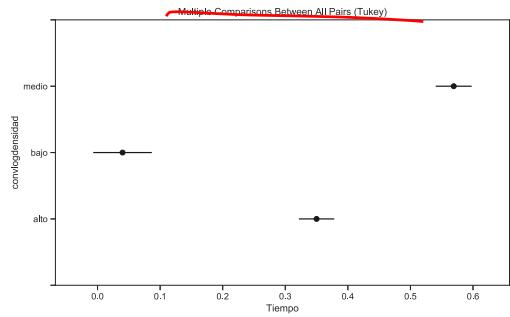
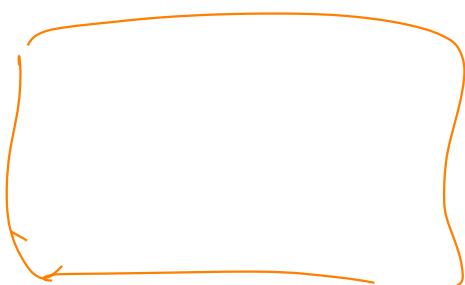
Del análisis del algoritmo de flujo se aprecia que se mantiene el algoritmo de *Boykov Kolmogorov* como el de intervalo de desviación más amplio, y por tanto afecta más, en el par donde se combina. Para la densidad del grafo se aprecia que a pesar de existir diferencia en todos los pares, el que más influye es de la combinación medio con bajo niveles de densidad, se aprecia que el bajo tiene un intervalo mayor, pero el medio posee valores más altos, lo que significa que existe más diferencia entre los valores del rango bajo, pero los de nivel medio influyen más. Para el caso del generador de grafos, se aprecia similitud en los intervalos, por lo que más influye el *Watts Strogatz Newman*. Según la cantidad de vértices en la prueba de *Tukey* no existe diferencia significativa entre los límites superiores e inferiores, y se mantiene que más influye el de mayor cantidad de vértices.

Para conocer la relación entre todas las variables y su influencia en el tiempo de ejecución se realiza la interacción entre las ANOVAS de cada factor, los resultados muestran que los factores que más influyen en el tiempo de ejecución son el generador de grafos y el algoritmo seleccionado, en estos es donde el *p* valor es menor que 0,05 por tanto se rechaza la hipótesis nula, aceptando la hipótesis alternativa. Los resultados globales se muestran en el cuadro siguiente:

El código que se empleó para realizar el análisis estadístico es el siguiente:

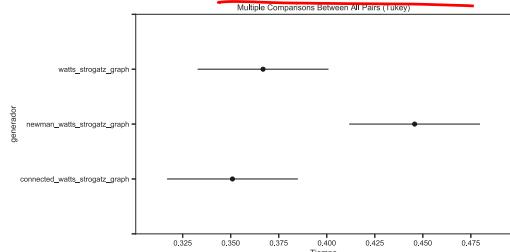


(a) Tukey: Tiempo vs algoritmo

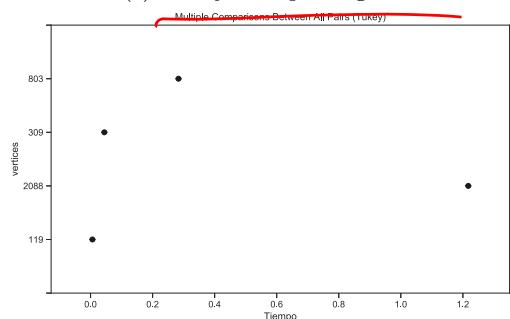
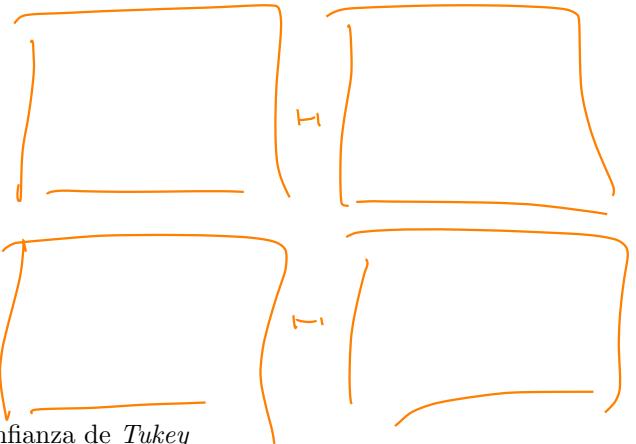


(b) Tukey: Tiempo vs densidad del grafo

\quad quad



(c) Tukey: Tiempo vs generador



(d) Tukey: Tiempo vs vértices

Figura 3: Intervalos de confianza de Tukey

Cuadro 10: Interacción entre ANOVAS de un factor vs tiempo de ejecución

	sum_sq	df	F	PR(>F)
generador	0.49851765	2	33.2428314	9.57E-09
algoritmo_flujo	1.76023524	2	117.378575	1.33E-48
vertices	-1.16E-11	3	-5.17E-10	1
convlogdensidad	-1.62E-11	2	-1.08E-09	1
generador:algoritmo_flujo	0.00835511	4	0.27857381	0.8919532
algoritmo_flujo:vertices	2.6038422	6	57.877735	1.44E-63
vertices:convlogdensidad	19.3562824	6	430.247555	5.04E-210
generador:vertices	48.5962079	6	1080.18775	0
generador:convlogdensidad	0.47147159	4	15.7196782	4.32E-10
Residual	13.3091476	1775		

```

1 import csv
2 import pandas as pd
3 import scipy.stats as stats
4 import matplotlib.pyplot as plt
5 import researchpy as rp
6 import statsmodels.api as sm
7 from statsmodels.formula.api import ols
8 import numpy as np
9 import pingouin as pg
10 import seaborn as sns
11 from statsmodels.stats.multicomp import pairwise_tukeyhsd

```

Tarea4Estadisticsfinal.py

```

1 np.float64})
2 #mejorar los valores de la mediana
3 logX = np.log1p(df['mediana'])
4 df = df.assign(convlogmediana=logX.values)
5 df.drop(['mediana'], axis= 1, inplace= True)
6
7 #agrupar los valos de densidad y convertirlo en grupos de baja, media y alta
8 #densidad
9
10 logX = np.log1p(df['densidad'])
11 df = df.assign(convlogdensidad=logX.values)
12 df.drop(['densidad'], axis= 1, inplace= True)
13
14 his = plt.hist(round(df["convlogdensidad"],4), bins=3, density=True, facecolor='g',
15 alpha=0.75)
16 plt.xlabel("rangos")
17 plt.ylabel("cantidad de valores")
18 plt.title("Histograma para agrupar densidad")
19 plt.savefig("Imagenes/Histogramadensidad"+".png")
20 plt.savefig("Imagenes/Histogramadensidad"+".eps")
21

```

```
22 print("bins s"
```

Tarea4Estadisticsfinal.py

```
1 anova_factores=[“generador”, “algoritmo_flujo”, “vertices”, “convlogdensidad”]
2 plt.figure(figsize=(8, 10))
3 for i in anova_factores:
4
5     print(rp.summary_cont(df[‘convlogmediana’].groupby(df[i])))
6
7     anovaporfactor = pg.anova(dv=‘convlogmediana’, between=i, data=df, detailed=True , )
8     pg._export_table(anovaporfactor,(“estadisttable/Tabla_ANOVA”+i+“.csv”))
9
10    ejes=sns.boxplot(x=df[“convlogmediana”], y=df[i], data=df, palette=“Set2”)
11    plt.savefig(“Imagenes/aboxplot”+ i+“.png”, bbox_inches=‘tight’)
12    plt.savefig(“Imagenes/aboxplot” + i + “.eps”, bbox_inches=‘tight’)
13    tukey = pairwise_tukeyhsd(endog = df[“convlogmediana”],      # Data
14                           groups= df[i],    # Groups
15                           alpha=0.05)          # Significance level
16
17    tukey.plot_simultaneous(xlabel=‘Tiempo’, ylabel=i)      # Plot group confidence
18    intervals
19    # plt.vlines(x=49.57,ymin=-0.5,ymax=4.5, color=“red”)
20
21    plt.savefig(“Imagenes/tablatukey”+ i+“.png”, bbox_inches=‘tight’)
22    plt.savefig(“Imagenes/tablatukey” + i + “.eps”, bbox_inches=‘tight’)
23    print(tukey.summary())
24
25    excel_tukey = open(“estadisttable/Tukey”+i+“.csv”, ‘w’)
26    with excel_tukey:
27        writer = csv.writer(excel_tukey)
28        writer.writerows(tukey.summary())
29
30    model_name = ols(‘convlogmediana ~ generador+algoritmo_flujo+vertices+
31                      convlogdensidad +generador*algoritmo_flujo+algoritmo_flujo*vertices+vertices*
32                      convlogdensidad+generador*vertices+generador*convlogdensidad+algoritmo_flujo+
33                      vertices*convlogdensidad’, data=df).fit()
34    model_name.summary()
35    aov_table = sm.stats.anova_lm(model_name, typ=2)
36    df1=pd.DataFrame(aov_table)
37    df1.to_csv(“multianova.csv”)
38
39    plt.show()
40    print (“fin”)
```

Tarea4Estadisticsfinal.py

Referencias

- [1] Leonardo J. Caballero. *Materiales del entrenamiento de programación en Python*. 2019.
- [2] Steven H. Strogatz Duncan J. Watts. Collective dynamics of ‘small-world’ networks. *Nature*, 393:440–442, Jun. 1998. doi: 10.1038/30918. ~~URL <https://www.yourteam.com/references/Watts-CollectiveDynamics-SmallWorldNetworks.pdf>.~~

- [3] Desarrolladores NetworkX. https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.flow.boykov_kolmogorov.html, . Accessed:2019-03-31.
- [4] Desarrolladores NetworkX. https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.flow.edmonds_karp.html, . Accessed:2019-03-31.
- [5] Desarrolladores NetworkX. <https://networkx.github.io/documentation/stable/reference/generators.html>, . Accessed: 2019-03-29.
- [6] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.flow.maximum_flow.html, . Accessed: 2019-03-17.

Notes

10-2

Tarea No.4: Rectificada

Dayli Machado (5275)

3 de junio de 2019

1. Objetivo

Determinar mediante un diseño de experimentos empleando el análisis de varianzas de un factor y otras pruebas estadísticas la influencia que puede tener en la variable dependiente *tiempo de ejecución*, el orden y la densidad del grafo así como el generador de grafos y el algoritmo de flujo máximo seleccionado.

2. Generador de grafos, algoritmos de flujo máximo empleados y generación del .csv

De los tipos de generador de grafos se seleccionaron los generadores aleatorios y dentro de estos el grupo de tres modelos de generadores de grafos desarrollados por *Watts y Strogatz* [NetworkX(c)]. Estos permiten de una forma relativamente sencilla y con menor número de parámetros generar los grafos. Una propiedad importante de estos tres generadores de grafos es que se desarrollan bajo la teoría de red de mundos pequeños, bajo esta teoría se crean nodos principales los cuales están alejados entre sí y generalmente se grafican más grandes que los demás, alrededor de estos se crean nodos que sí son vecinos entre sí y se grafican con tamaños más pequeños, de esta manera se garantiza la propiedad de crear grupos dentro del mismo grafo permitiendo que sea relativamente fácil realizar la visita entre todos los nodos. Esta propiedad refleja mejor el comportamiento de fenómenos reales como las redes eléctricas, neuronales y sociales. Otra propiedad de estos generadores de grafos es que la distancia esperada entre dos nodos elegidos al azar crece de manera proporcional al logaritmo de la cantidad de nodos de la red mientras no se trate de los nodos que están más agrupados, propiedad que detectaron los creadores a partir del comportamiento real de diferentes fenómenos [Duncan J. Watts(1998)]. De ahí que los generadores de grafos seleccionados fueron los siguientes:

- *grafo de Watts Strogatz Newman*
- *grafo de Watts Strogatz*
- *grafo de Watts Strogatz Conectado*

Los algoritmos de flujo máximo seleccionados fueron los siguientes:

- *Algoritmo Boykov-Kolmogorov*

- *Algoritmo de flujo máximo*
- *Algoritmo Edmonds-Karp*

El algoritmo de máximo flujo determina la ruta a través de la cual puede pasar el máximo flujo, de ahí que uno de los parámetros que requiere es la capacidad, y un su defecto la asume como infinita [NetworkX(d)]. Se recomienda emplearlo en grafos dirigidos pero funciona también para no dirigidos.

El algoritmo de *Boykov-Kolmogorov* encuentra el flujo máximo de un sólo producto este devuelve la red residual resultante después de calcular el flujo máximo, debe tener capacidad en sus pesos sino los toma como infinitos [NetworkX(a)]. Se recomienda para grafos dirigidos, aunque puede emplearse también para no dirigidos.

El algoritmo de *Edmonds-Karp* calcula el flujo máximo de un producto y además devuelve la red residual del mismo, se emplea en grafos dirigidos y no dirigidos. Al igual que los anteriores debe asignarse una capacidad sino la toma como infinita [NetworkX(b)].

A continuación se muestra el fragmento de código desarrollado para generar los grafos empleando los diferentes algoritmos:

```

1 genera_grafo = {"newman_watts_strogatz_graph": nx.newman_watts_strogatz_graph ,
2                 "watts_strogatz_graph": nx.watts_strogatz_graph ,
3                 "connected_watts_strogatz_graph": nx.
4                 connected_watts_strogatz_graph}
5 algoritmos_flujomax = { "maximum_flow": nx.maximum_flow ,
6                         "boykov_kolmogorov": boykov_kolmogorov ,
7                         "edmonds_karp": edmonds_karp}

```

Tarea4csvfinal.py

```

1 for generador_grafo in genera_grafo:
2     for instancia_grafo_x_nodos in [round(pow(2.6, value + 1)) for value in range(4,
3 ]): # eleva a la potencia partiendo de la base 2.6

```

Tarea4csvfinal.py

En el siguiente fragmento se muestra como se ha desarrollado el código para asignar el peso normalmente distribuido a cada arista, apoyándose en [Caballero(2019)].

```

1         grafo_temp = genera_grafo[generador_grafo]( instancia_grafo_x_nodos ,
2                                         round((
3                                         instancia_grafo_x_nodos * 0.15) / 2),
4                                         0.15,
5                                         seed=None)
6
6         aristas = grafo_temp.number_of_edges()
7         pesos_normalmente_distribuidos = np.random.normal(15, 0.2, aristas)
8
8         increment = 0 # incremento para que itere dentro del for que es el grafo ,
9         magia !!!
10        for (u, v) in grafo_temp.edges():
11            grafo_temp.edges[u, v][ "capacity" ] = pesos_normalmente_distribuidos [
12            increment]
13            increment += 1
14
14        for instancia_grafo in range(1, 6):
15            for algoritmo_flujo in algoritmos_flujomax:
16                tabla_tiempo_ejec = []
17                for medicion in range(1, 6):
18                    hora_inicio = dt.datetime.now()
19                    obj = algoritmos_flujomax[algoritmo_flujo](grafo_temp, fuente ,
20                      sumidero, capacity="capacity")
21                    hora_fin = dt.datetime.now()
22                    tiempo_consumido_segundos = (hora_fin - hora_inicio).
23                    total_seconds()
24                    tabla_tiempo_ejec.append(tiempo_consumido_segundos)
25                    media = stats.mean(tabla_tiempo_ejec)

```

Tarea4csvfinal.py

Después se genera el .csv con la siguiente estructura:

```

1      estructura_CSV["grafo"].append("vertices" + str(
2          instancia_grafo_x_nodos) + "aristas" + str(aristas))
3      estructura_CSV["algoritmo_flujo"].append(algoritmo_flujo)
4      estructura_CSV["generador"].append(generador_grafo)
5      estructura_CSV["vertices"].append(instancia_grafo_x_nodos)
6      estructura_CSV["aristas"].append(aristas)
7      estructura_CSV["fuente"].append(fuente)
8      estructura_CSV["sumidero"].append(sumidero)
9      estructura_CSV["densidad"].append(nx.density(grafo_temp))
10     estructura_CSV["media"].append(round(media, 5))
11     estructura_CSV["mediana"].append(round(stats.median(
12         tabla_tiempo_ejec), 5))
12     estructura_CSV["varianza"].append(round(stats.pvariance(
13         tabla_tiempo_ejec, mu=media), 5))
13     estructura_CSV["desviacion"].append(round(stats.pstdev(
14         tabla_tiempo_ejec, mu=media), 5))

```

Tarea4csvfinal.py

Con los datos generados se pasa a realizar el análisis estadístico de los mismos.

3. Análisis de varianza (ANOVA), prueba de *Tukey* y relación general entre factores

Para realizar el análisis del comportamiento de la variable dependiente *tiempo de ejecución* con respecto a cada factor a analizar se realizó un análisis de varianza (ANOVA) para cada factor.

En el caso del análisis de la densidad vs *tiempo de ejecución* se convirtieron los valores de densidad a rangos de valores genéricos, para ello se realizó un histograma para dividir los rangos y llevarlos a una escala cualitativa en correspondencia con el arreglo obtenido del *bins*. El arreglo se muestra en el cuadro siguiente y el histograma en la figura 1 de la página 4

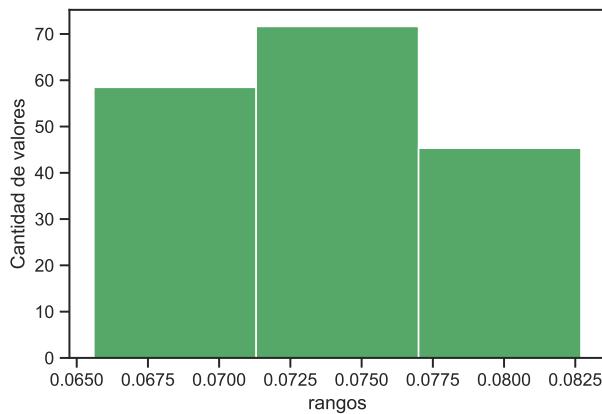


Figura 1: Histograma para determinar escala de densidad

Visualmente del histograma es complejo identificar los rangos correctos para establecer la escala, por lo que se trabajó con el arreglo de rangos que devuelve el histograma siguiente:

A continuación se muestran los cuadros que resumen el resultado del ANOVA para cada factor:

Cuadro 1: Arreglo para los rangos de densidad

0.0656	0.0713	0.077	0.0827
--------	--------	-------	--------

Cuadro 2: ANOVA del tiempo de ejecución vs algoritmo de flujo

Source	SS	DF	MS	F	p-unc	np2
algoritmo_flujo	1.76	2	0.88	3.46	0.03	0.00
Within	455.99	1797	0.25			

Cuadro 3: ANOVA del tiempo de ejecución vs densidad del grafo

Source	SS	DF	MS	F	p-unc	np2
convlogdensidad	61.56	2	30.78	139.62	0.00	0.13
Within	396.19	1797	0.22			

Cuadro 4: ANOVA del tiempo de ejecución vs generador

Source	SS	DF	MS	F	p-unc	np2
generador	3.09	2	1.54	6.11	0.00	0.00
Within	454.66	1797	0.253			

Cuadro 5: ANOVA del tiempo de ejecución vs vértices

Source	SS	DF	MS	F	p-unc	np2
vertices	433.22	3	144.40	10571.46	0	0.94
Within	24.53	1796	0.014			

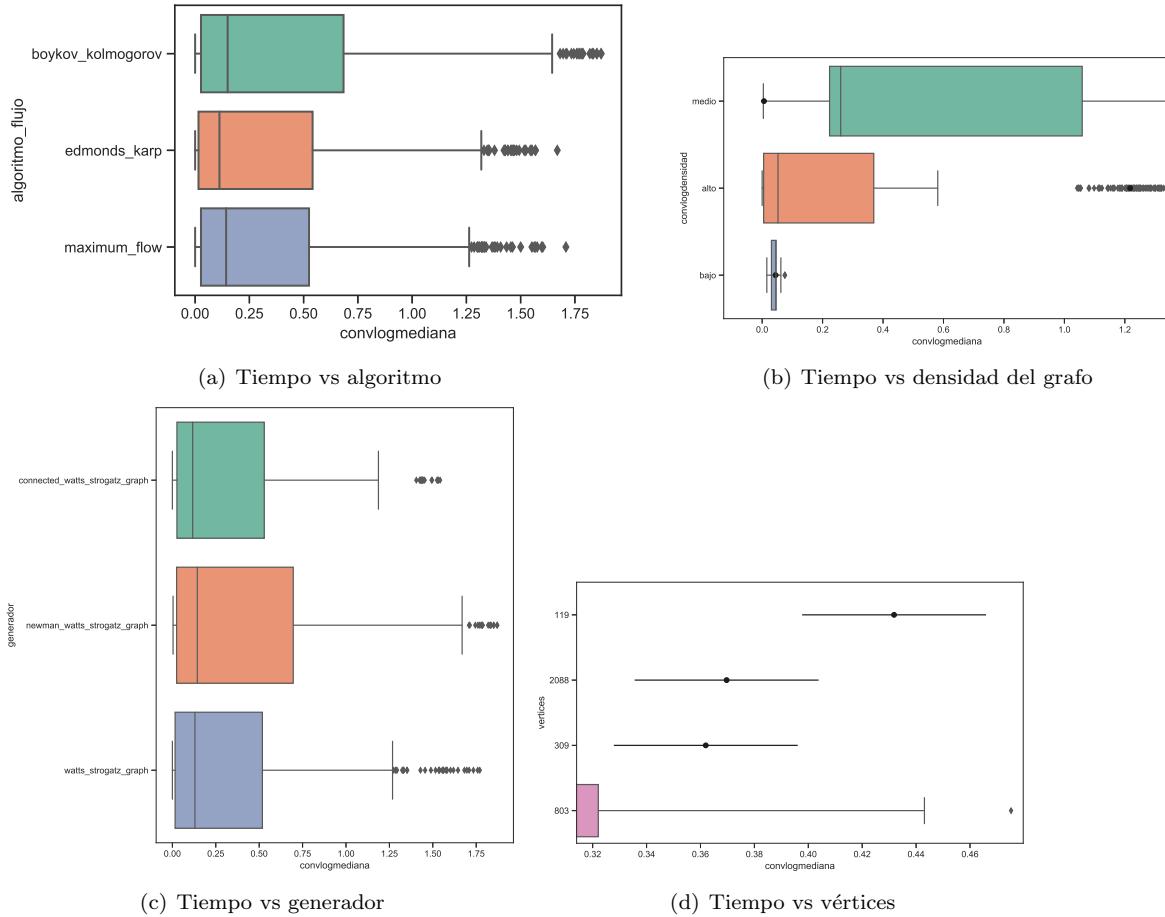


Figura 2: Diagramas de cajas por factor vs tiempo de ejecución

Al analizar los resultados individualmente se observa que en todos los casos el valor de $p - \text{valor}$ es menor que el valor de alfa de 0,05. Por lo que en todos los casos se rechaza la hipótesis nula y se acepta la hipótesis alternativa, la que afirma que existen diferencias entre las medias globales de cada factor y las medias de cada grupo por factor para los cuatro casos analizados. Lo anterior se refleja en los siguientes diagramas de cajas realizados para cada caso que se reflejan en la figura 2 de la página 6. De estos se reafirma la idea de que existe diferencia entre las medias de cada grupo por factor por lo que sí se puede decir que existe una influencia de cada factor en la variable dependiente *tiempo de ejecución*. Al analizar cada factor se pudiera decir que para el 95 % de las veces como nivel de confianza y un margen de error de 0,05, al realizar el experimento para evaluar cuento influye el algoritmo de flujo máximo seleccionado en el tiempo promedio de ejecución, el algoritmo que más influye es el *Boykov Kolmogorov*. En el caso del generador de grafos, aparentemente el que más influye en el *tiempo de ejecución* es el Watts Strogatz Newman, para el factor cantidad de vértices, la mayor influencia en el tiempo de ejecución está en el caso que poseen mayor cantidad de vértices, y según la densidad, influye más el rango medio de densidad.

Para tener más certeza sobre cuál de las categorías por factor es la que más influye en la variable dependiente, es recomendable aplicar después del análisis ANOVA una prueba de rango múltiple, en este caso se aplicó la prueba de *TUKEY*, para comprobar la hipótesis por pares en cada grupo de factor. A continuación se muestran los cuadros con los resultados para cada grupo por factor.

