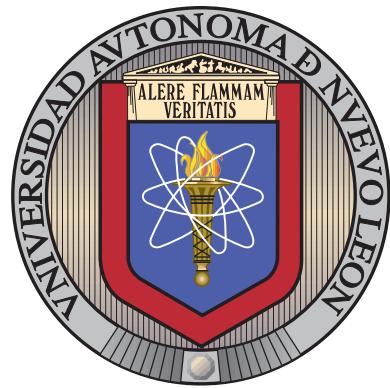


UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN

FACULTAD DE INGENIERÍA MECÁNICA Y ELÉCTRICA



RETROALIMENTACIÓN DE LAS TAREAS
REALIZADAS DURANTE EL PERÍODO
FEBRERO-MAYO DE 2019

POR

ALBERTO MARTÍNEZ NOA

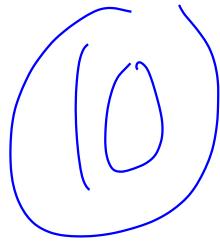
1985271

SAN NICOLÁS DE LOS GARZA, NUEVO LEÓN

MAYO 2019

1. Introducción

En este documento se encuentran las tareas realizadas en el periodo comprendido entre los meses de febrero y mayo, cada una de ellas recibió una retroalimentación en clases. Dicha retroalimentación fue considerada a la hora de rectificar los errores cometidos y aprender de ellos, por lo que se muestran a continuación las tareas revisadas y las corregidas.



Optimización de flujo en redes

Tarea 1: Representación de redes a través de la teoría de grafos

Alberto Martínez Noa Matrícula-1985271

11 de febrero de 2019

1. Grafo simple no dirigido acíclico

Se pueden encontrar varias aplicaciones a la modelación de un grafo simple no dirigido acíclico, una de las más claras son los árboles, el árbol (árbol libre) que es un grafo no dirigido, conexo y acíclico. Un árbol también puede definirse como un grafo no dirigido en el que hay exactamente un camino entre todo par de vértices” [1].

Un ejemplo de usos de árboles es en topología de red la de árbol, en esta topología los nodos de la red están ubicados en forma de árbol. Esta conexión es similar a muchas redes en estrella interconectadas con la diferencia de no poseer un nodo central, en cambio posee un nodo troncal desde el cual se ramifican el resto de los nodos como se muestra en la figura 1 que es una pequeña representación con solo 8 nodos de la red que posee la UEB Rolando Pérez Gollanes entidad dedicada al sacrificio y procesamiento de aves.

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

A=nx.Graph()
A.add_edges_from([('R1','Sw1'),('Sw1','Sw2'),
                  ('Sw1','Sw3'),('Sw2','Pc2'),('Sw2','Pc3'),
                  ('Sw3','Pc4'),('Sw3','Pc5')])

posicion=nx.spring_layout(A)
nx.draw_networkx_nodes(A,posicion, node_size=800)
nx.draw_networkx_edges(A,posicion, width=2)
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial')

plot.axis('off')
plot.savefig("Graf1.eps")
plot.show(A)
```

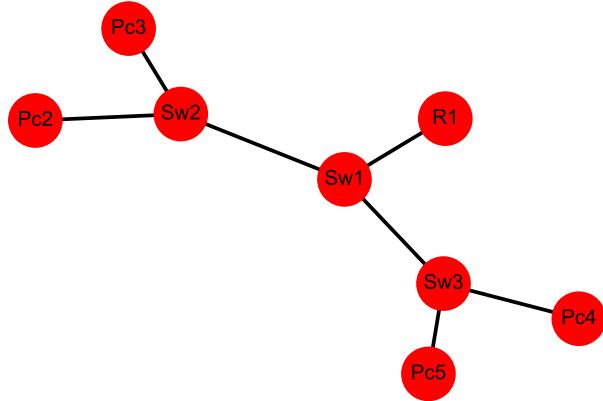


Figura 1: Grafo simple no dirigido acíclico

2. Grafo simple no dirigido cíclico

Un grafo simple no dirigido cíclico puede usarse áreas como geografía. Si se considera un mapa, digamos de Europa, que cada país sea un vértice y conecte dos vértices con una arista si esos países comparten una frontera. Un problema famoso que quedó sin resolver durante más de cien años fue el problema de los cuatro colores. Aproximadamente esto indica que cualquier mapa puede ser coloreado con a lo sumo 4 colores de tal manera que los países adyacentes no tengan el mismo color. Este problema motivó muchos desarrollos en la teoría de grafos y finalmente se demostró con la ayuda de una computadora en 1976[4].

Otra aplicación es en la representación de redes sociales, una red social se conceptualiza como un grafo, es decir, un conjunto de vértices (o nodos, unidades, puntos) que representan entidades u objetos sociales y un conjunto de líneas que representan una o más relaciones sociales entre ellos[2].

Por ejemplo, si consideramos un grupo de 9 doctores del núcleo académico de PISIS (Posgrado en Ingeniería de Sistemas) y construimos una red, tomando como vértices a los doctores y la colaboración de ellos en artículos publicados como aristas dará lugar a un grafo simple no dirigido cíclico como se muestra en la figura 2.