Cuadro 6: *Tukey* para factor: Algoritmo de Flujo

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
boykov_kolmogorov	edmonds_karp	-0.062	-0.130	0.006	Falso
boykov_kolmogorov	maximum_flow	-0.069	-0.138	-0.001	Verdadero
edmonds_karp	maximum_flow	-0.007	-0.0759	0.060	Falso

Cuadro 7: *Tukey* para factor: Densidad

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alto	bajo	-0.310	-0.385	-0.235	Verdadero
alto	medio	0.219	0.162	0.276	Verdadero
bajo	medio	0.529	0.453	0.604	Verdadero

Cuadro 8: *Tukey* para factor: Generador grafo

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
c_w_s_g	n_w_s_g	0.094	0.026	0.163	Verdadero
c_w_s_g	w_s_g	0.015	-0.052	0.084	Falso
n_w_s_g	w_s_g	-0.078	-0.147	-0.010	Verdadero

Cuadro 9: *Tukey* para factor: Cantidad de vértices

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
119	2088	1.211	1.191	1.231	Verdadero
119	309	0.038	0.018	0.058	Verdadero
119	803	0.277	0.257	0.297	Verdadero
2088	309	-1.172	-1.193	-1.152	Verdadero
2088	803	-0.934	-0.954	-0.913	Verdadero
309	803	0.239	0.218	0.259	Verdadero

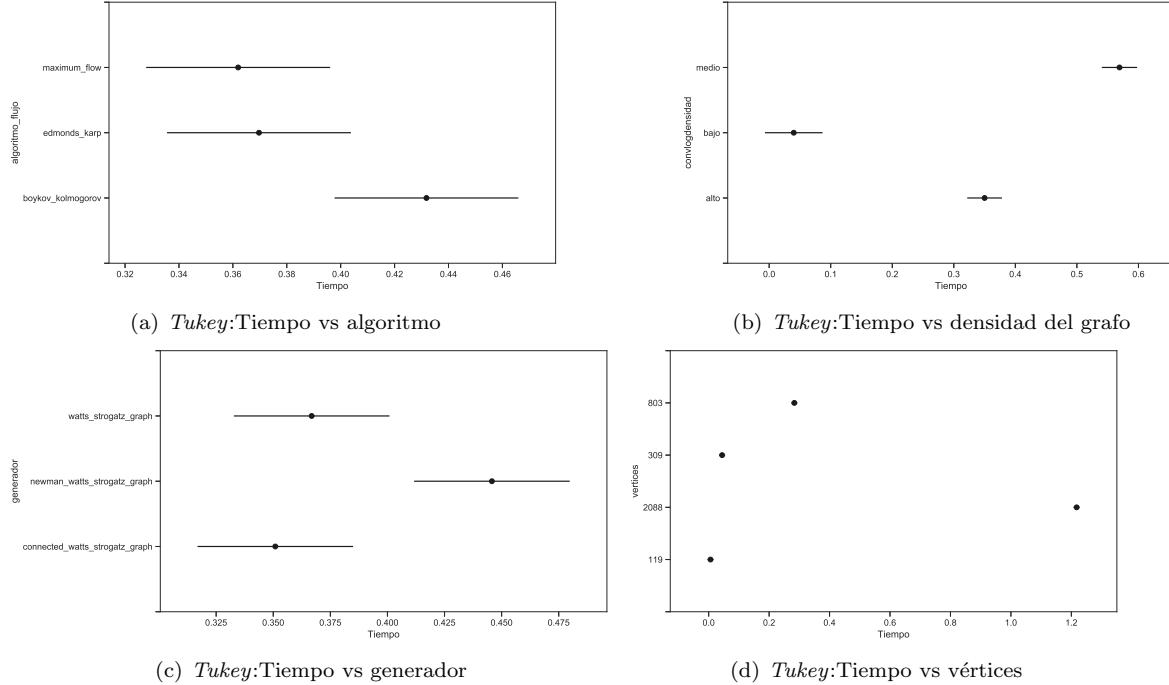


Figura 3: Intervalos de confianza de Tukey

De estas tablas se analizan aquellos pares cuyos valores de diferencia de medias sean mayores y se devuelve cual es del par el que más influye. El mismo resultado se puede apreciar mejor en la figura 3 de la página 8.

Del análisis del algoritmo de flujo se aprecia que se mantiene el algoritmo de *Boykov Kolmogorov* como el de intervalo de desviación más amplio, y por tanto afecta más, en el par donde se combina. Para la densidad del grafo se aprecia que a pesar de existir diferencia en todos los pares, el que más influye es de la combinación medio con bajo niveles de densidad, se aprecia que el bajo tiene un intervalo mayor, pero el medio posee valores más altos, lo que significa que existe más diferencia entre los valores del rango bajo, pero los de nivel medio influyen más. Para el caso del generador de grafos, se aprecia similitud en los intervalos, por lo que más influye el *Watts Strogatz Newman*. Según la cantidad de vértices en la prueba de Tukey no existe diferencia significativa entre los límites superiores e inferiores, y se mantiene que más influye el de mayor cantidad de vértices.

Para conocer la relación entre todas las variables y su influencia en el tiempo de ejecución se realiza la interacción entre las ANOVAS de cada factor, los resultados muestran que los factores que más influyen en el tiempo de ejecución son el generador de grafos y el algoritmo seleccionado, en estos es donde el p valor es menor que 0,05 por tanto se rechaza la hipótesis nula, aceptando la hipótesis alternativa. Los resultados globales se muestran en el cuadro siguiente:

El código que se empleó para realizar el análisis estadístico es el siguiente:

Cuadro 10: Interacción entre ANOVAS de un factor vs tiempo de ejecución

	sum_sq	df	F	PR(>F)
generador	0.498	2	33.242	0.000
algoritmo_flujo	1.7604	2	117.378	0.000
vertices	0.000	3	0.000	1.000
convlogdensidad	0.000	2	0.000	1.000
generador:algoritmo_flujo	0.008	4	0.278	0.891
algoritmo_flujo:vertices	2.603	6	57.877	0.000
vertices:convlogdensidad	19.356	6	430.247	0.000
generador:vertices	48.5969	6	1080.187	0.000
generador:convlogdensidad	0.471	4	15.719	0.000
Residual	13.309	1775		

```

1 import csv
2 import pandas as pd
3 import scipy.stats as stats
4 import matplotlib.pyplot as plt
5 import researchpy as rp
6 import statsmodels.api as sm
7 from statsmodels.formula.api import ols
8 import numpy as np
9 import pingouin as pg
10 import seaborn as sns
11 from statsmodels.stats.multicomp import pairwise_tukeyhsd

```

Tarea4Estadisticsfinal.py

```

1         float64})
2 #mejorar los valores de la mediana
3 logX = np.log1p(df[ 'mediana' ])
4 df = df.assign(convlogmediana=logX.values)
5 df.drop([ 'mediana' ], axis= 1, inplace= True)
6
7 #agrupar los valos de densidad y convertirlo en grupos de baja , media y alta densidad
8
9
10 logX = np.log1p(df[ 'densidad' ])
11 df = df.assign(convlogdensidad=logX.values)
12 df.drop([ 'densidad' ], axis= 1, inplace= True)
13
14
15 his = plt.hist(round(df[ "convlogdensidad" ]),4 ,bins=3, density=True, facecolor='g',
16 alpha=0.75)
17 plt.xlabel ("rangos")
18 plt.ylabel("cantidad de valores")
19 plt.title("Histograma para agrupar densidad")
20 plt.savefig("Imagenes/Histogramadensidad"+".png")
21 plt.savefig("Imagenes/Histogramadensidad"+".eps")
22 print(" bins s"

```

Tarea4Estadisticsfinal.py

```

1 anova_factores=[ "generador" , "algoritmo_flujo" , "vertices" , "convlogdensidad" ]
2 plt.figure(figsize=(8, 10))
3 for i in anova_factores:

```

```

4
5     print(rp.summary_cont(df[ 'convlogmediana' ].groupby(df[ i ])))
6
7     anovaporfactor = pg.anova (dv='convlogmediana' , between=i , data=df , detailed=True
8     ,
9     pg._export_table (anovaporfactor ,("estadisttable/Tabla_ANOVA"+i+".csv"))
10
11    ejes=sns.boxplot(x=df[ "convlogmediana" ] , y=df[ i ] , data=df , palette="Set2")
12    plt.savefig("Imagenes/aboxplot"+ i +".png" , bbox_inches='tight')
13    plt.savefig("Imagenes/aboxplot" + i + ".eps" , bbox_inches='tight')
14    tukey = pairwise_tukeyhsd(endog = df["convlogmediana"] ,           # Data
15                           groups= df[ i ] ,          # Groups
16                           alpha=0.05)                # Significance level
17
18    tukey.plot_simultaneous(xlabel='Tiempo' , ylabel=i)      # Plot group confidence
19    intervals
20    plt.vlines(x=49.57,ymin=-0.5,ymax=4.5, color="red")
21
22    plt.savefig("Imagenes/tablatukey"+ i +".png" , bbox_inches='tight')
23    plt.savefig("Imagenes/tablatukey" + i + ".eps" , bbox_inches='tight')
24    print(tukey.summary())
25
26    excel_tukey = open("estadisttable/Tukey"+i+".csv" , 'w')
27    with excel_tukey :
28        writer = csv.writer(excel_tukey)
29        writer.writerow(tukey.summary())
30
31    model_name = ols('convlogmediana ~ generador+algoritmo_flujo+vertices+convlogdensidad+
32                      +generador*algoritmo_flujo+algoritmo_flujo*vertices+vertices*convlogdensidad+
33                      +generador*vertices+generador*convlogdensidad+algoritmo_flujo+vertices*
34                      convlogdensidad' , data=df).fit()
35    model_name.summary()
36    aov_table = sm.stats.anova_lm(model_name , typ=2)
37    df1=pd.DataFrame(aov_table)
38    df1.to_csv("multianova.csv")
39
40    plt.show()
41    print ("fin")

```

Tarea4Estadisticsfinal.py

Referencias

[Caballero(2019)] Leonardo J. Caballero. *Materiales del entrenamiento de programación en Python.* 2019.

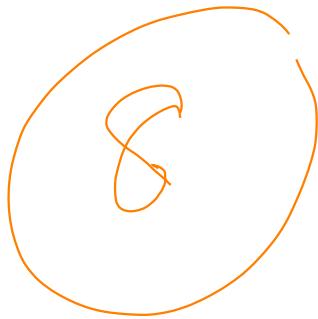
[Duncan J. Watts(1998)] Steven H. Strogatz Duncan J. Watts. Collective dynamics of ‘small-world’ networks. *Nature*, 393:440–442, jun. 1998. doi: 10.1038/30918. URL <https://worrydream.com/refs/Watts-CollectiveDynamicsOfSmallWorldNetworks.pdf>.

[NetworkX(a)] Desarrolladores NetworkX. https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.flow.boykov_kolmogorov.html, a. Accessed:2019-03-31.

[NetworkX(b)] Desarrolladores NetworkX. https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.flow.edmonds_karp.html, b. Accessed:2019-03-31.

[NetworkX(c)] Desarrolladores NetworkX. <https://networkx.github.io/documentation/stable/reference/generators.html>, c. Accessed: 2019-03-29.

[NetworkX(d)] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.flow.maximum_flow.html, d. Accessed: 2019-03-17.



Tarea No.5: Flujo en Redes

Dayli Machado (5275)

30 de abril de 2019

1. Objetivo

Determinar mediante un diseño de experimento, empleando el análisis de varianza de un factor y otras pruebas estadísticas, la influencia que pueden tener en las variables dependientes *tiempo de ejecución del algoritmo de flujo máximo seleccionado y el máximo flujo posible a obtener para diferentes combinaciones de fuentes y sumideros*, en función de las siguientes características estructurales de cada vértice: ~~distribución~~ de grado, coeficiente de agrupamiento, centralidad de cercanía, centralidad de carga, excentricidad y page rank.

2. Generador de grafos y algoritmo de flujo máximo seleccionado

Para la evaluación anterior se seleccionaron los generadores de grafos aleatorios los cuales según [6] son útiles para comprender los procesos estocásticos que ocurren en una red. Adicionalmente plantea que la generación de grafos aleatorios que controlan algunas condiciones, como la distribución de grado, el coeficiente de agrupamiento y la secuencia de grados, permite estudiar cómo se forman las estructuras de redes de la vida real y plantea como ejemplos prácticos de aplicación, la participación electoral, la propagación de epidemias, así como el comportamiento en redes sociales.

En la evaluación anterior se seleccionaron dentro del grupo de generadores de grafos aleatorios, tres modelos de generadores desarrollados por ~~Watts Strogatz~~ [8]. Estos permiten de una forma relativamente sencilla y con menor número de parámetros generar los grafos. Una propiedad importante de estos tres generadores es que se desarrollan bajo la teoría de red de mundos pequeños. Esta propiedad revela según sus creadores y también a criterio de [5] que la sociedad humana es una red social con forma de mundo pequeño y que están interconectados entre sí, con estructura de red, cuyos nódulos son personas y los enlaces, las interrelaciones entre ellos. Bajo esta teoría se crean nodos principales los cuales están alejados entre sí y generalmente se grafican más grandes que los demás, alrededor de estos se crean nodos que sí son vecinos entre sí y se grafican con tamaños más pequeños, de esta manera se garantiza la propiedad de crear grupos dentro del mismo grafo permitiendo que sea relativamente fácil realizar la visita entre todos los nodos, reflejando mejor

el comportamiento de fenómenos reales. Otra propiedad de estos generadores de grafos es que la distancia esperada entre dos nodos elegidos al azar crece de manera proporcional al logaritmo de la cantidad de nodos de la red mientras no se trate de los nodos que están más agrupados, propiedad que detectaron los creadores a partir del comportamiento real de diferentes fenómenos[4].

Para esta tarea de los generadores de grafos empleados en la anterior que fueron los siguientes:

- *grafo de Watts-Strogatz Newman*
- *grafo de Watts-Strogatz*
- *grafo de Watts-Strogatz Conectado*

Se seleccionó de los que menos influían en el tiempo de ejecución el generador de grafo de *Watts-Strogatz Conectado* en combinación con el algoritmo de flujo máximo *Edmons-Karp* de los siguientes algoritmos de flujo máximo empleados:

- *Algoritmo Boykov-Kolmogorov*
- *Algoritmo de Flujo Máximo*
- *Algoritmo Edmonds-Karp*

El algoritmo de *Edmonds-Karp* calcula el flujo máximo de un producto, debe emplearse con grafos dirigidos aunque también se emplea para no dirigidos, requiere asignársele una capacidad sino la toma como infinita [7] y, además, devuelve la red residual de flujo máximo (carácteristica que el algoritmo *Flujo Máximo* no posee, pues este lo que devuelve es un arreglo del flujo máximo que pasa por las aristas seleccionadas) de ahí que se seleccionara el *Edmons-Karp* como algoritmo para calcular el flujo máximo en esta evaluación.

El ejemplo de aplicación que combina generador de grafo aleatorio con algoritmo de flujo máximo seleccionados puede ser la propagación de epidemias endémicas entre regiones determinadas, donde cada vértice serían las regiones objeto de estudios y las aristas el número de epidemias endémicas de cada región que se pueden propagar de un lugar a otro.

A continuación se muestra el fragmento de código desarrollado para generar los grafos con la capacidad asignada apoyándose en [1],[3],[2]:

\cik {_ , _ , _ }

```

1 def Generador_grafo(n,k,p):
2     #S=nx.watts_strogatz_graph(n, k, p)
3     S=nx.connected_watts_strogatz_graph(n,k,p, tries=100, seed=None)
4     scale = 2
5     rang = 11
6     size = S.number_of_edges()
7     e=S.edges(nbunch=None, data=True, default=None)
8     X = truncnorm(a=rang/scale , b=rang/scale , scale=scale).rvs(size=size)
9     X = X.round().astype(int)+rang
10    G=nx.Graph()
11    count=0;
12    for i in e:
13        G.add_edge(i[0],i[1],capacity=X[count])
14        count+=1
15    df = pd.DataFrame()
16    df = nx.to_pandas_adjacency(G, dtype=int , weight='capacity')
17    df.to_csv("grafo5.csv", index=None, header=None)
18    PrintGraph(S,X)
19
20 Generador_grafo(20,7,0.5)

```

Tarea5GeneradorGrafos.py

2.1. Generación de datos para el análisis estadístico

Los grafos se generaron con un código y luego otro los lee y dibuja con el algoritmo de acomodo usado en la tarea anterior y que mejor se ajustaba para esta tarea el *kamada_kawai*. El código usado para ello se muestra a continuación:

```

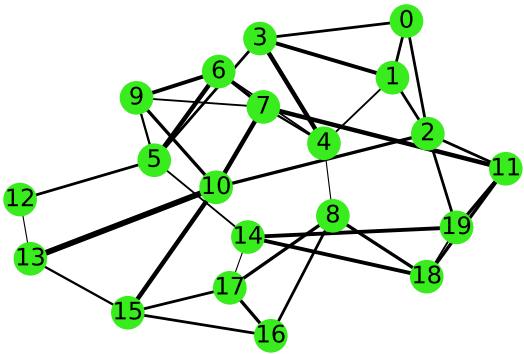
1 def lee_grafo(adress):
2     ds = pd.read_csv(adress, header=None)
3     G = nx.from_pandas_adjacency(ds)
4     return G
5
6 def calcula_tiempo(G,a,b):
7     start_time=time()
8     # fm=nx.maximum_flow(G, a, b,capacity="weight")
9     for i in range(100):
10         fm=edmonds_karp(G, a, b,capacity="weight")
11         print(i)
12     time_elapsed = time() - start_time
13
14     return time_elapsed
15
16 def DibujaGrafo(G):
17
18     pesos=[]
19     for edge in G.edges():
20         pesos.append(G.edges[edge]['weight'])
21     pesos[:] = [x/7*x/10 for x in pesos]
22
23     # pos=nx.spring_layout(G)
24     pos=nx.kamada_kawai_layout(G, scale=10)
25
26     nx.draw_networkx_nodes(G, pos, node_size=400, node_color="#38ec1d", node_shape='o')
27     nx.draw_networkx_edges(G, pos, width=pesos, edge_color='black')
28     labels = {}
29     for i in G.nodes:
30         labels[i]=str(i)
31     nx.draw_networkx_labels(G, pos, labels, font_size=15)
32
33     plt.axis('off')
34     plt.savefig("fig5a.png", dpi=600)
35     plt.savefig("fig5a.eps", dpi=600)
36     df=pd.DataFrame(pos)
37     df.to_csv("pos_grafo5.csv", index=None, header=None)
38     return pos

```

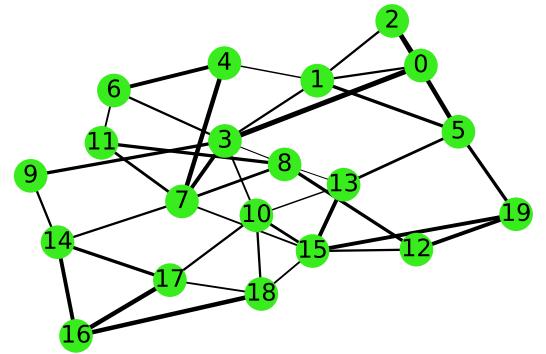
T5ProTimeFLujo.py

Los grafos generados se muestran en la figura 1 de la página 5

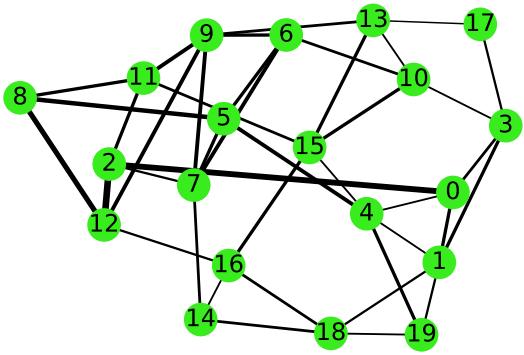
Luego se calcularon las propiedades para cada vértice de cada grafo usando el siguiente código:



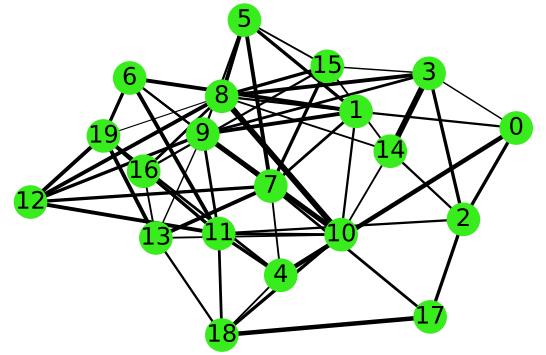
(a) Grafo 1



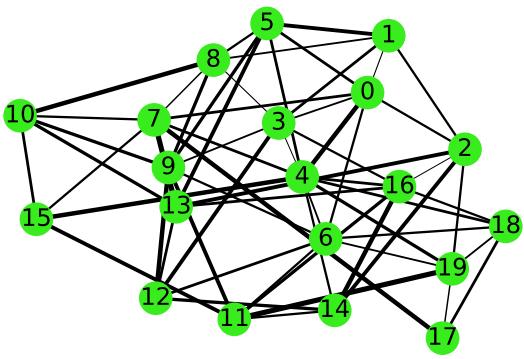
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Se ometió

```

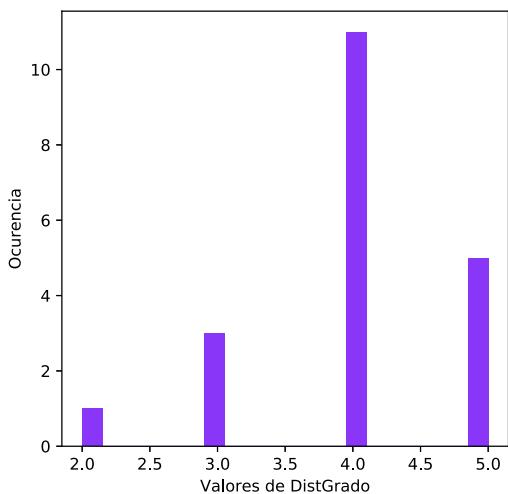
1 def lee_grafo(nombre):
2     ds = pd.read_csv(nombre, header=None)
3     G = nx.from_pandas_adjacency(ds)
4     return G
5
6 def lee_propiedades(nombre):
7     ds = pd.read_csv(nombre)
8     return ds
9
10 i=[ "grafo1.csv" , "grafo2.csv" , "grafo3.csv" , "grafo4.csv" , "grafo5.csv" ]
11 g=[]
12 for x in i:
13     G=lee_grafo(x)
14
15 dic={}
16 Nodes=G.nodes;
17 dic[ "Nodo" ]=Nodes
18 dic[ "DistGrado" ]=[G.degree(i) for i in Nodes]
19 dic[ "CoefAgrup" ]=[nx.clustering(G,i) for i in Nodes]
20 dic[ "CentCercania" ]=[nx.closeness_centrality(G,i) for i in Nodes]
21 dic[ "CentCarga" ]=[nx.load_centrality(G,i) for i in Nodes]
22 dic[ "ExcentCarga" ]=[round(nx.eccentricity(G,i), 2) for i in Nodes]
23 PageR=nx.pagerank(G,weight="capacity")
24 dic[ "PageR" ]=[PageR[i] for i in Nodes]
25 df=pd.DataFrame(dic)
26 df.to_csv("propiedades"+str(x)+".csv", index=None)
27 g.append("propiedades"+str(x)+".csv")
28
29 print(g)
30
31 j=[ "DistGrado" , "CoefAgrup" , "CentCercania" , "CentCarga" , "ExcentCarga" , "PageR" ]
32 for y in g:
33     H=lee_propiedades(y)
34     for i in j:
35         fig = plt.figure(figsize=(5, 5))
36         ax = fig.add_subplot(1, 1, 1)
37         his = ax.hist(H[i], bins=len(H[j]), facecolor="#8a36f8", alpha=0.75)
38         ax.set_xlabel("Valores de " + i)
39         ax.set_ylabel("Ocurrencia")
40     #     plt.savefig("histograma"+ y + i + ".png", bbox_inches="tight", dpi=100)
41     #     plt.savefig("histograma"+ y + i + ".eps", bbox_inches="tight", dpi=100)
42     #     plt.show()
43
44 print("fin")

```

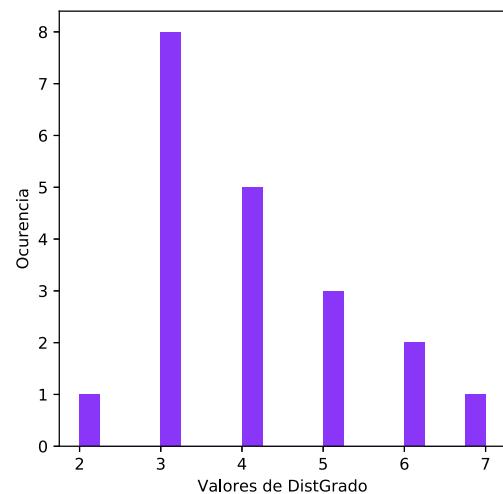
T5histoporopiedad.py

Para tener un avance del comportamiento de estas propiedades de los vértices las cuales se emplearan en el análisis de varianza más adelante, para cada grafo se realizaron los histogramas de cada una de las propiedades (distribución de grado, coeficiente de agrupamiento, centralidad de cercanía, centralidad de carga, excentricidad y *pagerank*) por cada grafo. Los mismos se muestran a continuación:

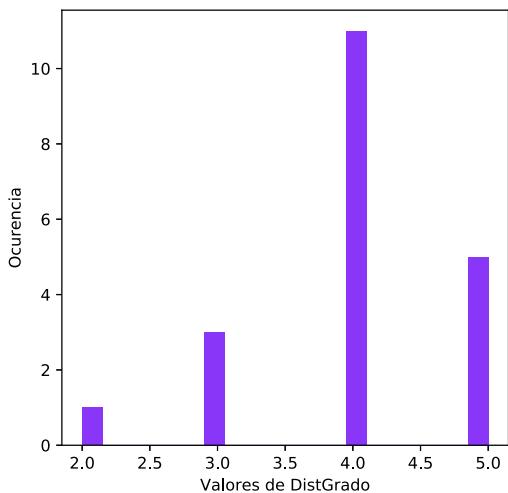
Análisis del comportamiento de la distribución de grado en cada grafo. Ver figura 2 de la página 7



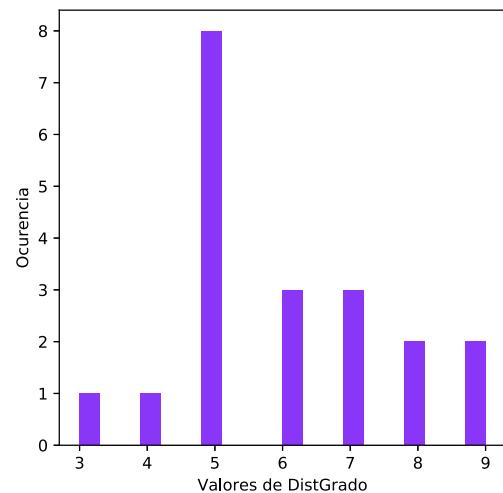
(a) *Grafo 1*



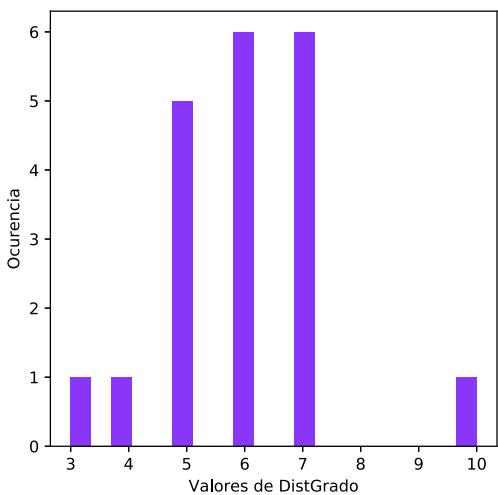
(b) *Grafo 2*



(c) *Grafo 3*



(d) *Grafo 4*



(e) *Grafo 5*

Figura 2: Histogramas de distribución de grado por grafo

Análisis del comportamiento del coeficiente de agrupamiento. Ver figura 3 de la página 9

Análisis del comportamiento de la centralidad de cercanía en cada grafo. Ver figura 4 de la página 10

Análisis del comportamiento de la centralidad de carga en cada grafo. Ver figura 5 de la página 11

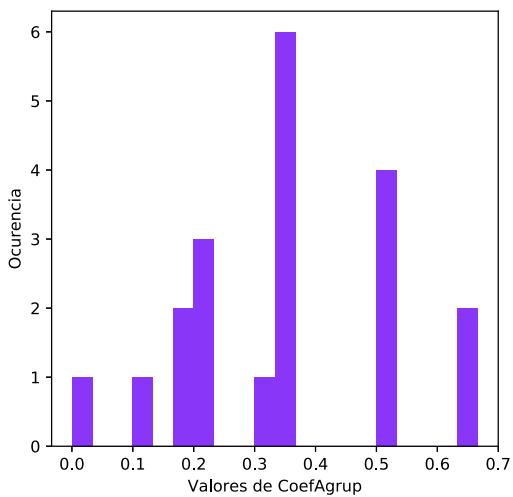
Análisis del comportamiento de la excentricidad en cada grafo. Ver figura 6 de la página 12

Análisis del comportamiento del *pagerank* en cada grafo. Ver figura 7 de la página 13

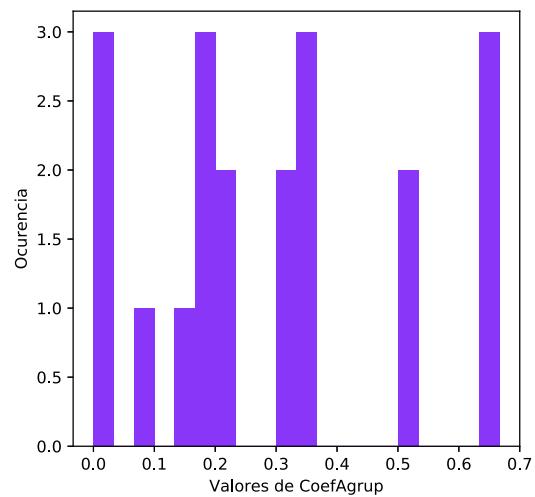
Al observar el comportamiento de estas variables se aprecia que no describen una distribución normal, por lo que no es recomendable hacer el análisis de varianza de influencia en el tiempo de ejecución del algoritmo y el flujo máximo usando el promedio, sino la mediana.

Para realizar el cálculo del tiempo de ejecución del algoritmo y el flujo máximo variando los nodos fuentes y sumideros de modo que toda la información quedase guardada en un .csv que pudiera emplearse en el análisis de varianza (ANOVA) se usó el siguiente código:

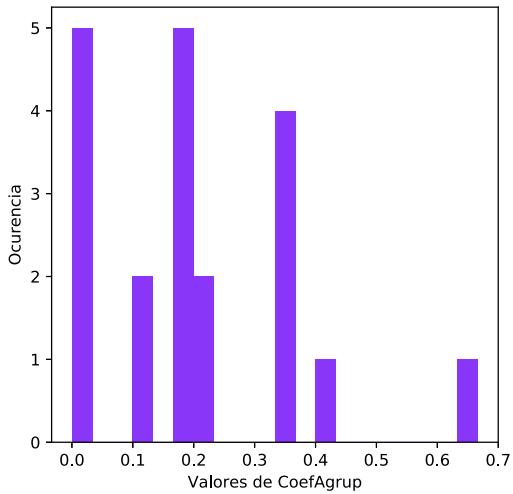




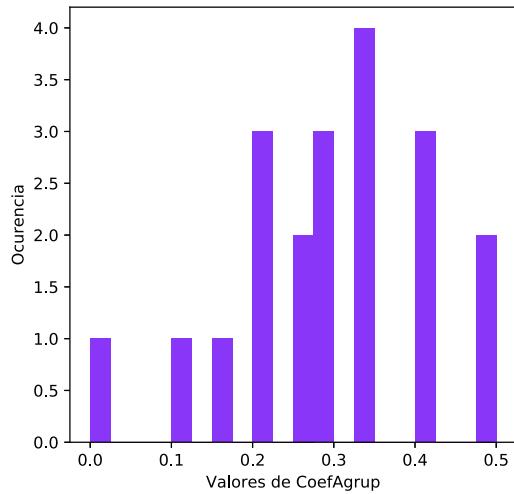
(a) *Grafo 1*



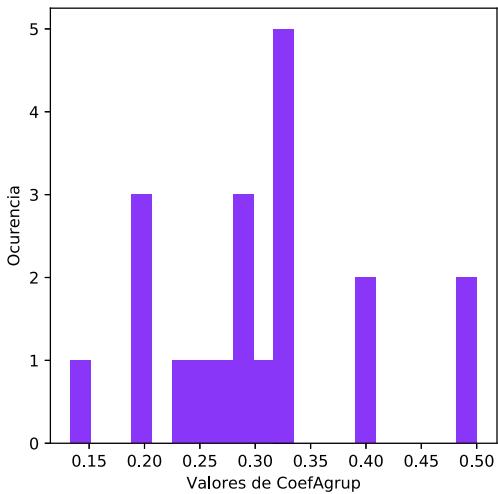
(b) *Grafo 2*



(c) *Grafo 3*



(d) *Grafo 4*



(e) *Grafo 5*

Figura 3: Histogramas del coeficiente de agrupamiento por grafo

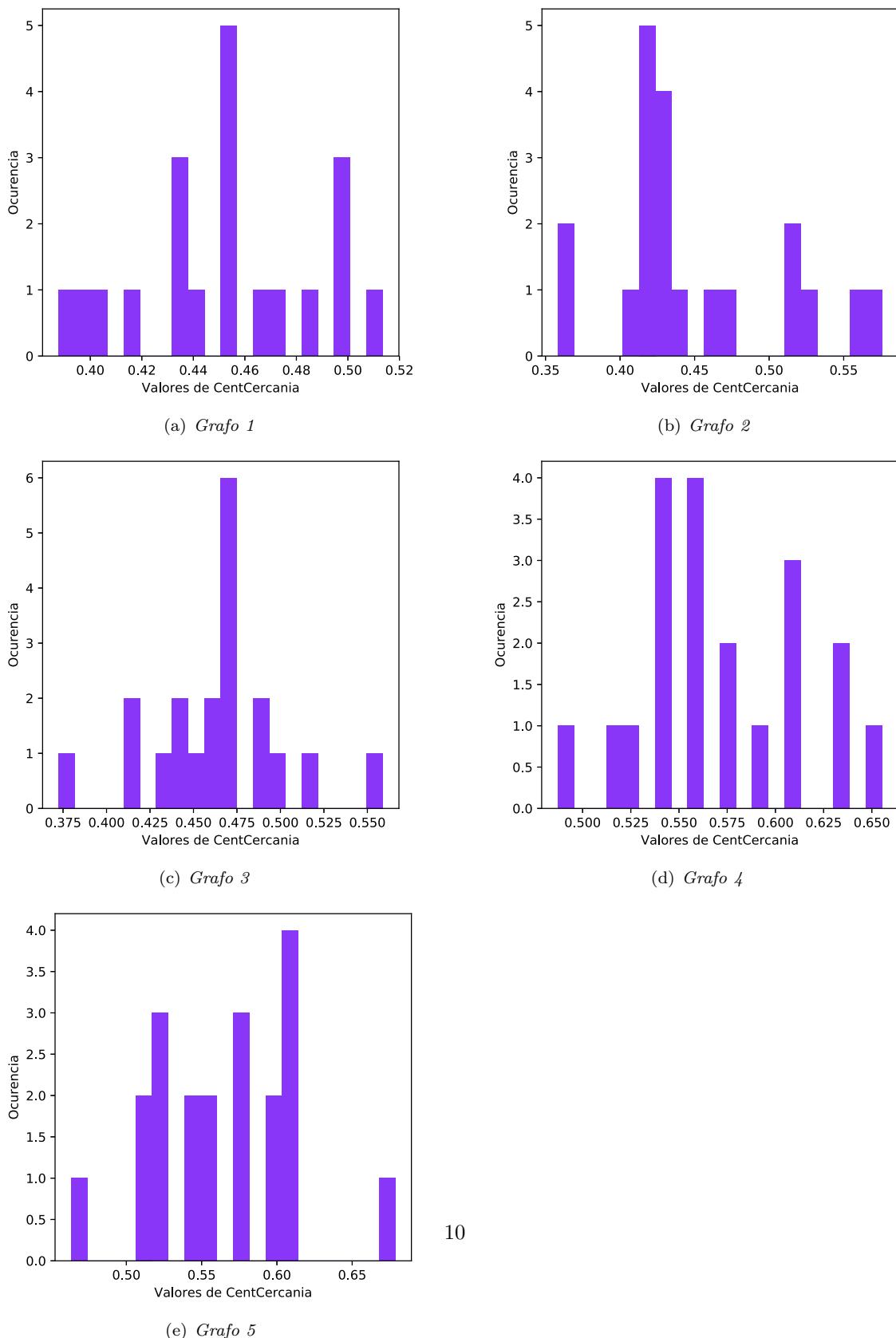


Figura 4: Histogramas de la centralidad de cercanía

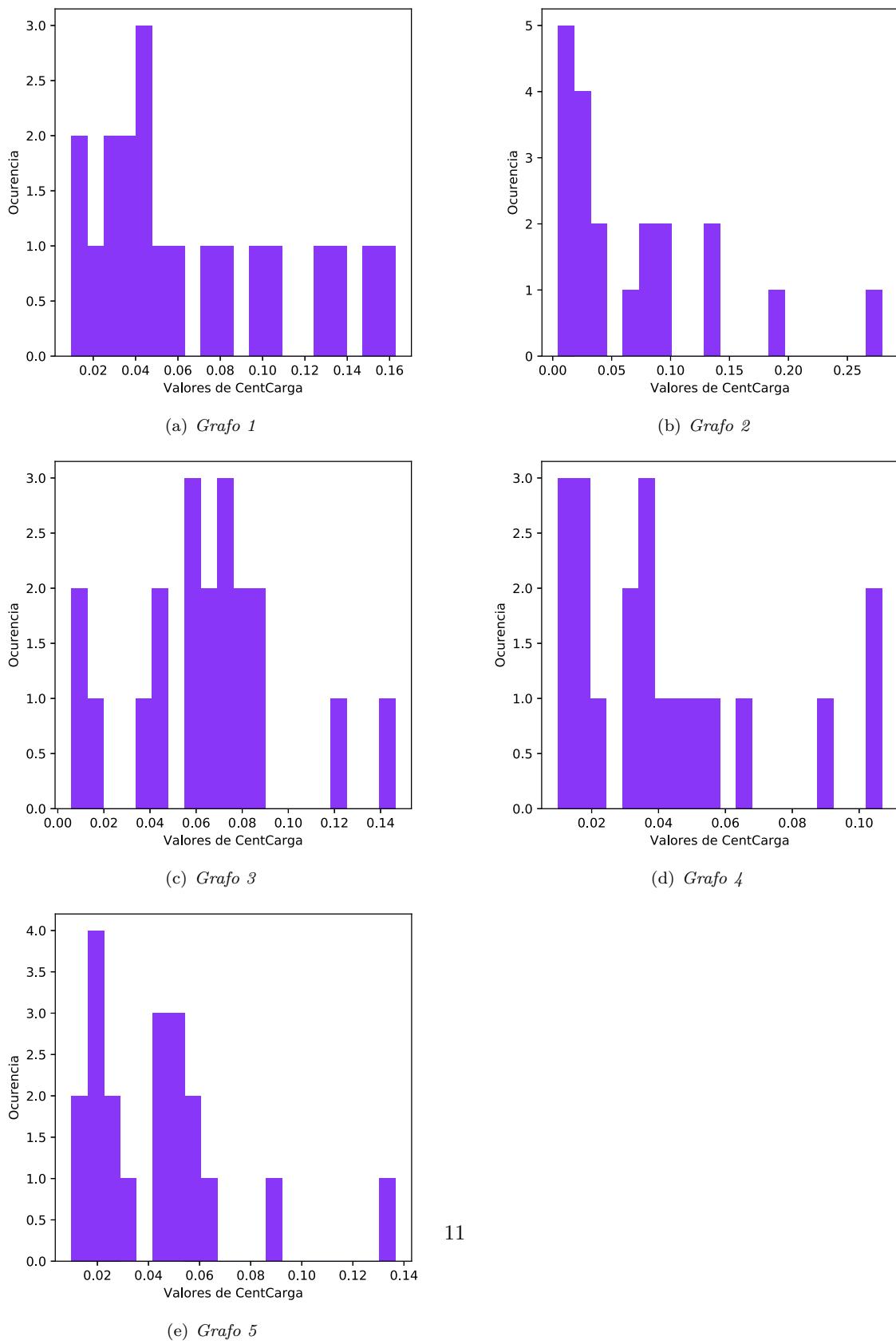
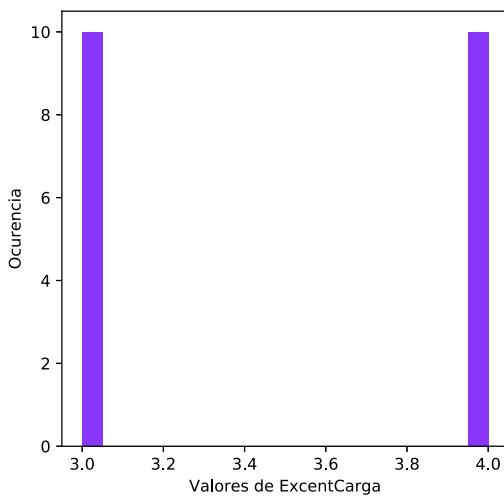
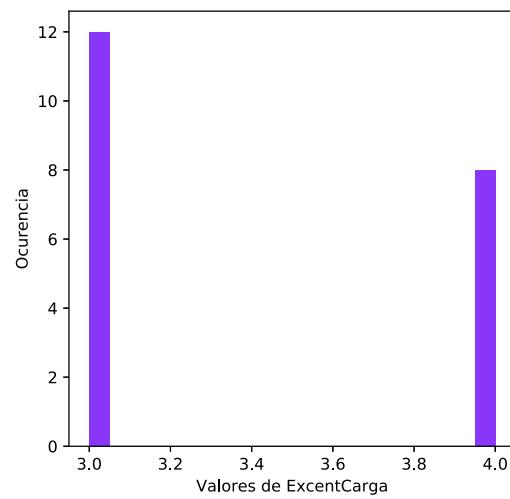


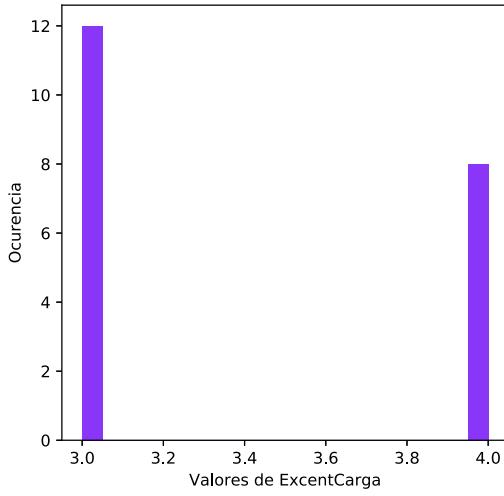
Figura 5: Histogramas de la centralidad de carga



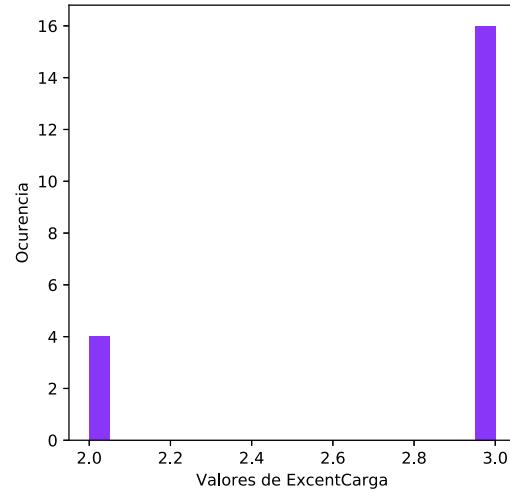
(a) *Grafo 1*



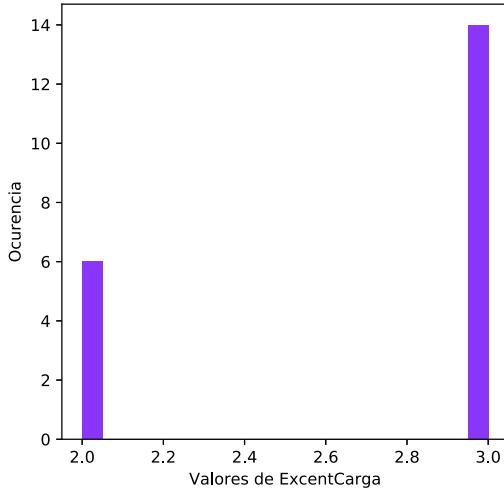
(b) *Grafo 2*



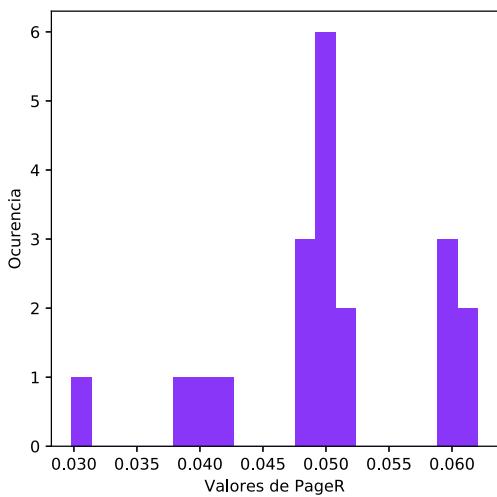
(c) *Grafo 3*



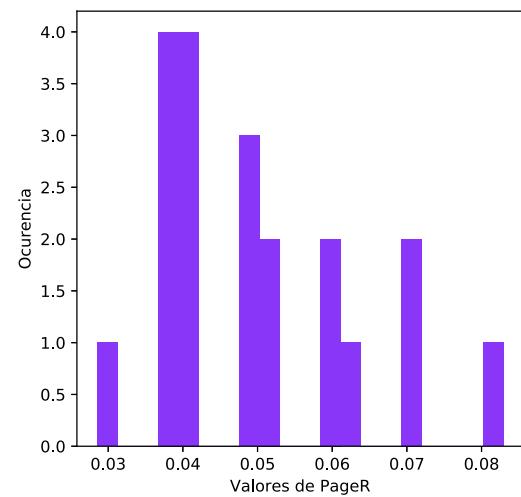
(d) *Grafo 4*



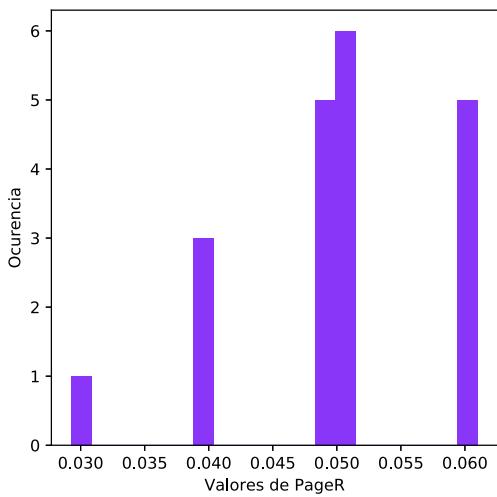
(e) *Grafo 5*



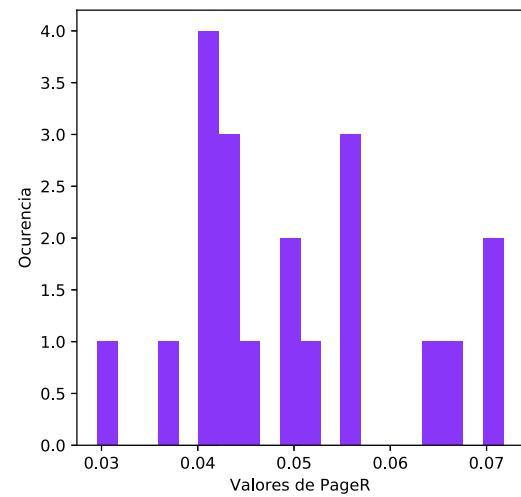
(a) *Grafo 1*



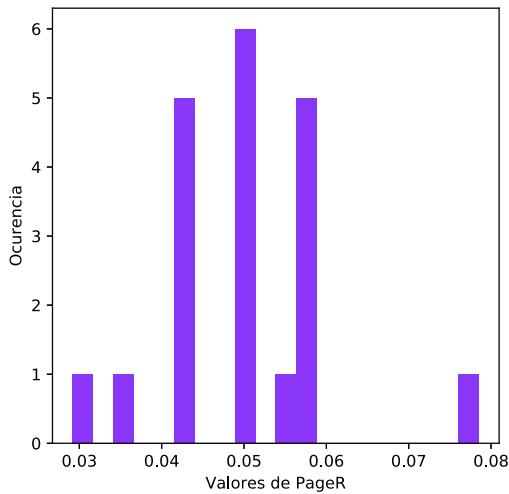
(b) *Grafo 2*



(c) *Grafo 3*



(d) *Grafo 4*



(e) *Grafo 5*

Figura 7: Histogramas del *pagerank*

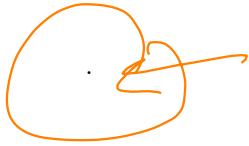
```

1 def calcula_tiempo(G,a,b):
2     start_time=time()
3     # fm=nx.maximum_flow(G, a, b,capacity="weight")
4     for i in range(100):
5         fm=edmonds_karp(G, a, b,capacity="weight")
6         print(i)
7     time_elapsed = time() - start_time
8
9     return time_elapsed
10
11
12 def Todo(G,nombre):
13     dic={"Grafo":[] ,
14         "Fuente":[] ,
15         "Sumidero":[] ,
16         "Mediatime":[] ,
17         "Medianatime":[] ,
18         "Stdtime":[] ,
19         "Flujomaximo":[] ,
20         "DistGrado":[] ,
21         "CoefAgrup":[] ,
22         "CentCercania":[] ,
23         "CentCarga":[] ,
24         "ExcentCarga":[] ,
25         "PageR":[] }
26
27 Nodes=G.nodes;
28 for i in Nodes:
29     for j in Nodes:
30         if i!=j:
31             t=[]
32             for k in range(10):
33                 t.append(calcula_tiempo(G,i,j))
34             dic["Grafo"].append(nombre)
35             dic["Fuente"].append(i)
36             dic["Sumidero"].append(j)
37             dic["Mediatime"].append(np.mean(t))
38             dic["Medianatime"].append(np.median(t))
39             dic["Stdtime"].append(np.std(t))
40             dic["Flujomaximo"].append(nx.maximum_flow_value(G,i,j,capacity="weight"))
41
42             dic["DistGrado"].append(G.degree(i))
43             dic["CoefAgrup"].append(nx.clustering(G,i))
44             dic["CentCercania"].append(nx.closeness_centrality(G,i))
45             dic["CentCarga"].append(nx.load_centrality(G,i))
46             dic["ExcentCarga"].append(nx.eccentricity(G,i))
47             PageR=nx.pagerank(G,weight="capacity")
48             dic["PageR"].append(PageR[i])
49
50
51 df=pd.DataFrame(dic)
52 df.to_csv("CSVunido.csv", index=None, mode="a")
53
54 i=[ "grafo1", "grafo2", "grafo3", "grafo4", "grafo5"]
55 for x in i:

```

```
57     print (x)
58 G=lee_grafo(x)
59 Todo(G,x)
```

T5Paraunircsvfinal.py



3. Influencia en el óptimo al variar nodos fuentes y sumideros.

Al variar los nodos fuentes y sumideros fue calculado el valor de flujo máximo que se obtenía en cada caso, evidenciándose que existe variación en el óptimo para cada uno de los cinco grafos. A continuación se muestra para cada uno de los cinco grafos generados inicialmente el grafo inicial con la propuesta de mejor y peor fuente y sumidero. La fuente aparece con color rojo y el sumidero con color azul. De igual modo se emplea el degradado de color rojo para representar con rojo más intenso la mayor cantidad de flujo y más claro la menor cantidad de flujo que pasa por las aristas de la red residual que devuelve el algoritmo de flujo máximo seleccionado.

En la figura 8 de la página 16 se observa la variación realizada para el grafo uno, en la figura 9 de la página 17 la variación del grafo dos, en la figura 10 de la página 18 la variación del grafo tres, en la figura 11 de la página 19 la variación del grafo cuatro y figura 12 de la página 20 la variación del grafo cinco.

Seguidamente se muestra el código empleado en esta sección

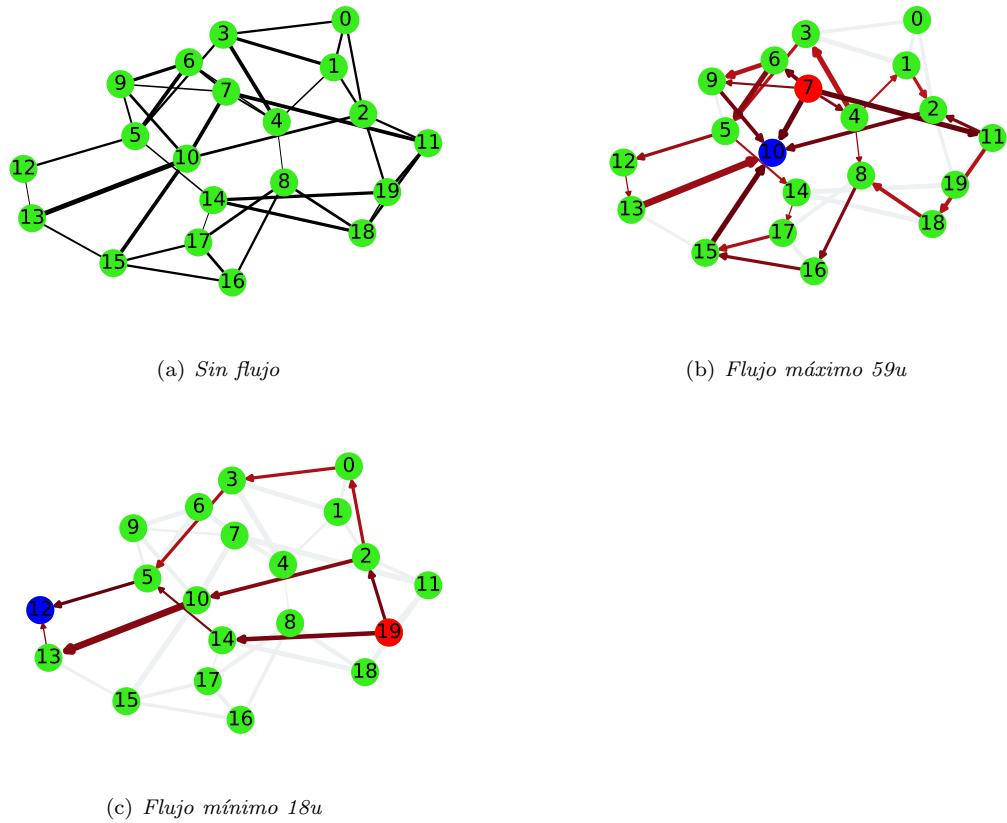


Figura 8: Variación de fuente y sumidero en grafo 1

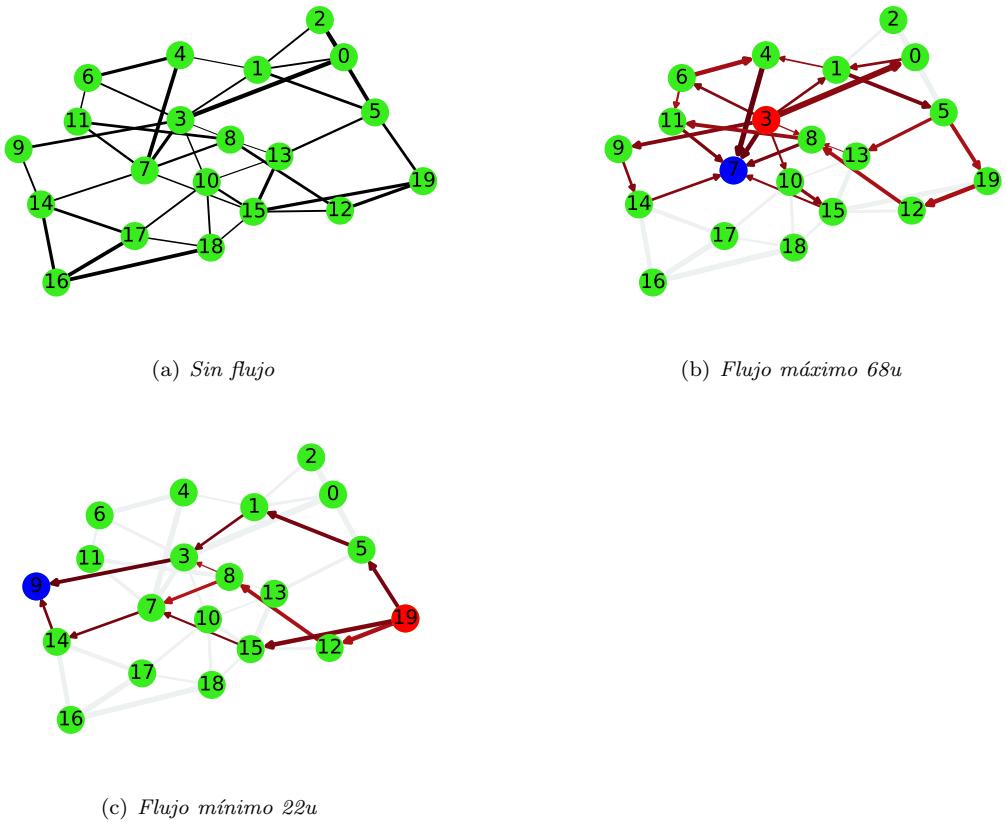


Figura 9: Variación de fuente y sumidero en grafo 2

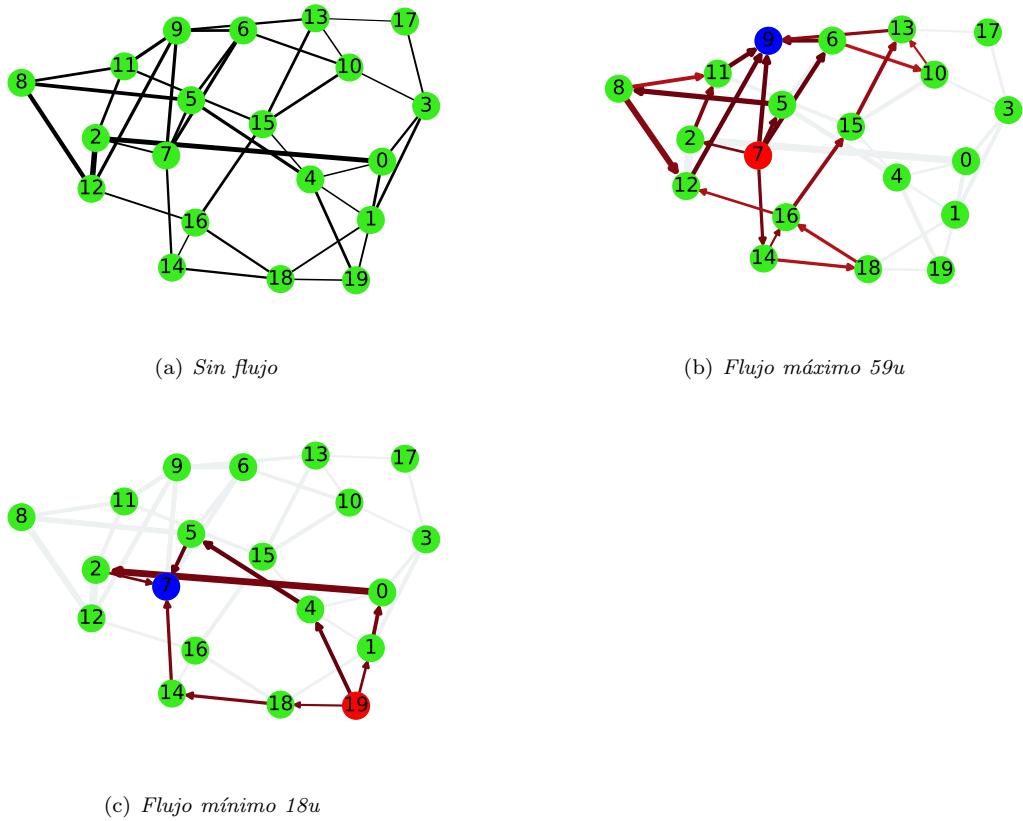


Figura 10: Variación de fuente y sumidero en grafo 3

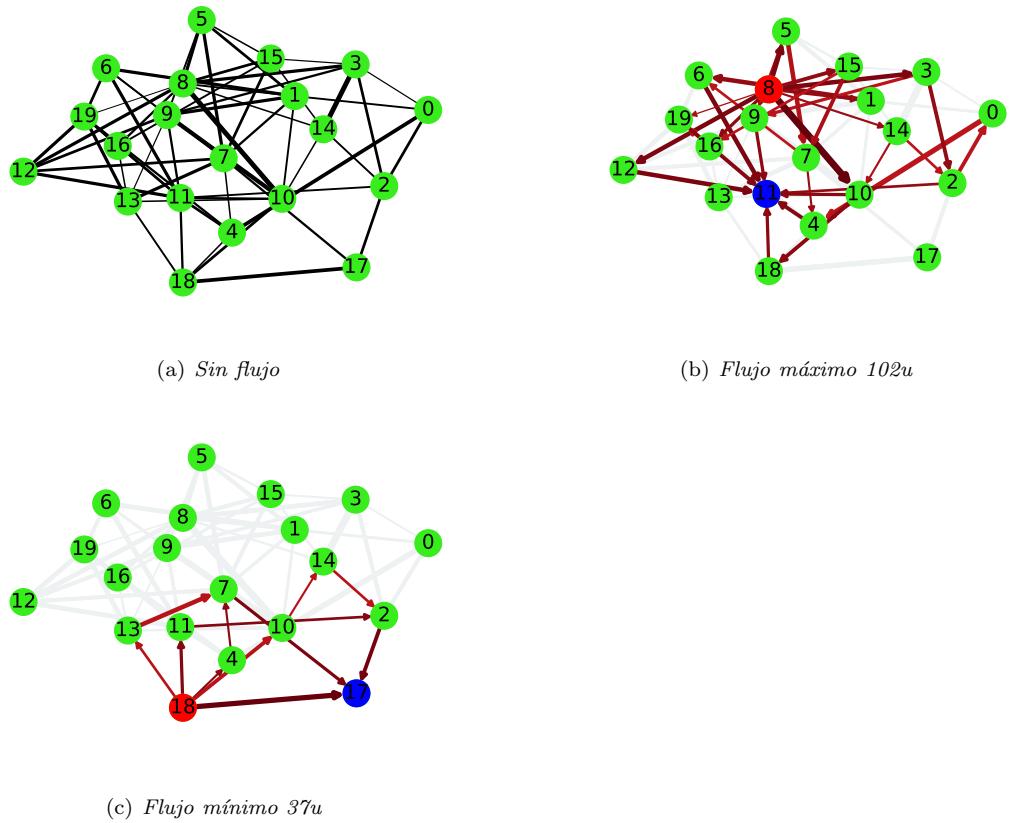


Figura 11: Variación de fuente y sumidero en grafo 4

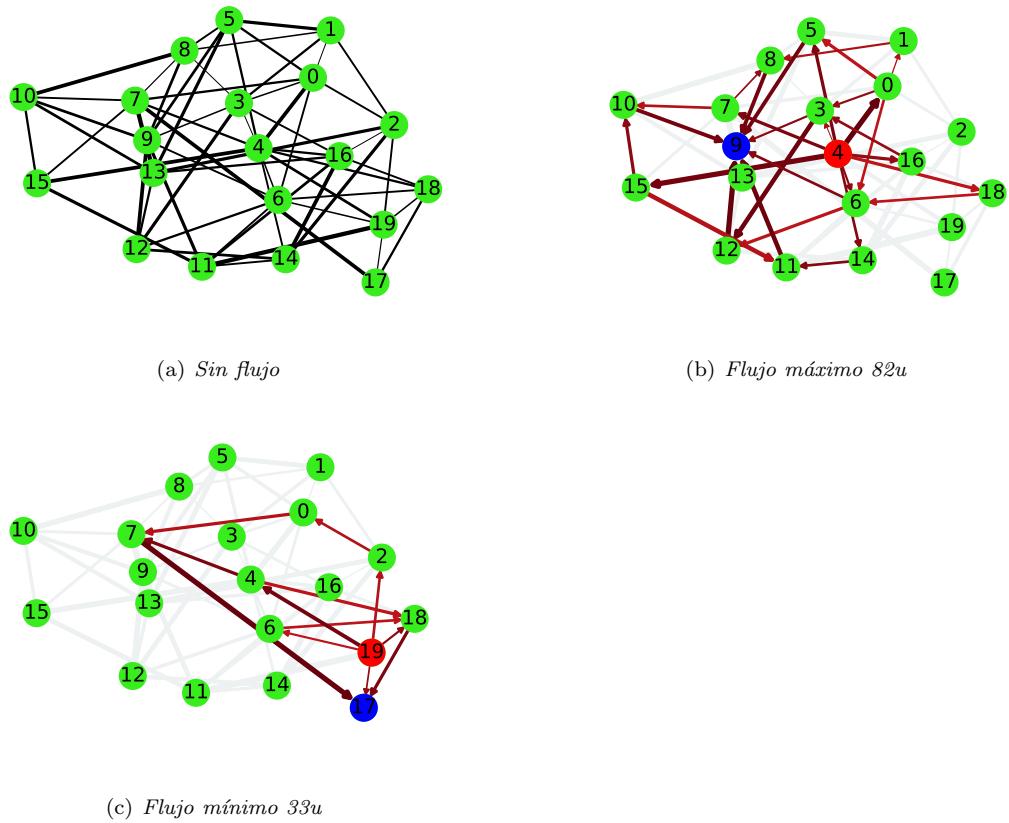


Figura 12: Variación de fuente y sumidero en grafo 5

```

1 def Todo(G):
2     dic={"Fuente":[] ,
3         "Sumidero":[] ,
4         "Mediatime":[] ,
5         "Medianatime":[] ,
6         "Stdtime":[] ,
7         "Flujomaximo":[] ,
8         "DistGrado":[] ,
9         "CoefAgrup":[] ,
10        "CentCercania":[] ,
11        "CentCarga":[] ,
12        "ExcentCarga":[] ,
13        "PageR":[] }
14
15 Nodes=G.nodes;
16 for i in Nodes:
17     for j in Nodes:
18         if i!=j:
19             t=[]
20             for k in range(10):
21                 t.append(calcula_tiempo(G,i,j))
22             dic["Fuente"].append(i)
23             dic["Sumidero"].append(j)
24             dic["Mediatime"].append(np.mean(t))
25             dic["Medianatime"].append(np.median(t))
26             dic["Stdtime"].append(np.std(t))
27             dic["Flujomaximo"].append(nx.maximum_flow_value(G,i,j,capacity="weight"))
28
29             dic["DistGrado"].append(G.degree(i))
30             dic["CoefAgrup"].append(nx.clustering(G,i))
31             dic["CentCercania"].append(nx.closeness_centrality(G,i))
32             dic["CentCarga"].append(nx.load_centrality(G,i))
33             dic["ExcentCarga"].append(nx.eccentricity(G,i))
34             PageR=nx.pagerank(G,weight="capacity")
35             dic["PageR"].append(PageR[i])
36
37
38 df=pd.DataFrame(dic)
39 df.to_csv("Todosvaloresg5.csv", index=None)
40
41 def DibujarRes(G,pos,fuentes,sumideros):
42 #    pos=nx.spring_layout(R)
43 sinflujo=[]
44 flowconflujo=[]
45 conflujo=[]
46 sinflujopesos=[]
47 conflujopesos=[]
48 maxi=0
49 for edge in G.edges():
50     if G.edges[edge]['flow']==0:
51         sinflujo.append(edge)
52         sinflujopesos.append(G.edges[edge]['capacity'])
53     elif G.edges[edge]['flow']>0:
54         conflujo.append(edge)
55         conflujopesos.append(G.edges[edge]['capacity'])
56         flowconflujo.append(G.edges[edge]['flow'])

```

```

57     flowconfluo [:] = [x+50 for x in flowconfluo]
58     for i in flowconfluo:
59         if i>maxi:
60             maxi=i
61     sinfluojopesos [:] = [x/10*x/5 for x in sinfluojopesos]
62     confluojopesos [:] = [x/10*x/5 for x in confluojopesos]
63     nx.draw_networkx_nodes(G, pos, node_size=400, node_color='#38ec1d', node_shape='o')
64     nx.draw_networkx_nodes(G, pos, nodelist=fuentes, node_size=400, node_color='r', node_shape='o')
65     nx.draw_networkx_nodes(G, pos, nodelist=sumideros, node_size=400, node_color='b', node_shape='o')
66     nx.draw_networkx_edges(G, pos, edgelist=sinflujo, edge_color='#eef1f2', width=sinfluojopesos, arrows=False)
67     nx.draw_networkx_edges(G, pos, edgelist=conflujo, edge_cmap=plt.cm.Reds, width=confluojopesos, edge_color=flowconfluo, edge_vmin=0,edge_vmax=maxi)
68     labels = {}
69     for i in G.nodes:
70         labels[i]=str(i)
71     nx.draw_networkx_labels(G, pos, labels, font_size=15)
72     plt.axis('off')
73     plt.savefig("fig5b1.png",dpi=600)
74     plt.savefig("fig5b1.eps",dpi=600)
75
76
77 G=lee_grafo("grafo5.csv")
78 GeneraPropiedades(G)
79 DibujaGrafo(G)
80 Todo(G)
81
82 R=edmonds_karp(G,4,9, capacity="weight")
83 pos = pd.read_csv("pos_grafo5.csv", header=None)
84 DibujarRes(R, pos,[4],[9])

```

T5ProTimeFLujo.py

4. Análisis de varianza (ANOVA) y prueba de *Tukey* para determinar la relación entre las propiedades de los nodos y las variables dependientes estudiadas.

Para realizar el análisis del comportamiento de las variables dependientes *tiempo de ejecución* y *flujo máximo* con respecto a cada factor a analizar se realizó un análisis de varianza (ANOVA) para cada factor.

En el caso de las propiedades que se usan como factores en el ANOVA se realizó un histograma para cada una de ellas de modo que se pudiera llevar a escalas los valores de las mismas.

Los histogramas se emplean para establecer los rangos de valores asignándoles una etiqueta de bajo, medio y alto para cada propiedad.

El resultado del análisis ANOVA con respecto al tiempo de ejecución para cada propiedad se muestra en los cuadros del uno al seis. Segundo los resultados en todas las propiedades se rechaza la hipótesis nula por lo que todas influyen en el tiempo excepto la centralidad de carga de rango alto, en la que según los resultados obtenidos se acepta la hipótesis nula. Esto se puede corroborar al analizar los diagramas de cajas de cada una de las propiedades. Ver figura 13 de la página 24. Del análisis de los resultados se aprecia que el que menos influye es la centralidad de carga, como se observa también en los diagramas de caja y prueba de Tukey (ver figura 14 de la página 25) el resto de las propiedades sí influyen en el tiempo de ejecución del algoritmo al variar los nodos fuentes y sumideros.

Source	Cuadro 1: ANOVA Centralidad de Carga					
	SS	DF	MS	F	p-unc	np2
CentCarga	0	2	0	0.058	0.94360248	0
Within	0.126	1897	0	-	-	-

Source	Cuadro 2: ANOVA Centralidad de cercanía					
	SS	DF	MS	F	p-unc	np2
CentCercania	0.024	2	0.012	221.416	3.76E-87	0.189
Within	0.102	1897	0	-	-	-

Source	Cuadro 3: ANOVA Coeficiente de agrupamiento					
	SS	DF	MS	F	p-unc	np2
CoefAgrup	0.007	2	0.004	56.285	1.79E-24	0.056
Within	0.119	1897	0	-	-	-

Los resultados Tukey de este análisis se muestran en la figura 14 de la página 25

Al realizar el análisis ANOVA para verificar la influencia de las propiedades en el flujo máximo se obtuvieron los resultados que se muestran en los cuadros del 7 al 12. En los mismos se aprecia que se rechaza la hipótesis nula en todos los casos excepto en la propiedad de centralidad de carga en la que se acepta la hipótesis nula. Se incluyen los diagramas de caja (ver figura 15 de la página 27) y las pruebas de Tukey para profundizar en los resultados (ver figura 16 de la página 28). Se observa que la propiedad que menos influye es la centralidad de carga para valores altos, el resto de las propiedades sí influye al variar los nodos fuentes y sumideros en el valor de flujo máximo.

Referencias

- [1] Leonardo J. Caballero. *Materiales del entrenamiento de programación en Python*. 2019.
- [2] Desarrolladores de Networkx. Add-edge. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.MultiDiGraph.add_edge.html. Accessed:2019-04-19.

Source	Cuadro 4: ANOVA Distribución de Grado					
	SS	DF	MS	F	p-unc	np2
DistGrado	0.023	2	0.011	209.861	4.61E-83	0.181
Within	0.103	1897	0	-	-	-

Source	Cuadro 5: ANOVA Excentricidad					
	SS	DF	MS	F	p-unc	np2
ExcentCarga	0.013	2	0.006	104.776	6.90E-44	0.099
Within	0.113	1897	0	-	-	-

Source	Cuadro 6: ANOVA <i>PageRange</i>					
	SS	DF	MS	F	p-unc	np2
PageR	0.01	2	0.005	82.216	5.72E-35	0.08
Within	0.116	1897	0	-	-	-

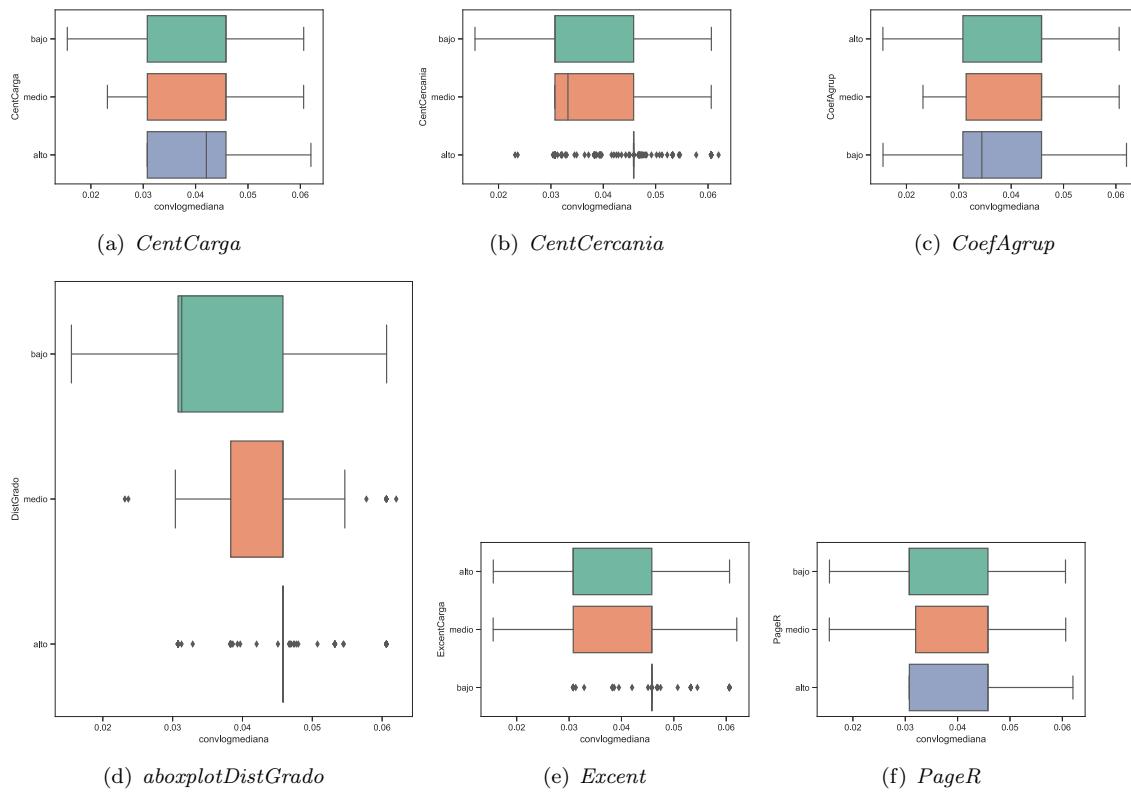


Figura 13: Diagrama de caja del ANOVA con Tiempo

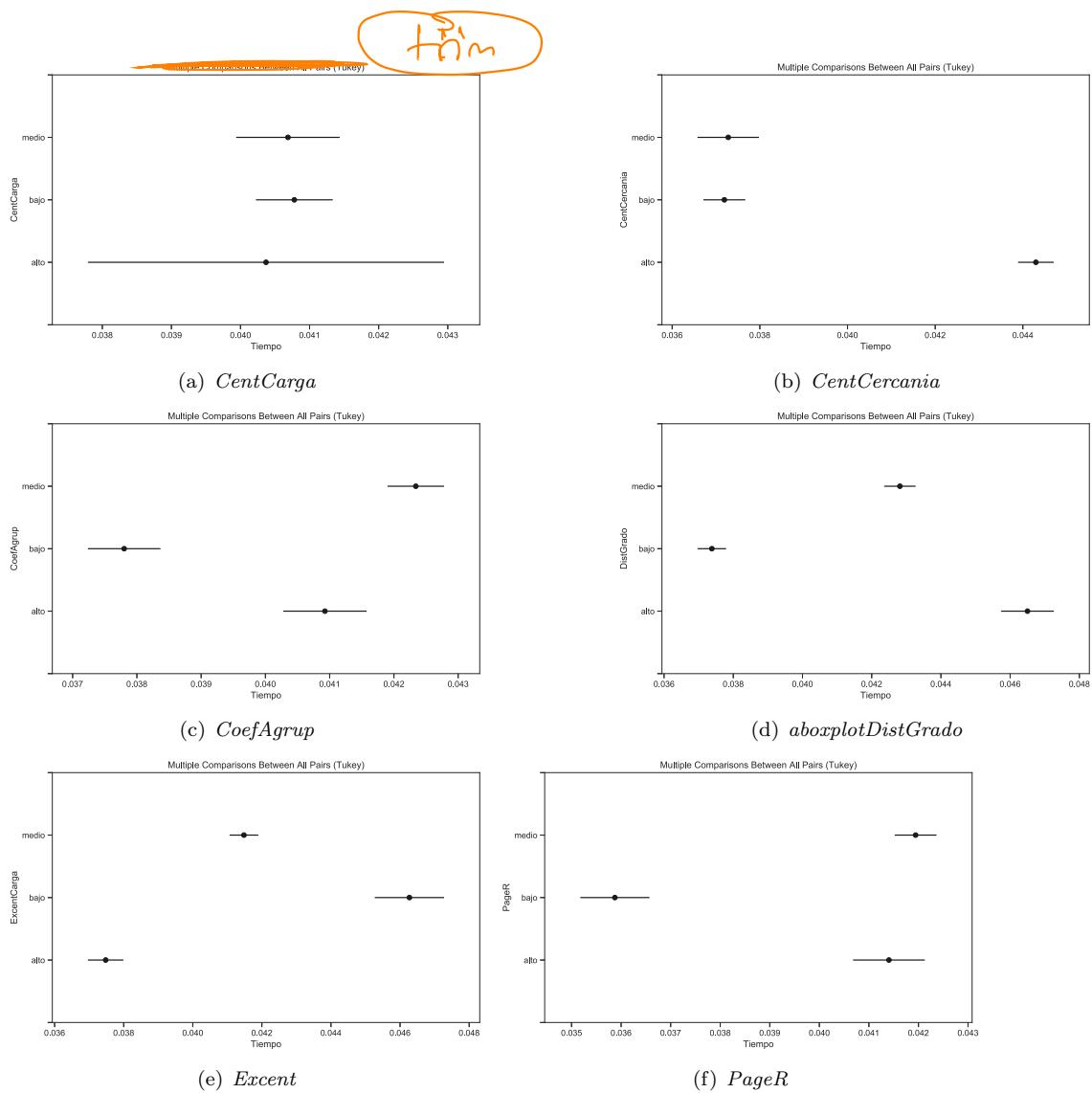


Figura 14: Diagrama de TUKEY para ANOVA con Tiempo

Cuadro 7: ANOVA flujo max CentCarga

Source	SS	DF	MS	F	p-unc	np2
CentCarga	0.471	2	0.236	2.473	0.08461665	0.003
Within	180.834	1897	0.095	-	-	-

Cuadro 8: ANOVA flujo max CentCercanía

Source	SS	DF	MS	F	p-unc	np2
CentCercania	68.685	2	34.343	578.476	7.16E-197	0.379
Within	112.62	1897	0.059	-	-	-

Cuadro 9: ANOVA flujo max CoefAgrup

Source	SS	DF	MS	F	p-unc	np2
CoefAgrup	19.784	2	9.892	116.181	2.53E-48	0.109
Within	161.521	1897	0.085	-	-	-

Cuadro 10: ANOVA flujo max DistGrado

Source	SS	DF	MS	F	p-unc	np2
DistGrado	68.851	2	34.426	580.732	1.77E-197	0.38
Within	112.454	1897	0.059	-	-	-

Cuadro 11: ANOVA flujo max Excent

Source	SS	DF	MS	F	p-unc	np2
ExcentCarga	54.856	2	27.428	411.472	3.69E-149	0.303
Within	126.45	1897	0.067	-	-	-

Cuadro 12: ANOVA flujo max PageR

Source	SS	DF	MS	F	p-unc	np2
PageR	40.988	2	20.494	277.069	2.70E-106	0.226
Within	140.317	1897	0.074	-	-	-

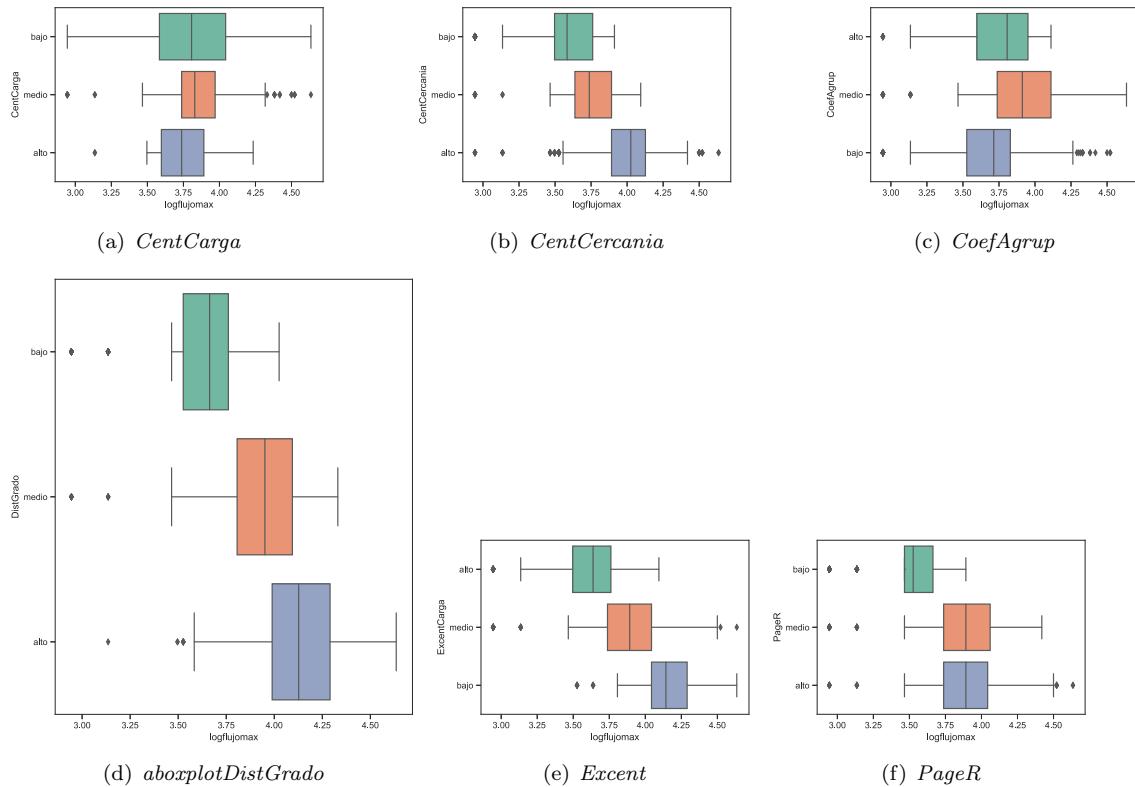


Figura 15: Diagrama de caja del ANOVA con flujo máximo

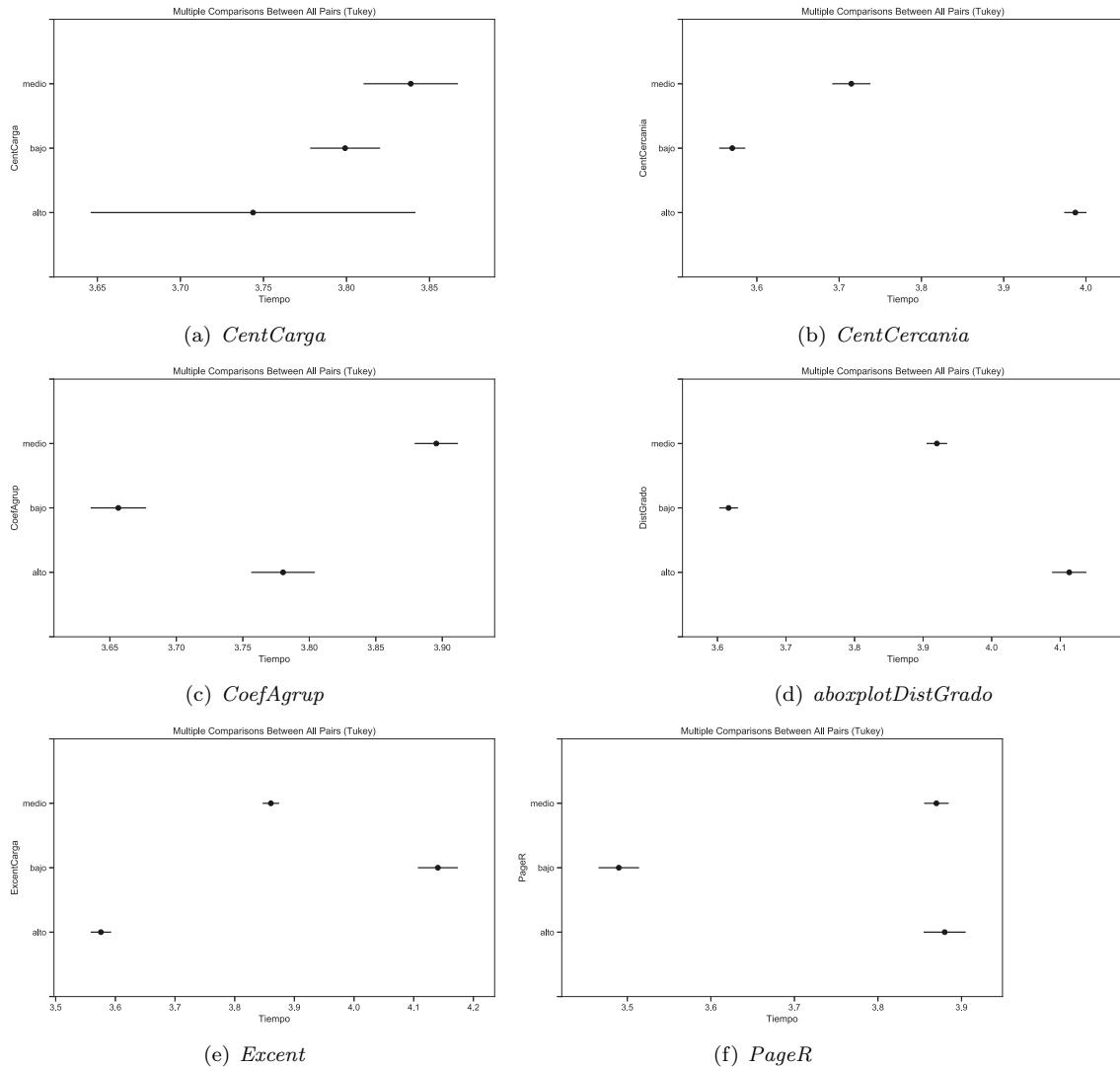
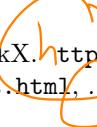


Figura 16: Diagrama de TUKEY para ANOVA con flujo máximo

- [3] Desarrolladores de Scipy.Org. Función estadística `scipy.stats.truncnorm`. <https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.truncnorm.html>. Accessed:2019-04-19.
- [4] Steven H. Strogatz  Duncan J. Watts. Collective dynamics of ‘small-world’ networks. *Nature*, 393:440–442, jun. 1998. doi: 10.1038/30918. URL <https://worrydream.com/refs/Watts-CollectiveDynamicsOfSmallWorldNetworks.pdf>.
- [5] Pedro A. Solares Hernández. *Redes aleatorias de pequeño mundo y libres de escalas*. Universidad Politécnica de Valencia, 2017. Accessed:2019-04-26.
- [6] Valerie De la Cruz.  what are the applications of random graphs? develop python with pycharm. <https://www.quora.com/What-are-the-applications-of-random-graphs>, 2018.
- [7] Desarrolladores NetworkX.  https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.flow.edmonds_karp.html, . Accessed:2019-03-31.
- [8] Desarrolladores NetworkX.  <https://networkx.github.io/documentation/stable/reference/generators.html>, . Accessed: 2019-03-29.

Tarea No.5: Rectificada

Dayli Machado (5275)

3 de junio de 2019

1. Objetivo

Determinar mediante un diseño de experimento, empleando el análisis de varianza de un factor y otras pruebas estadísticas, la influencia que pueden tener en las variables dependientes *tiempo de ejecución del algoritmo de flujo máximo seleccionado y el máximo flujo posible a obtener para diferentes combinaciones de fuentes y sumideros*, en función de las siguientes características estructurales de cada vértice: distribución de grado, coeficiente de agrupamiento, centralidad de cercanía, centralidad de carga, excentricidad y *page rank*.

2. Generador de grafos y algoritmo de flujo máximo seleccionado

Para la evaluación anterior se seleccionaron los generadores de grafos aleatorios los cuales son útiles para comprender los procesos estocásticos que ocurren en una red [7]. Adicionalmente plantea que la generación de grafos aleatorios que controlan algunas condiciones, como la distribución de grado, el coeficiente de agrupamiento y la secuencia de grados, permite estudiar cómo se forman las estructuras de redes de la vida real y plantea como ejemplos prácticos de aplicación, la participación electoral, la propagación de epidemias, así como el comportamiento en redes sociales.

En la evaluación anterior se seleccionaron dentro del grupo de generadores de grafos aleatorios, tres modelos de generadores desarrollados por *Watts y Strogatz* [4]. Estos permiten de una forma relativamente sencilla y con menor número de parámetros generar los grafos. Una propiedad importante de estos tres generadores es que se desarrollan bajo la teoría de red de mundos pequeños. Esta propiedad revela según sus creadores que la sociedad humana es una red social con forma de mundo pequeño y que están interconectados entre sí, con estructura de red, cuyos nódulos son personas y los enlaces, las interrelaciones entre ellos [6].

Bajo esta teoría se crean nodos principales los cuales están alejados entre sí y generalmente se grafican más grandes que los demás, alrededor de estos se crean nodos que sí son vecinos entre sí y se grafican con tamaños más pequeños, de esta manera se garantiza la propiedad de crear grupos

dentro del mismo grafo permitiendo que sea relativamente fácil realizar la visita entre todos los nodos, reflejando mejor el comportamiento de fenómenos reales. Otra propiedad de estos generadores de grafos es que la distancia esperada entre dos nodos elegidos al azar crece de manera proporcional al logaritmo de la cantidad de nodos de la red mientras no se trate de los nodos que están más agrupados, propiedad que detectaron los creadores a partir del comportamiento real de diferentes fenómenos [8].

Para esta tarea de los generadores de grafos empleados en la anterior que fueron los siguientes:

- *grafo de Watts-Strogatz Newman*
- *grafo de Watts-Strogatz*
- *grafo de Watts-Strogatz Conectado*

Se seleccionó de los que menos influían en el tiempo de ejecución el generador de grafo de *Watts-Strogatz Conectado* en combinación con el algoritmo de flujo máximo *Edmons-Karp* de los siguientes algoritmos de flujo máximo empleados:

- *Algoritmo Boykov-Kolmogorov*
- *Algoritmo de Flujo Máximo*
- *Algoritmo Edmonds-Karp*

El algoritmo de *Edmonds-Karp* calcula el flujo máximo de un producto, debe emplearse con grafos dirigidos aunque también se emplea para no dirigidos, requiere asignársele una capacidad sino la toma como infinita [3] y, además, devuelve la red residual de flujo máximo (característica que el algoritmo *Flujo Máximo* no posee, pues este lo que devuelve es un arreglo del flujo máximo que pasa por las aristas seleccionadas) de ahí que se seleccionara el *Edmons-Karp* como algoritmo para calcular el flujo máximo en esta evaluación.

El ejemplo de aplicación que combina generador de grafo aleatorio con algoritmo de flujo máximo seleccionados puede ser la propagación de epidemias endémicas entre regiones determinadas, donde cada vértice serían las regiones objeto de estudios y las aristas el número de epidemias endémicas de cada región que se pueden propagar de un lugar a otro.

A continuación se muestra el fragmento de código desarrollado para generar los grafos con la capacidad asignada apoyándose en [1, 5, 2]:

```

1 def Generador_grafo(n,k,p):
2     #S=nx.watts_strogatz_graph(n, k, p)
3     S=nx.connected_watts_strogatz_graph(n,k,p, tries=100, seed=None)
4     scale = 2
5     rang = 11
6     size = S.number_of_edges()
7     e=S.edges(nbunch=None, data=True, default=None)
8     X = truncnorm(a=rang/scale , b=rang/scale , scale=scale).rvs(size=size)
9     X = X.round().astype(int)+rang
10    G=nx.Graph()
11    count=0;
12    for i in e:
13        G.add_edge(i[0],i[1],capacity=X[count])
14        count+=1
15    df = pd.DataFrame()
16    df = nx.to_pandas_adjacency(G, dtype=int , weight='capacity')
17    df.to_csv("grafo5.csv", index=None, header=None)
18    PrintGraph(S,X)
19
20 Generador_grafo(20,7,0.5)

```

Tarea5GeneradorGrafos.py

2.1. Generación de datos para el análisis estadístico

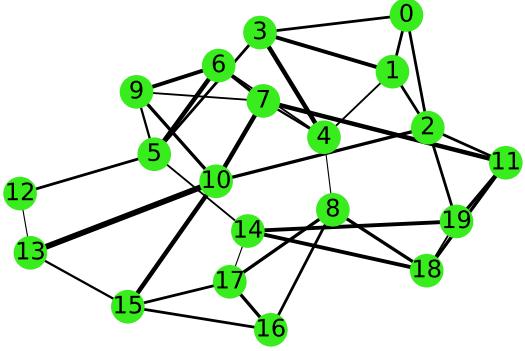
Los grafos se generaron con un código y luego otro los lee y dibuja con el algoritmo de acomodo usado en la tarea anterior y que mejor se ajustaba para esta tarea el *kamada_kawai*. El código usado para ello se muestra a continuación:

```
1 def lee_grafo (adress):
2     ds = pd.read_csv(adress , header=None)
3     G = nx.from_pandas_adjacency(ds)
4     return G
5
6 def calcula_tiempo(G,a,b):
7     start_time=time()
8     # fm=nx.maximum_flow(G, a, b,capacity="weight")
9     for i in range (100):
10         fm=edmonds_karp(G, a, b,capacity="weight")
11         print(i)
12         time_elapsed = time() - start_time
13
14     return time_elapsed
15
16 def DibujaGrafo(G):
17
18     pesos=[]
19     for edge in G.edges():
20         pesos.append(G.edges [edge] [ 'weight'])
21     pesos [:] = [x/7*x/10 for x in pesos]
22
23     # pos=nx.spring_layout(G)
24     pos=nx.kamada_kawai_layout(G,scale=10)
25
26     nx.draw_networkx_nodes(G, pos , node_size=400, node_color='#38ec1d' , node_shape='o')
27     nx.draw_networkx_edges(G, pos , width=pesos , edge_color='black')
28     labels = {}
29     for i in G.nodes:
30         labels [i]=str(i)
31     nx.draw_networkx_labels(G, pos , labels , font_size=15 )
32
33     plt.axis('off')
34     plt.savefig("fig5a.png" ,dpi=600)
35     plt.savefig("fig5a.eps" ,dpi=600)
36     df=pd.DataFrame(pos)
37     df.to_csv("pos_grafo5.csv" , index=None , header=None)
38     return pos
```

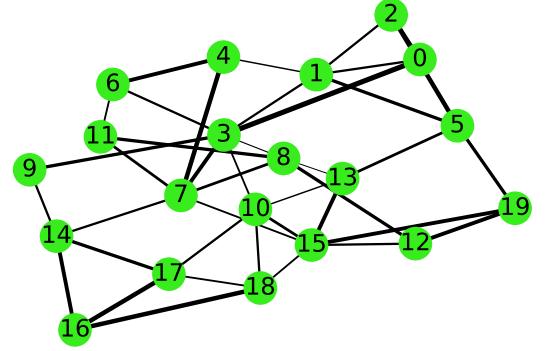
T5ProTimeFLujo.py

Los grafos generados se muestran en la figura 1 de la página 5

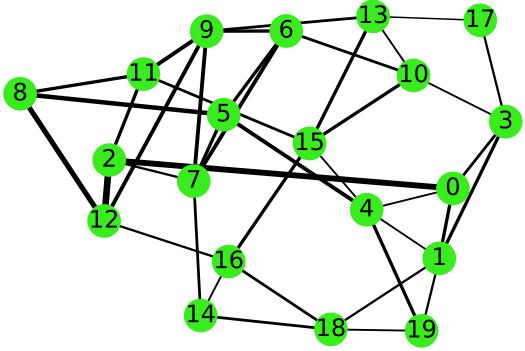
Luego se calcularon las propiedades para cada vértice de cada grafo usando el siguiente código:



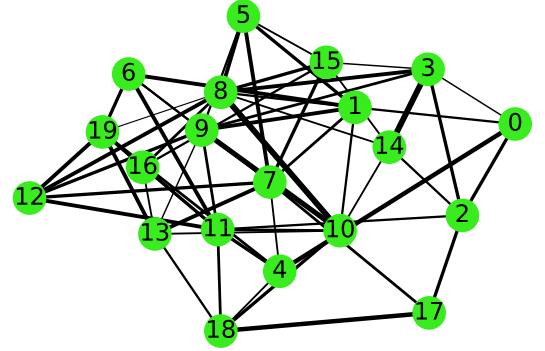
(a) *Grafo 1*



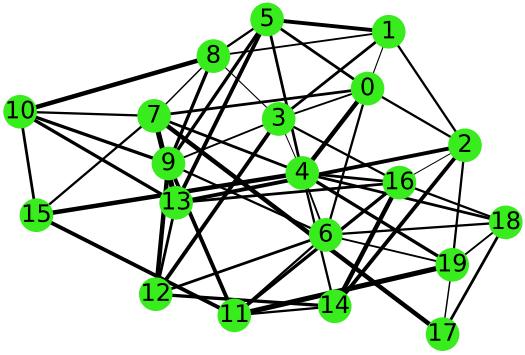
(b) *Grafo 2*



(c) *Grafo 3*



(d) *Grafo 4*



(e) *Grafo 5*

```

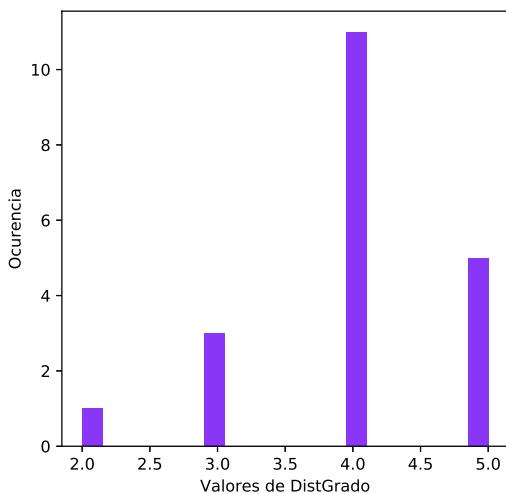
1 def lee_grafo(nombre):
2     ds = pd.read_csv(nombre, header=None)
3     G = nx.from_pandas_adjacency(ds)
4     return G
5
6 def lee_propiedades(nombre):
7     ds = pd.read_csv(nombre)
8     return ds
9
10 i=[ "grafo1.csv" , "grafo2.csv" , "grafo3.csv" , "grafo4.csv" , "grafo5.csv" ]
11 g=[]
12 for x in i:
13     G=lee_grafo(x)
14
15 dic={}
16 Nodes=G.nodes;
17 dic[ "Nodo" ]=Nodes
18 dic[ "DistGrado" ]=[G.degree(i) for i in Nodes]
19 dic[ "CoefAgrup" ]=[nx.clustering(G,i) for i in Nodes]
20 dic[ "CentCercania" ]=[nx.closeness_centrality(G,i) for i in Nodes]
21 dic[ "CentCarga" ]=[nx.load_centrality(G,i) for i in Nodes]
22 dic[ "ExcentCarga" ]=[round(nx.eccentricity(G,i), 2) for i in Nodes]
23 PageR=nx.pagerank(G, weight="capacity")
24 dic[ "PageR" ]=[PageR[i] for i in Nodes]
25 df=pd.DataFrame(dic)
26 df.to_csv("propiedades"+str(x)+".csv", index=None)
27 g.append("propiedades"+str(x)+".csv")
28
29 print(g)
30
31 j=[ "DistGrado" , "CoefAgrup" , "CentCercania" , "CentCarga" , "ExcentCarga" , "PageR" ]
32 for y in g:
33     H=lee_propiedades(y)
34     for i in j:
35         fig = plt.figure(figsize=(5, 5))
36         ax = fig.add_subplot(1, 1, 1)
37         his = ax.hist(H[i], bins=len(H[j]), facecolor="#8a36f8", alpha=0.75)
38         ax.set_xlabel("Valores de " + i)
39         ax.set_ylabel("Ocurrencia")
40     #     plt.savefig("histograma"+ y + i + ".png", bbox_inches="tight", dpi=100)
41     #     plt.savefig("histograma"+ y + i + ".eps", bbox_inches="tight", dpi=100)
42     #     plt.show()
43
44 print("fin")

```

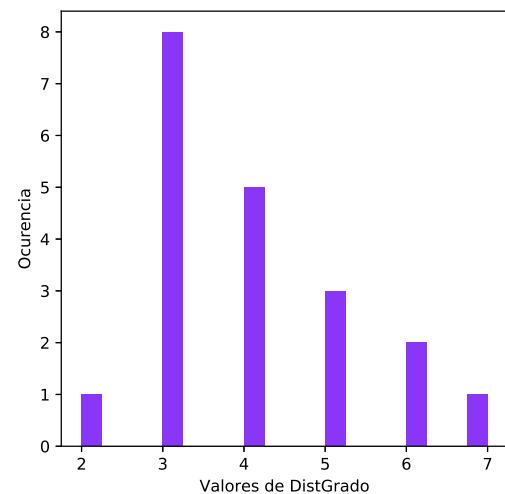
T5histoporopiedad.py

Para tener un avance del comportamiento de estas propiedades de los vértices las cuales se emplearan en el análisis de varianza más adelante, para cada grafo se realizaron los histogramas de cada una de las propiedades (distribución de grado, coeficiente de agrupamiento, centralidad de cercanía, centralidad de carga, excentricidad y *pagerank*) por cada grafo. Los mismos se muestran a continuación:

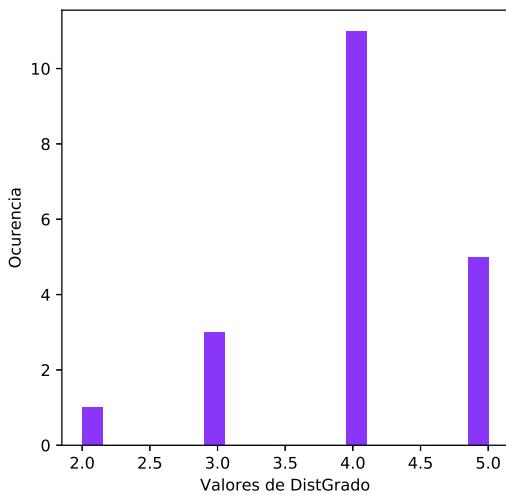
Análisis del comportamiento de la distribución de grado en cada grafo. Ver figura 2 de la página 7



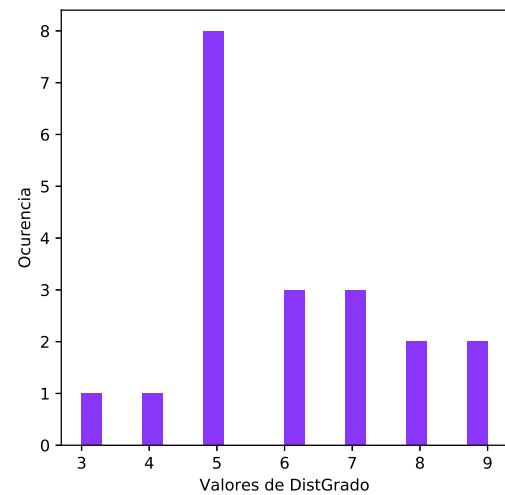
(a) *Grafo 1*



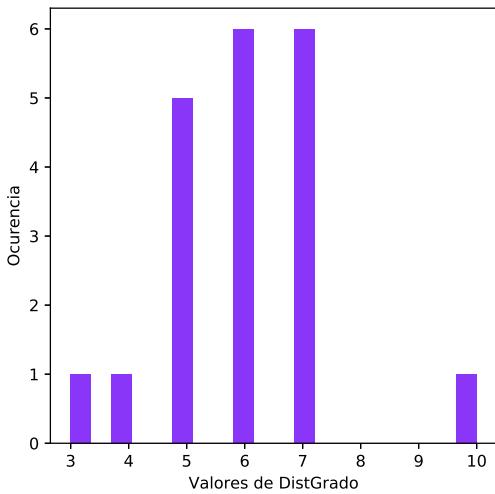
(b) *Grafo 2*



(c) *Grafo 3*



(d) *Grafo 4*



(e) *Grafo 5*

Figura 2: Histogramas de distribución de grado por grafo

Análisis del comportamiento del coeficiente de agrupamiento. Ver figura 3 de la página 9

Análisis del comportamiento de la centralidad de cercanía en cada grafo. Ver figura 4 de la página 10

Análisis del comportamiento de la centralidad de carga en cada grafo. Ver figura 5 de la página 11

Análisis del comportamiento de la excentricidad en cada grafo. Ver figura 6 de la página 12

Análisis del comportamiento del *pagerank* en cada grafo. Ver figura 7 de la página 13

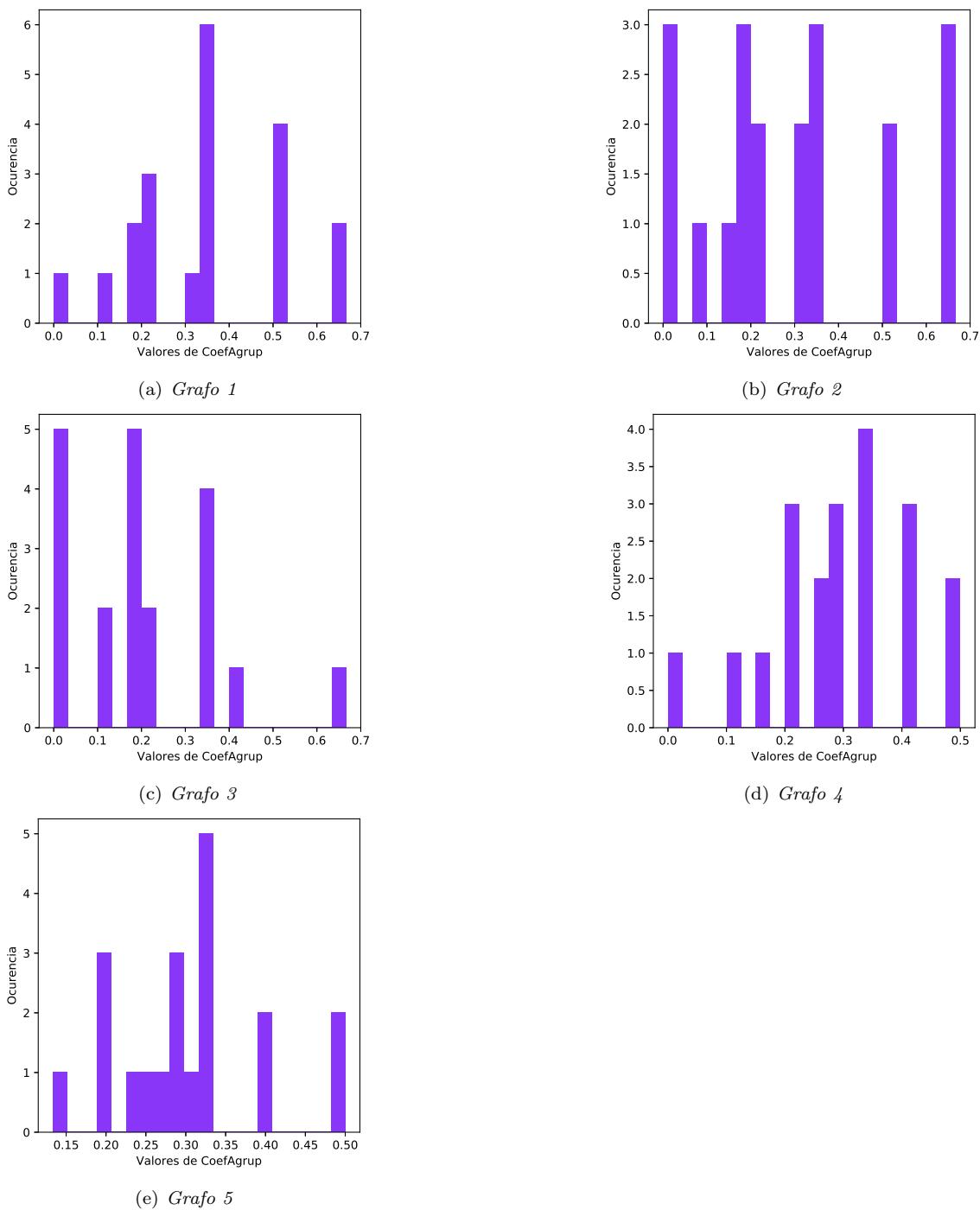


Figura 3: Histogramas del coeficiente de agrupamiento por grafo

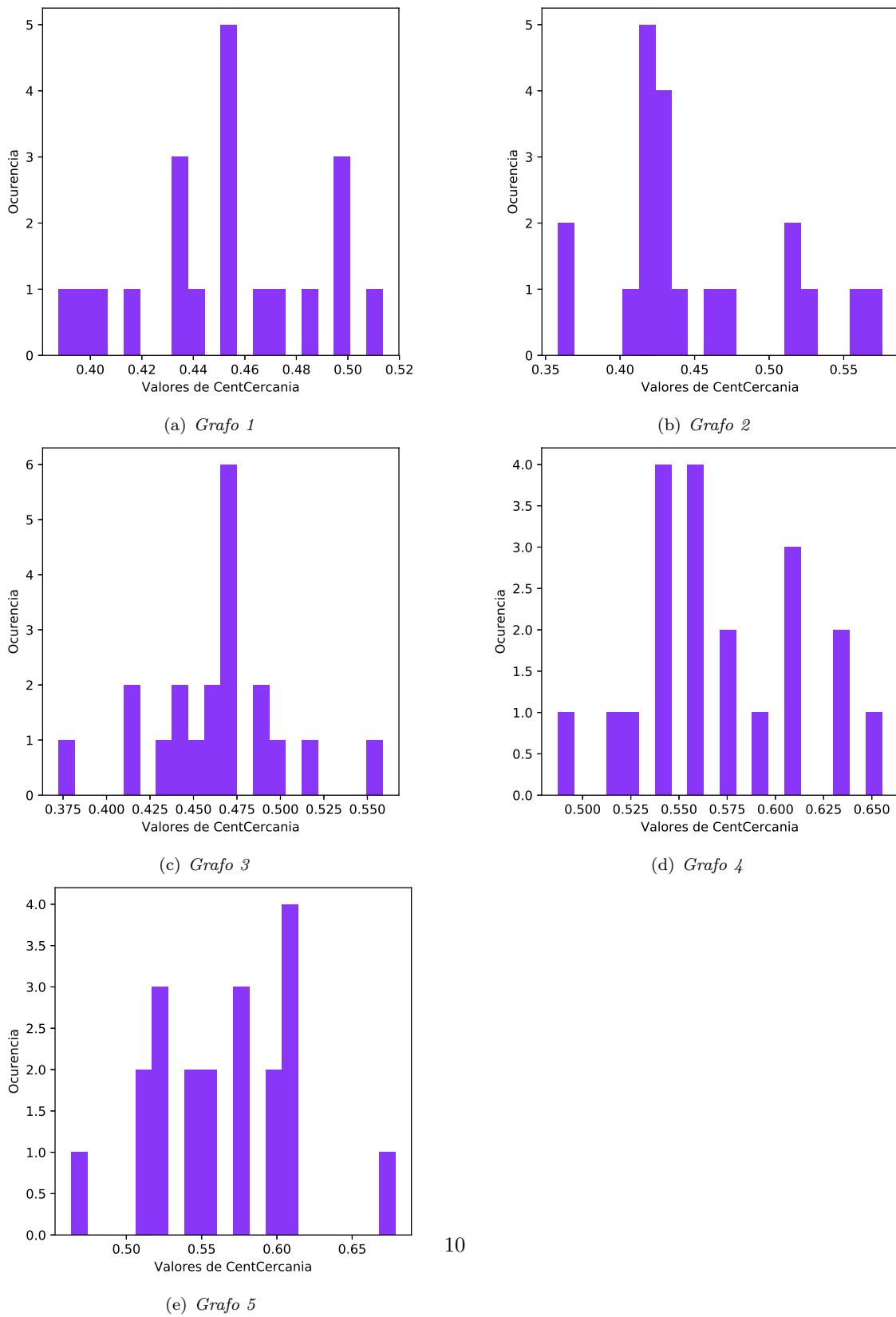


Figura 4: Histogramas de la centralidad de cercanía

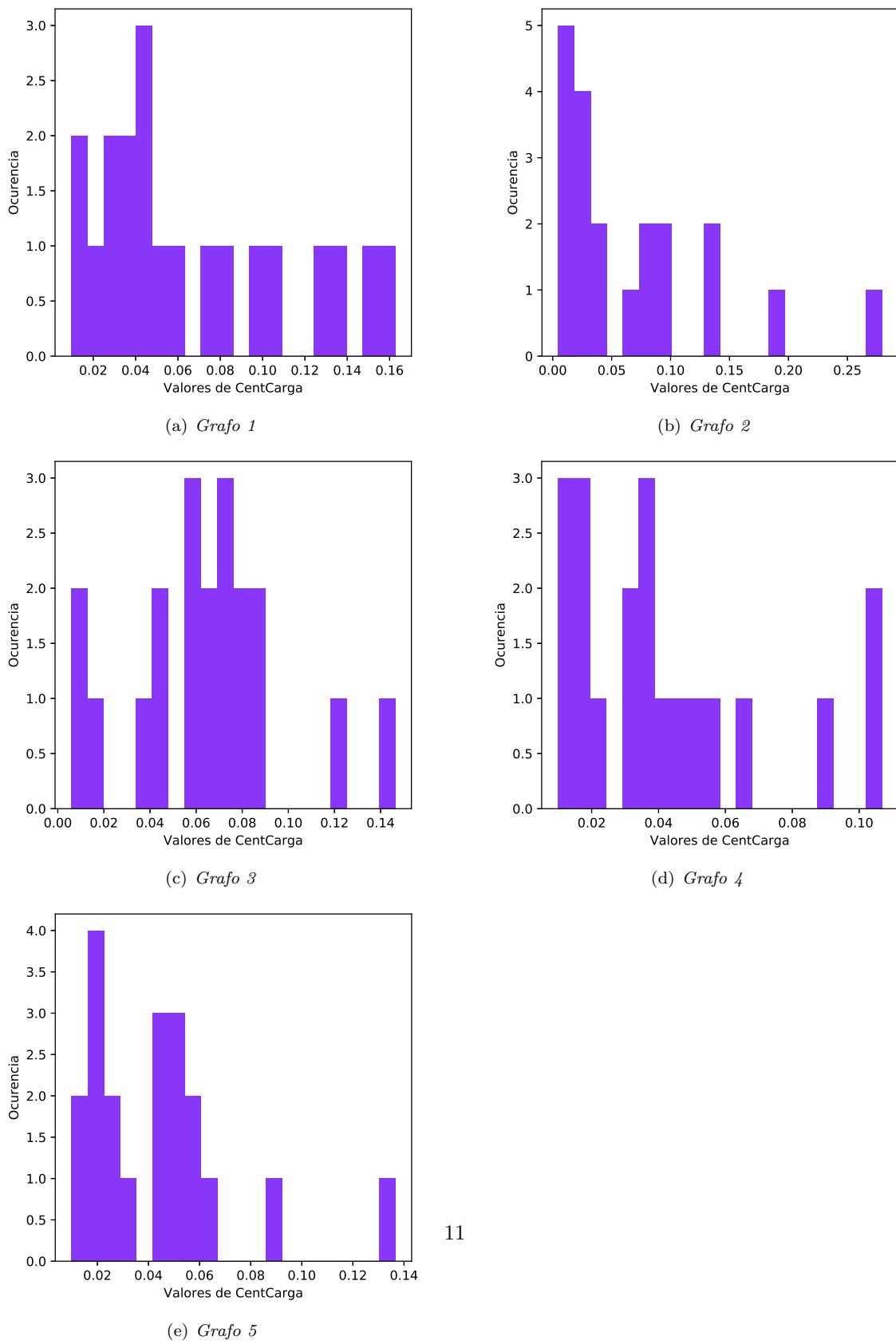
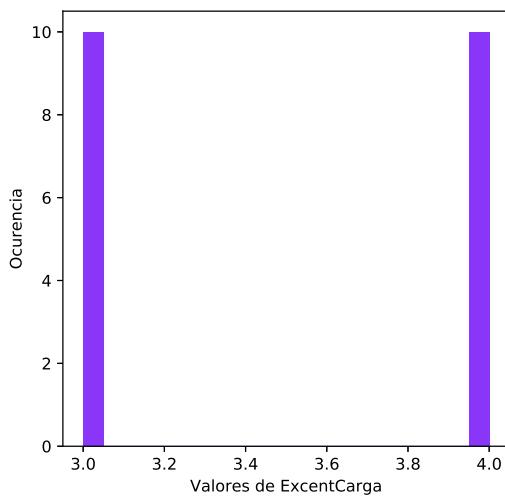
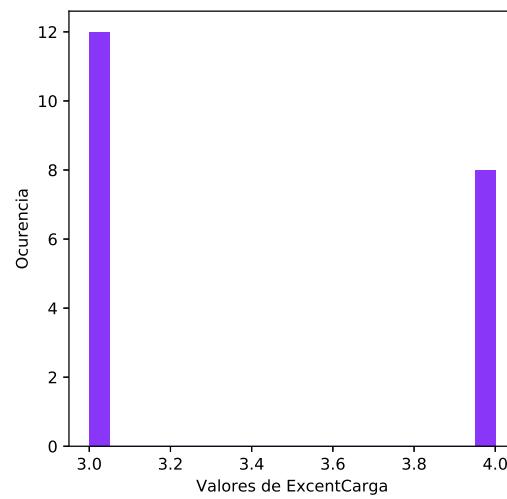


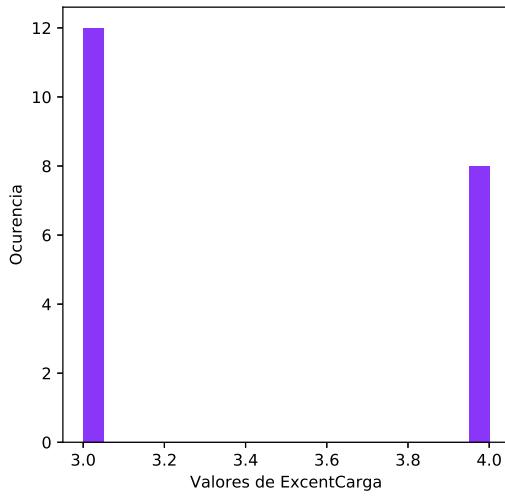
Figura 5: Histogramas de la centralidad de carga



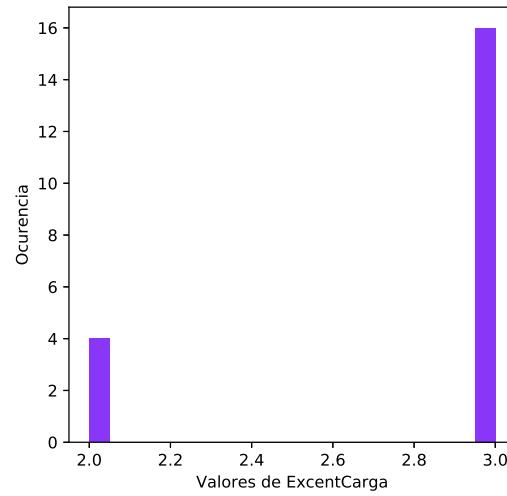
(a) *Grafo 1*



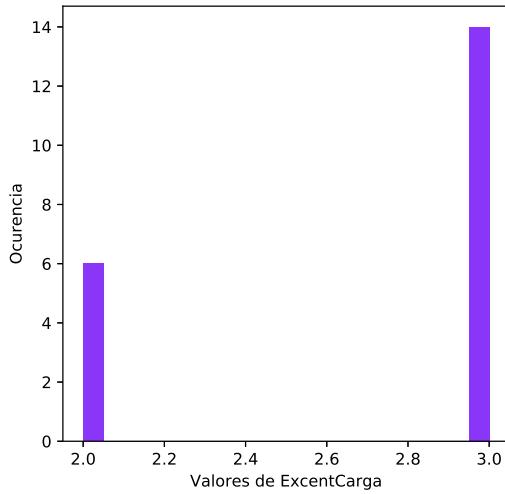
(b) *Grafo 2*



(c) *Grafo 3*

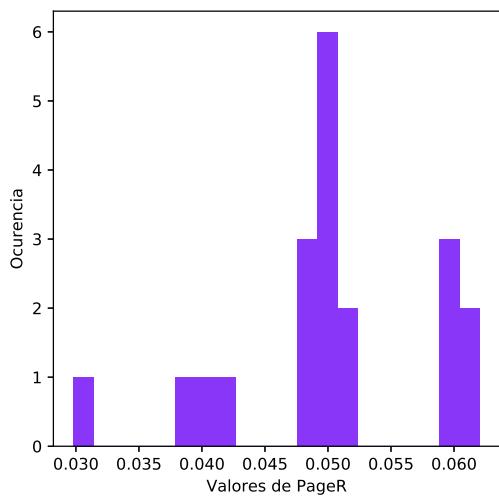


(d) *Grafo 4*

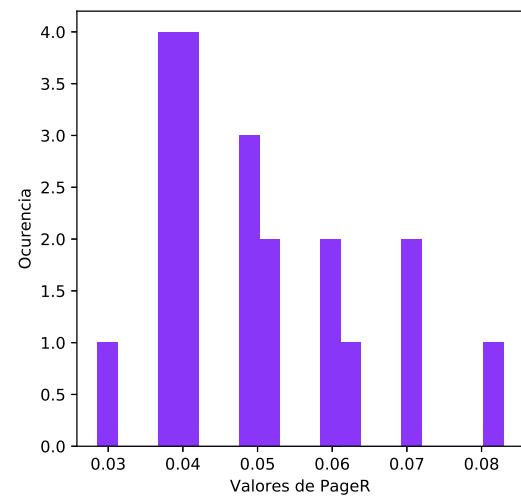


(e) *Grafo 5*

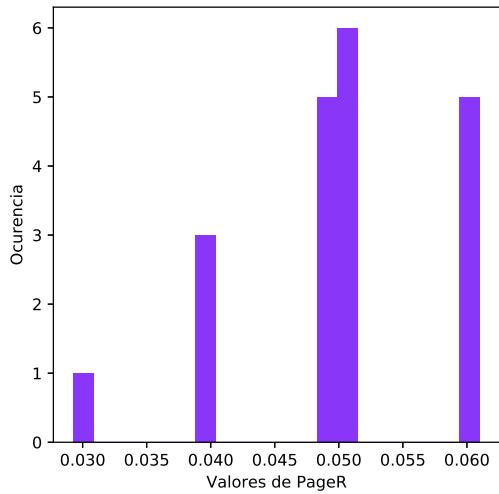
Figura 6: Histogramas de la excentricidad



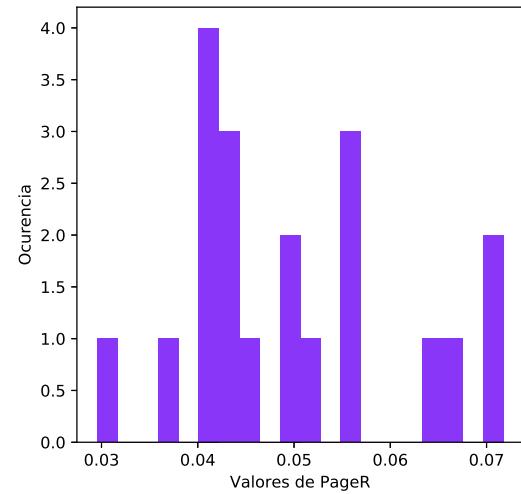
(a) *Grafo 1*



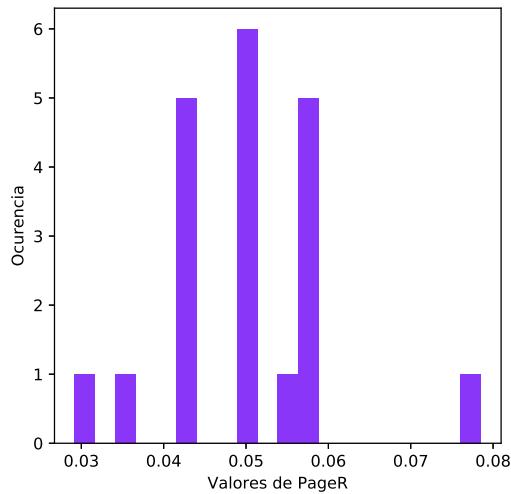
(b) *Grafo 2*



(c) *Grafo 3*



(d) *Grafo 4*



(e) *Grafo 5*

Al observar el comportamiento de estas variables se aprecia que no describen una distribución normal, por lo que no es recomendable hacer el análisis de varianza de influencia en el tiempo de ejecución del algoritmo y el flujo máximo usando el promedio, sino la mediana.

Para realizar el cálculo del tiempo de ejecución del algoritmo y el flujo máximo variando los nodos fuentes y sumideros de modo que toda la información quedase guardada en un .csv que pudiera emplearse en el análisis de varianza (ANOVA) se usó el siguiente código:

```

1 def calcula_tiempo(G,a,b):
2     start_time=time()
3     # fm=nx.maximum_flow(G, a, b,capacity="weight")
4     for i in range (100):
5         fm=edmonds_karp(G, a, b,capacity="weight")
6         print(i)
7         time_elapsed = time() - start_time
8
9     return time_elapsed
10
11
12 def Todo(G,nombre):
13     dic={"Grafo":[] ,
14         "Fuente":[] ,
15         "Sumidero":[] ,
16         "Mediatime":[] ,
17         "Medianatime":[] ,
18         "Stdtime":[] ,
19         "Flujomaximo":[] ,
20         "DistGrado":[] ,
21         "CoefAgrup":[] ,
22         "CentCercania":[] ,
23         "CentCarga":[] ,
24         "ExcentCarga":[] ,
25         "PageR":[] }
26
27 Nodes=G.nodes;
28 for i in Nodes:
29     for j in Nodes:
30         if i!=j:
31             t=[]
32             for k in range(10):
33                 t.append(calcula_tiempo(G,i,j))
34             dic["Grafo"].append(nombre)
35             dic["Fuente"].append(i)
36             dic["Sumidero"].append(j)
37             dic["Mediatime"].append(np.mean(t))
38             dic["Medianatime"].append(np.median(t))
39             dic["Stdtime"].append(np.std(t))
40             dic["Flujomaximo"].append(nx.maximum_flow_value(G,i,j,capacity="weight"))
41
42             dic["DistGrado"].append(G.degree(i))
43             dic["CoefAgrup"].append(nx.clustering(G,i))
44             dic["CentCercania"].append(nx.closeness_centrality(G,i))
45             dic["CentCarga"].append(nx.load_centrality(G,i))
46             dic["ExcentCarga"].append(nx.eccentricity(G,i))
47             PageR=nx.pagerank(G,weight="capacity")
48             dic["PageR"].append(PageR[i])

```

```

49
50
51     df=pd.DataFrame( dic )
52     df.to_csv("CSVunido.csv", index=None, mode="a")
53
54
55 i=[ "grafo1" , "grafo2" , "grafo3" , "grafo4" , "grafo5" ]
56 for x in i:
57     print (x)
58     G=lee_grafo(x)
59     Todo(G,x)

```

T5Paraunircsvfinal.py

3. Influencia en el óptimo al variar nodos fuentes y sumideros

Al variar los nodos fuentes y sumideros fue calculado el valor de flujo máximo que se obtenía en cada caso, evidenciándose que existe variación en el óptimo para cada uno de los cinco grafos. A continuación se muestra para cada uno de los cinco grafos generados inicialmente el grafo inicial con la propuesta de mejor y peor fuente y sumidero. La fuente aparece con color rojo y el sumidero con color azul. De igual modo se emplea el degradado de color rojo para representar con rojo más intenso la mayor cantidad de flujo y más claro la menor cantidad de flujo que pasa por las aristas de la red residual que devuelve el algoritmo de flujo máximo seleccionado.

En la figura 8 de la página 16 se observa la variación realizada para el grafo uno, en la figura 9 de la página 17 la variación del grafo dos, en la figura 10 de la página 18 la variación del grafo tres, en la figura 11 de la página 19 la variación del grafo cuatro y figura 12 de la página 20 la variación del grafo cinco.

Seguidamente se muestra el código empleado en esta sección

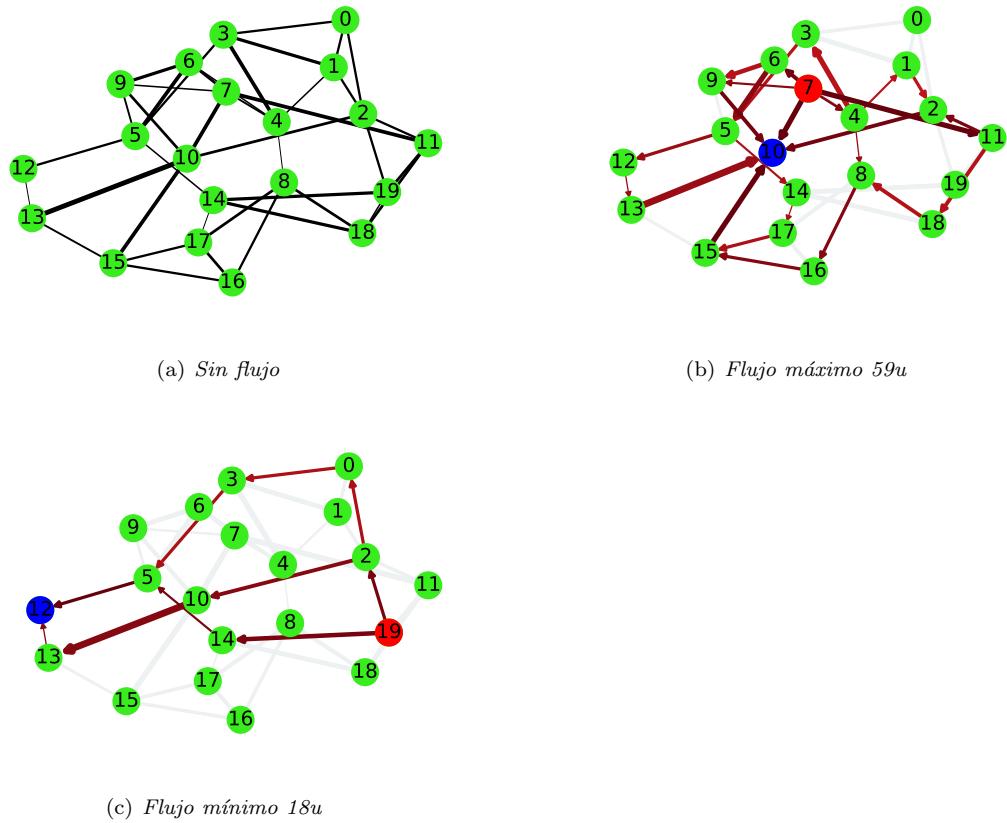


Figura 8: Variación de fuente y sumidero en grafo 1

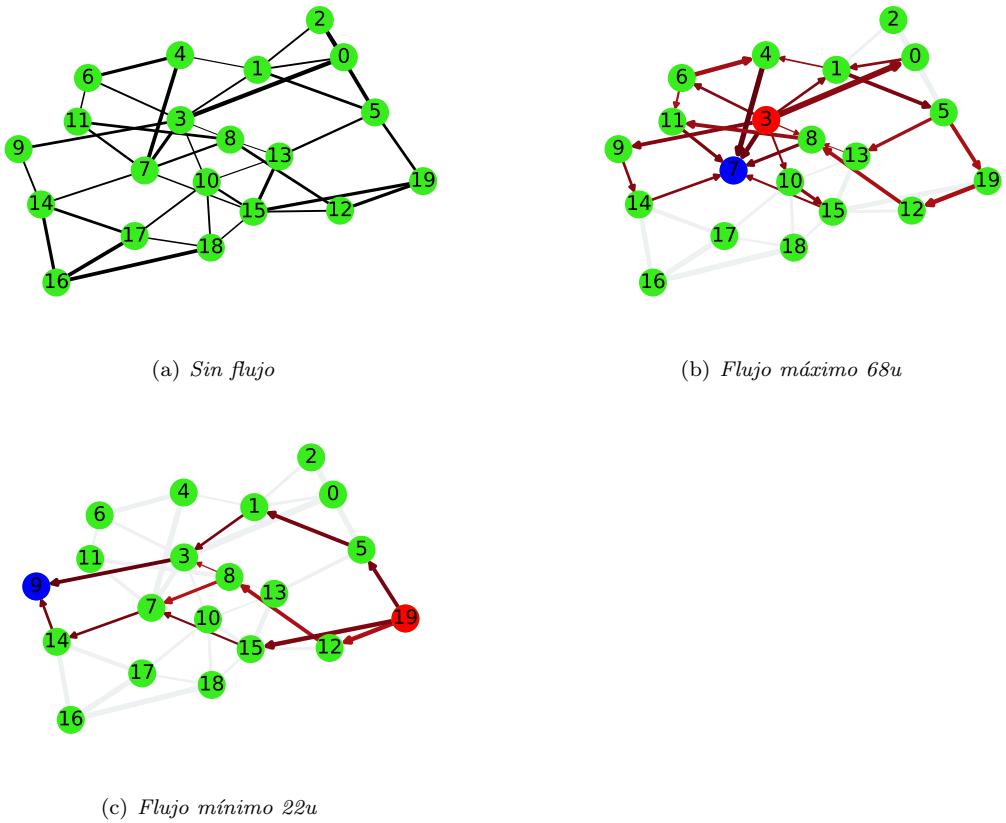


Figura 9: Variación de fuente y sumidero en grafo 2

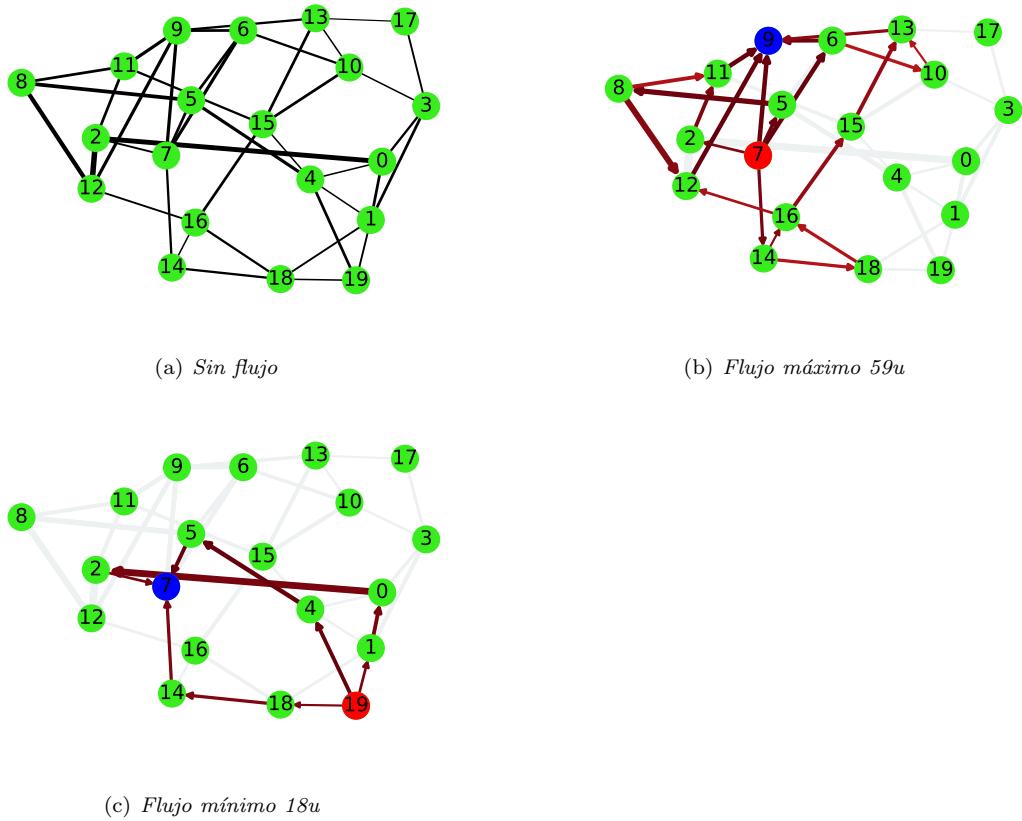


Figura 10: Variación de fuente y sumidero en grafo 3

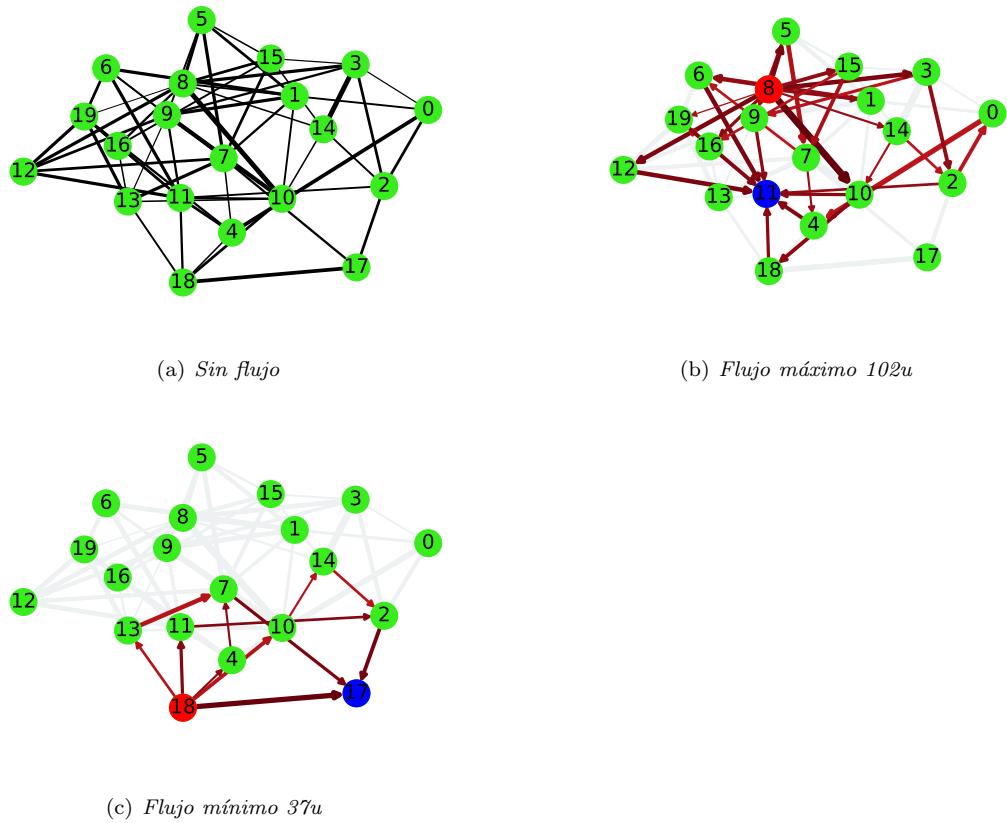


Figura 11: Variación de fuente y sumidero en grafo 4

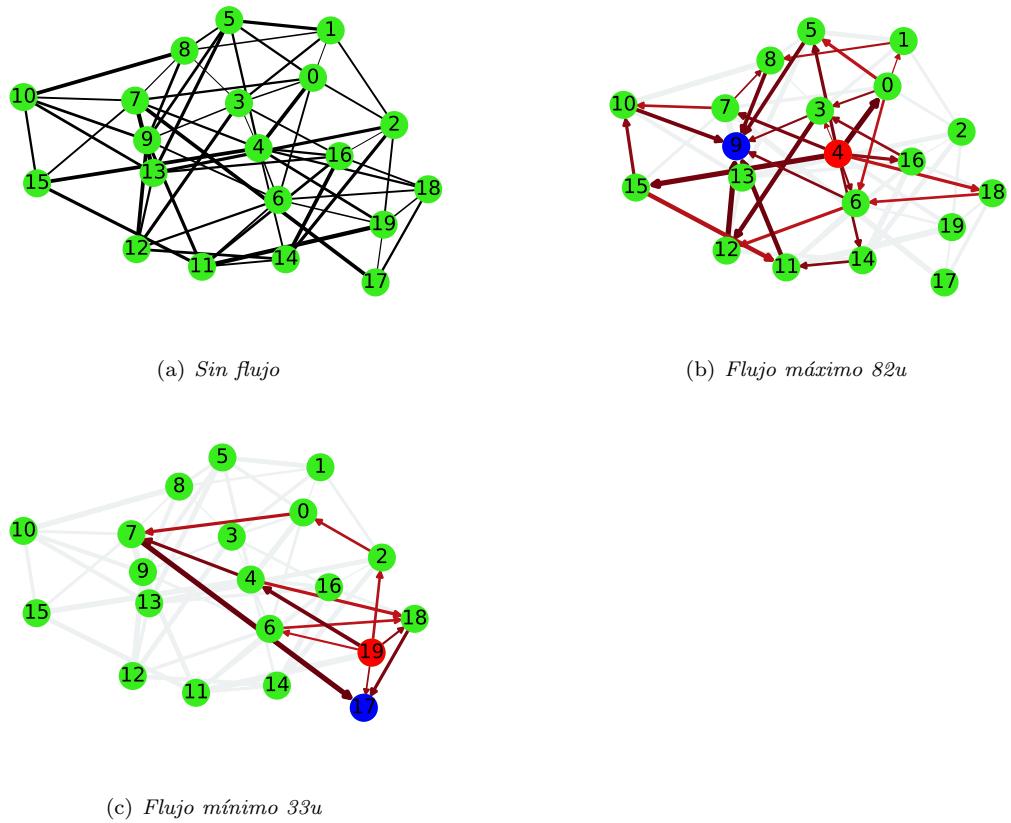


Figura 12: Variación de fuente y sumidero en grafo 5

```

1 def Todo(G):
2     dic={"Fuente":[] ,
3         "Sumidero":[] ,
4         "Mediatime":[] ,
5         "Medianatime":[] ,
6         "Stdtime":[] ,
7         "Flujomaximo":[] ,
8         "DistGrado":[] ,
9         "CoefAgrup":[] ,
10        "CentCercania":[] ,
11        "CentCarga":[] ,
12        "ExcentCarga":[] ,
13        "PageR":[] }
14
15 Nodes=G.nodes;
16 for i in Nodes:
17     for j in Nodes:
18         if i!=j:
19             t=[]
20             for k in range(10):
21                 t.append(calcula_tiempo(G,i,j))
22             dic["Fuente"].append(i)
23             dic["Sumidero"].append(j)
24             dic["Mediatime"].append(np.mean(t))
25             dic["Medianatime"].append(np.median(t))
26             dic["Stdtime"].append(np.std(t))
27             dic["Flujomaximo"].append(nx.maximum_flow_value(G,i,j,capacity="weight"))
28
29             dic["DistGrado"].append(G.degree(i))
30             dic["CoefAgrup"].append(nx.clustering(G,i))
31             dic["CentCercania"].append(nx.closeness_centrality(G,i))
32             dic["CentCarga"].append(nx.load_centrality(G,i))
33             dic["ExcentCarga"].append(nx.eccentricity(G,i))
34             PageR=nx.pagerank(G,weight="capacity")
35             dic["PageR"].append(PageR[i])
36
37
38 df=pd.DataFrame(dic)
39 df.to_csv("Todosvaloresg5.csv", index=None)
40
41 def DibujarRes(G, pos , fuentes , sumideros):
42 #    pos=nx.spring_layout(R)
43 sinflujo=[]
44 flowconflujo=[]
45 conflujo=[]
46 sinflujopesos=[]
47 conflujopesos=[]
48 maxi=0
49 for edge in G.edges():
50     if G.edges[edge]['flow']==0:
51         sinflujo.append(edge)
52         sinflujopesos.append( G.edges[edge]['capacity'] )
53     elif G.edges[edge]['flow']>0:
54         conflujo.append(edge)
55         conflujopesos.append( G.edges[edge]['capacity'] )
56         flowconflujo.append(G.edges[edge]['flow'])

```

```

57     flowconfluo [:] = [x+50 for x in flowconfluo]
58     for i in flowconfluo:
59         if i>maxi:
60             maxi=i
61     sinfluojopesos [:] = [x/10*x/5 for x in sinfluojopesos]
62     confluojopesos [:] = [x/10*x/5 for x in confluojopesos]
63     nx.draw_networkx_nodes(G, pos, node_size=400, node_color='#38ec1d', node_shape='o')
64     nx.draw_networkx_nodes(G, pos, nodelist=fuentes, node_size=400, node_color='r',
65                           node_shape='o')
66     nx.draw_networkx_nodes(G, pos, nodelist=sumideros, node_size=400, node_color='b',
67                           node_shape='o')
68     nx.draw_networkx_edges(G, pos, edgelist=sinflujo, edge_color='#eef1f2', width=
69                           sinfluojopesos, arrows=False)
70     nx.draw_networkx_edges(G, pos, edgelist=conflujo, edge_cmap=plt.cm.Reds, width=
71                           confluojopesos, edge_color=flowconfluo, edge_vmin=0,edge_vmax=maxi)
72     labels = {}
73     for i in G.nodes:
74         labels[i]=str(i)
75     nx.draw_networkx_labels(G, pos, labels, font_size=15)
76     plt.axis('off')
77     plt.savefig("fig5b1.png",dpi=600)
78     plt.savefig("fig5b1.eps",dpi=600)

79
80
81
82
83
84

```

T5ProTimeFLujo.py

4. Análisis de varianza (ANOVA) y prueba de *Tukey* para determinar la relación entre las propiedades de los nodos y las variables dependientes estudiadas

Para realizar el análisis del comportamiento de las variables dependientes *tiempo de ejecución* y *flujo máximo* con respecto a cada factor a analizar se realizó un análisis de varianza (ANOVA) para cada factor.

En el caso de las propiedades que se usan como factores en el ANOVA se realizó un histograma para cada una de ellas de modo que se pudiera llevar a escalas los valores de las mismas.

Los histogramas se emplean para establecer los rangos de valores asignándoles una etiqueta de bajo, medio y alto para cada propiedad.

El resultado del análisis ANOVA con respecto al tiempo de ejecución para cada propiedad se muestra en los cuadros del uno al seis. Según los resultados en todas las propiedades se rechaza la hipótesis nula por lo que todas influyen en el tiempo excepto la centralidad de carga de rango alto, en la que según los resultados obtenidos se acepta la hipótesis nula. Esto se puede corroborar al analizar los diagramas de cajas de cada una de las propiedades. Ver figura 13 de la página 25. Del análisis de los resultados se aprecia que el que menos influye es la centralidad de carga, como se observa también en los diagramas de caja y prueba de *Tukey* (ver figura 14 de la página 26) el resto de las propiedades sí influyen en el tiempo de ejecución del algoritmo al variar los nodos fuentes y sumideros.

Cuadro 1: ANOVA Centralidad de Carga

Source	SS	DF	MS	F	p-unc	np2
CentCarga	0	2	0	0.058	0.94360248	0
Within	0.126	1897	0	-	-	-

Cuadro 2: ANOVA Centralidad de cercanía

Source	SS	DF	MS	F	p-unc	np2
CentCercania	0.024	2	0.012	221.416	3.76E-87	0.189
Within	0.102	1897	0	-	-	-

Cuadro 3: ANOVA Coeficiente de agrupamiento

Source	SS	DF	MS	F	p-unc	np2
CoefAgrup	0.007	2	0.004	56.285	1.79E-24	0.056
Within	0.119	1897	0	-	-	-

Source	Cuadro 4: ANOVA Distribución de Grado					
	SS	DF	MS	F	p-unc	np2
DistGrado	0.023	2	0.011	209.861	4.61E-83	0.181
Within	0.103	1897	0	-	-	-

Source	Cuadro 5: ANOVA Excentricidad					
	SS	DF	MS	F	p-unc	np2
ExcentCarga	0.013	2	0.006	104.776	6.90E-44	0.099
Within	0.113	1897	0	-	-	-

Los resultados *Tukey* de este análisis se muestran en la figura 14 de la página 26

Source	Cuadro 6: ANOVA <i>PageRange</i>					
	SS	DF	MS	F	p-unc	np2
PageR	0.01	2	0.005	82.216	5.72E-35	0.08
Within	0.116	1897	0	-	-	-

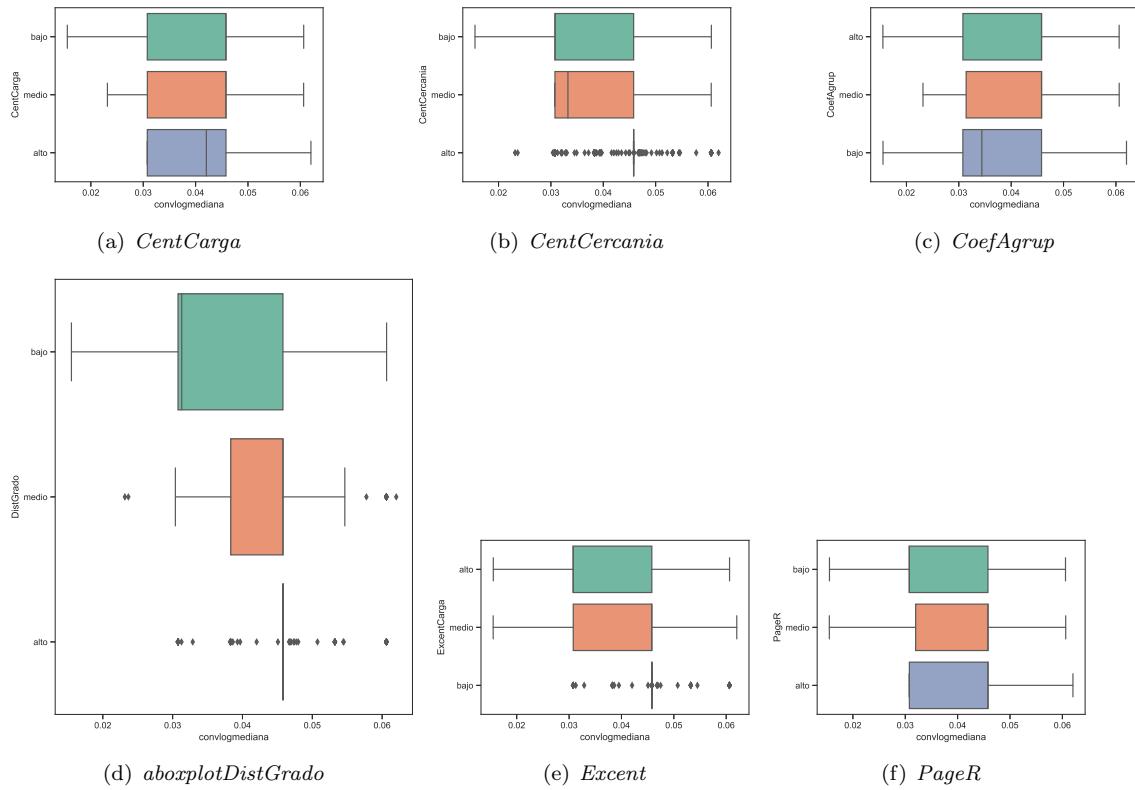


Figura 13: Diagrama de caja del ANOVA con Tiempo

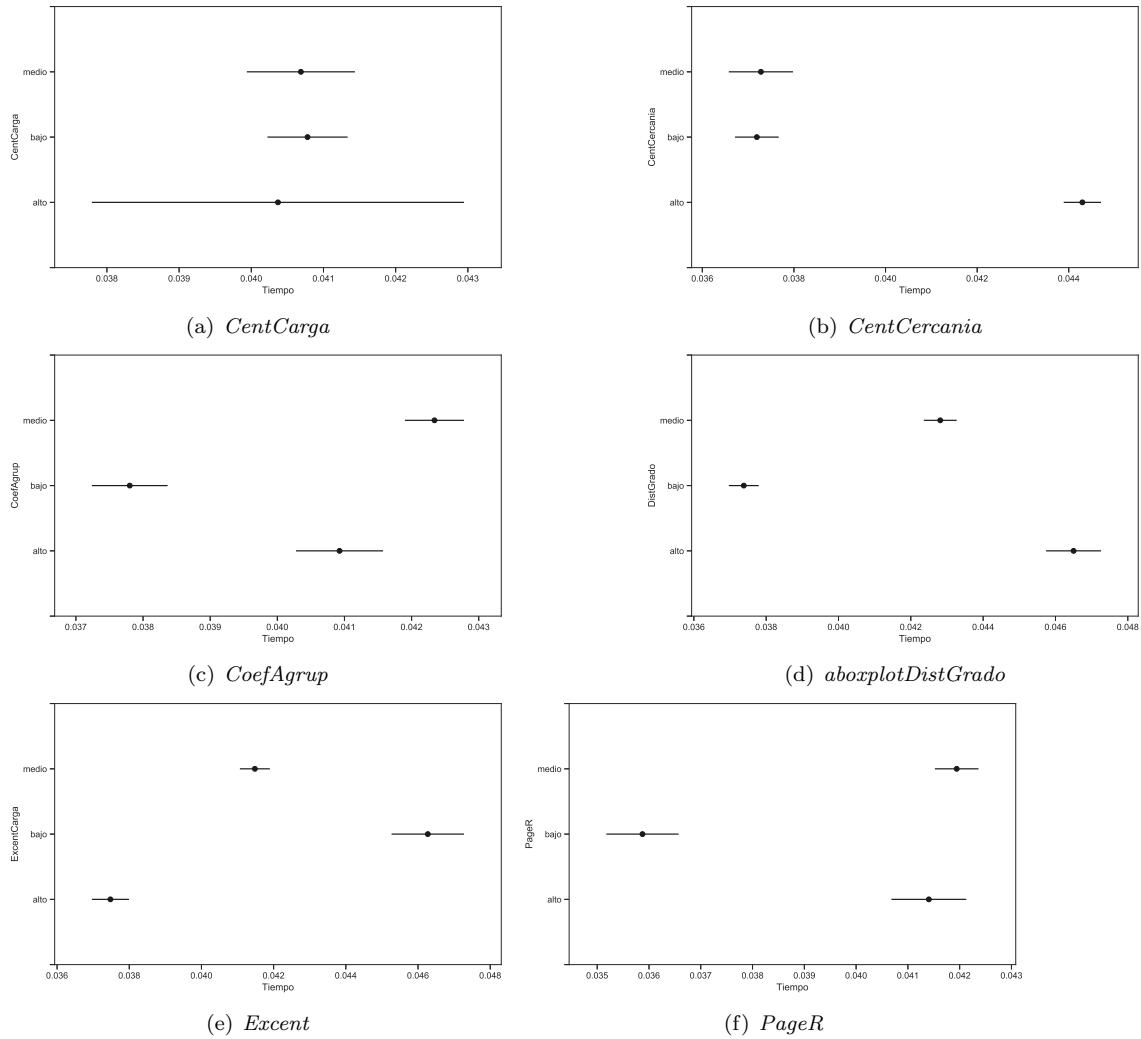


Figura 14: Diagrama de TUKEY para ANOVA con Tiempo

Al realizar el análisis ANOVA para verificar la influencia de las propiedades en el flujo máximo se obtuvieron los resultados que se muestran en los cuadros del 7 al 12. En los mismos se aprecia que se rechaza la hipótesis nula en todos los casos excepto en la propiedad de centralidad de carga en la que se acepta la hipótesis nula. Se incluyen los diagramas de caja (ver figura 15 de la página 28) y las pruebas de *Tukey* para profundizar en los resultados (ver figura 16 de la página 29). Se observa que la propiedad que menos influye es la centralidad de carga para valores altos, el resto de las propiedades sí influye al variar los nodos fuentes y sumideros en el valor de flujo máximo.

Source	Cuadro 7: ANOVA flujo max CentCarga					
	SS	DF	MS	F	p-unc	np2
CentCarga	0.471	2	0.236	2.473	0.08461665	0.003
Within	180.834	1897	0.095	-	-	-

Source	Cuadro 8: ANOVA flujo max CentCercanía					
	SS	DF	MS	F	p-unc	np2
CentCercanía	68.685	2	34.343	578.476	7.16E-197	0.379
Within	112.62	1897	0.059	-	-	-

Source	Cuadro 9: ANOVA flujo max CoefAgrup					
	SS	DF	MS	F	p-unc	np2
CoefAgrup	19.784	2	9.892	116.181	2.53E-48	0.109
Within	161.521	1897	0.085	-	-	-

Source	Cuadro 10: ANOVA flujo max DistGrado					
	SS	DF	MS	F	p-unc	np2
DistGrado	68.851	2	34.426	580.732	1.77E-197	0.38
Within	112.454	1897	0.059	-	-	-

Source	Cuadro 11: ANOVA flujo max Excent					
	SS	DF	MS	F	p-unc	np2
ExcentCarga	54.856	2	27.428	411.472	3.69E-149	0.303
Within	126.45	1897	0.067	-	-	-

Source	Cuadro 12: ANOVA flujo max PageR					
	SS	DF	MS	F	p-unc	np2
PageR	40.988	2	20.494	277.069	2.70E-106	0.226
Within	140.317	1897	0.074	-	-	-

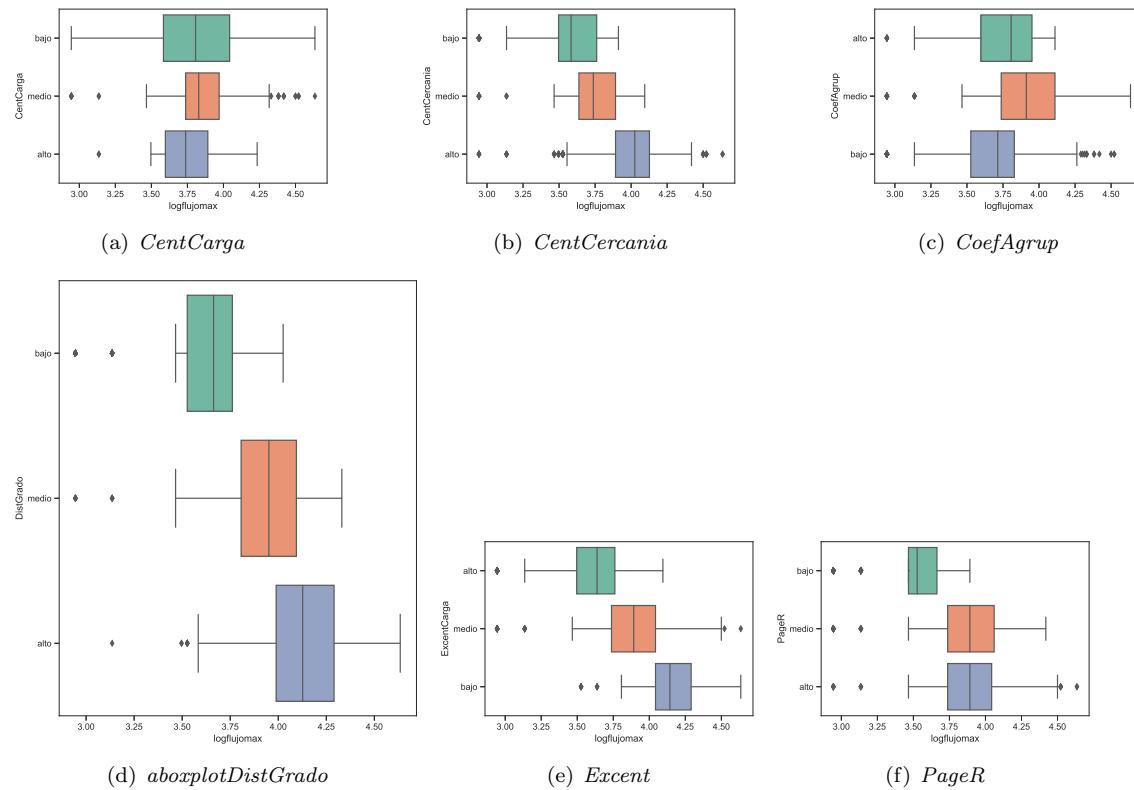


Figura 15: Diagrama de caja del ANOVA con flujo máximo

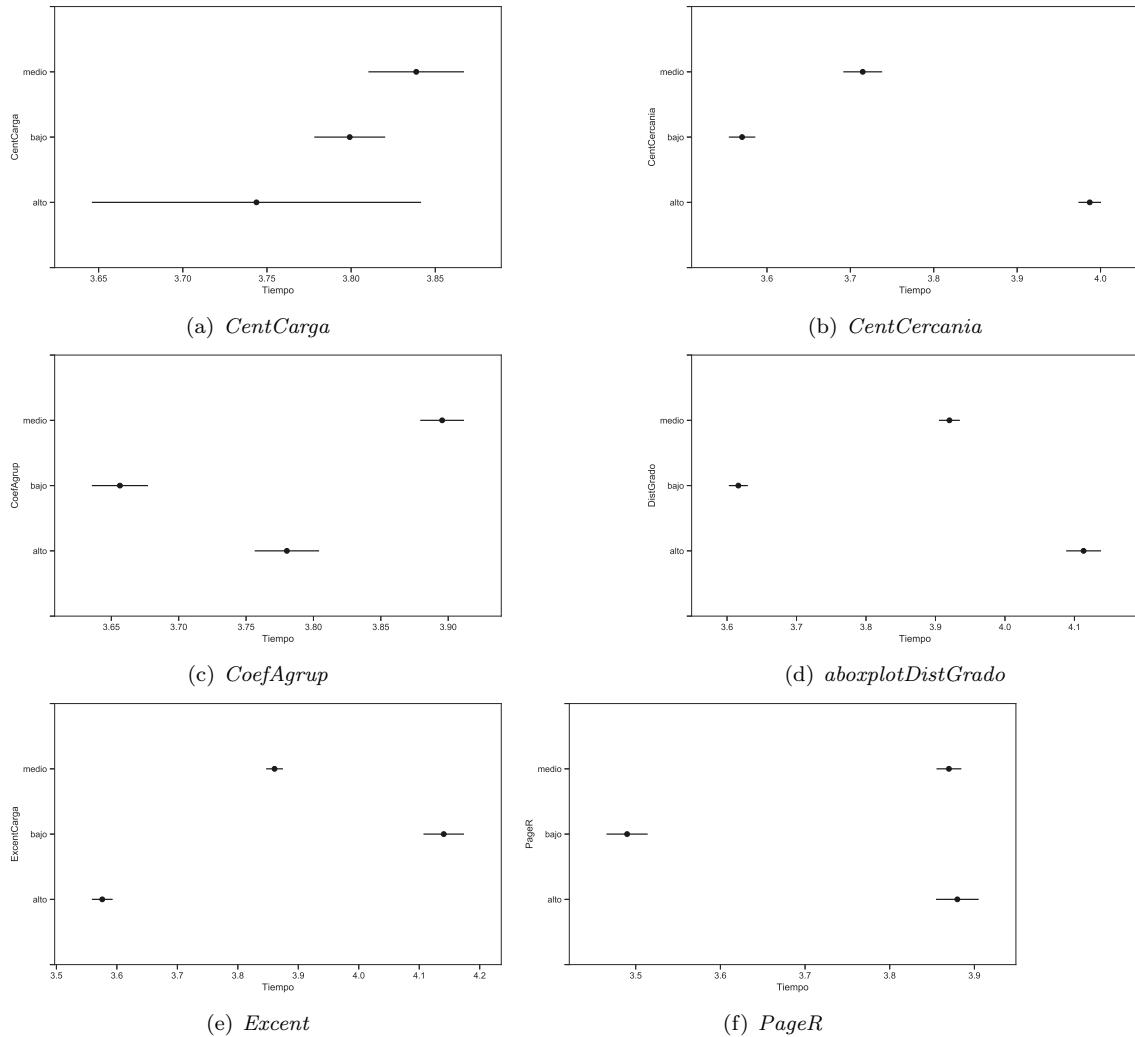


Figura 16: Diagrama de TUKEY para ANOVA con flujo máximo

Referencias

- [1] Leonardo J. Caballero. *Materiales del entrenamiento de programación en Python*. 2019.
- [2] Desarrolladores de Networkx. Add-edge. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.MultiDiGraph.add_edge.html, . Accessed:2019-04-19.
- [3] Desarrolladores de Networkx, . URL https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.flow.edmonds_karp.html.
- [4] Desarrolladores de Networkx. <https://https://networkx.github.io/documentation/stable/reference/generators.html>, . Accessed: 2019-03-29.
- [5] Desarrolladores de Scipy.Org. Función estadística *scipy.stats.truncnorm*. <https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.truncnorm.html>. Accessed:2019-04-19.
- [6] Pedro A. Solares Hernández. *Redes aleatorias de pequeño mundo y libres de escalas*. Universidad Politécnica de Valencia, 2017. Accessed:2019-04-26.
- [7] Valerie De la Cruz. What are the applications of random graphs? develop python with pycharm. URL <https://www.quora.com/What-are-the-applications-of-random-graphs>.
- [8] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, June 1998. doi: 10.1038/30918. URL <https://worrydream.com/refs/Watts-CollectiveDynamicsOfSmallWorldNetworks.pdf>.