```
import networkx as nx
```

```

import matplotlib.pyplot as plot

A=nx.Graph()
A.add_edges_from([('Dr.FL','Dra.AA'),('Dr.FL','Dra.YR'),
                 ('Dr.FL','Dr.RS'), ('Dr.FL','Dra.ES'),
                 ('Dra.YR','Dr.VB'), ('Dra.YR','Dr.RR'),
                 ('Dra.YR','Dr.RS'), ('Dra.AA','Dra.IM'),
                 ('Dra.AA','Dr.RR'), ('Dra.IM','Dr.VB'),
                 ('Dra.IM','Dra.AS'), ('Dra.IM','Dr.RS'),
                 ('Dra.IM','Dr.VB'), ('Dra.AS','Dr.RR'),
                 ('Dra.AS','Dr.RS'), ('Dra.ES','Dr.RR')])

posicion=nx.spring_layout(A)
nx.draw_networkx_nodes(A,posicion, node_size=1700, node_color= 'grey')
nx.draw_networkx_edges(A,posicion, width=2,edge_color='b')
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial',
                       font_color='y', font_weight='bold')

plot.axis('off')
plot.savefig("Graf2.eps")
plot.show(A)

```

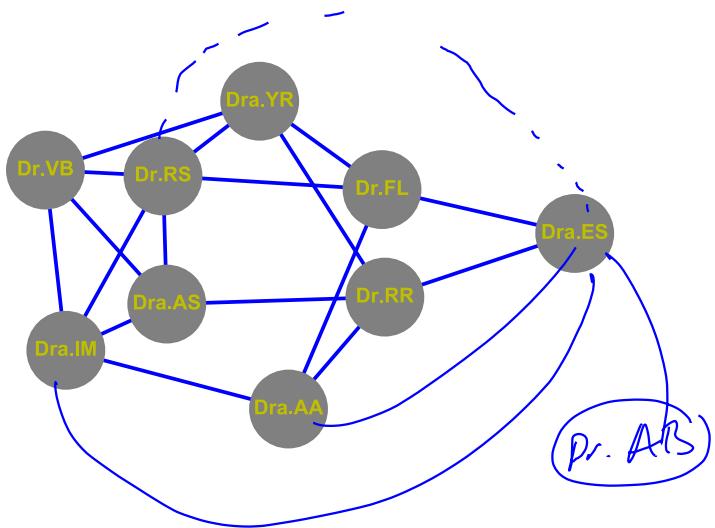


Figura 2: Grafo simple no dirigido cíclico

3. Grafo simple no dirigido reflexivo

Una de las aplicaciones de este tipo de grafo es en la modelación de comportamientos de elementos sociales en la vida real.

Por ejemplo, en un estudio sobre el comportamiento sexual de un grupo de adolescentes con edades comprendida entre 15 y 18 años, se representar las relaciones sexuales consentidas(aristas) que existen entre los individuos(vértices) del grupo, así como la satisfacción, para así saber tendencias por edades, posibles esquemas de propagación de enfermedades y promiscuidad entre otros factores del interés de los sexólogos. Este ejemplo se muestra en el grafo de la figura 3.

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

A=nx.Graph()
A.add_edge('Luis(15)', 'Maria(15)')
A.add_edge('Luis(15)', 'Luis(15)')
A.add_edge('Maria(15)', 'Yanet(17)')
A.add_edge('Yanet(17)', 'Yanet(17)')
A.add_edge('Maria(15)', 'Ferndo(16)')
A.add_edge('Ferndo(16)', 'Desisy(18)')
A.add_edge('Desisy(18)', 'Pedro(17)')
A.add_edge('Pedro(17)', 'Pedro(17)')
A.add_edge('Pedro(17)', 'Julia(18)')
A.add_edge('Pedro(17)', 'Rosi(16)')
A.add_edge('Ferndo(16)', 'Claudia(16)')
nodes_reflex = {'Luis(15)', 'Ferndo(16)', 'Julia(18)'}
nodes_no_reflex = {'Maria(15)', 'Yanet(17)', 'Rosi(16)', 'Desisy(18)', 'Pedro(17)', 'Claudia(16)'}

posicion=nx.spring_layout(A)
nx.draw_networkx_nodes(A,posicion,nodelist=nodes_reflex,
                      node_size=1500, node_color= 'r')
nx.draw_networkx_nodes(A,posicion,nodelist=nodes_no_reflex,
                      node_size=1500, node_color= 'y')
nx.draw_networkx_edges(A,posicion, width=2,edge_color='b')
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial',
                       font_color='Black', font_weight='bold')

plot.axis('off')
plot.savefig("Graf3.eps")
plot.show(A)
```

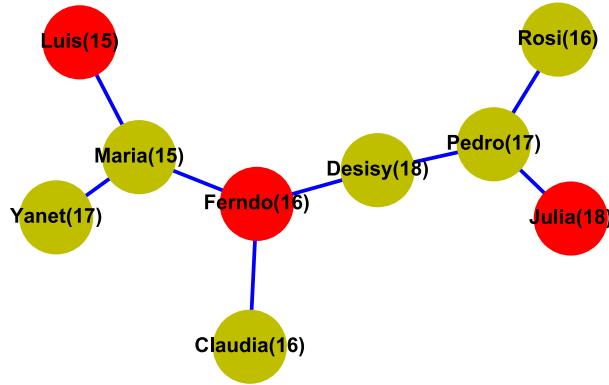


Figura 3: Grafo simple no dirigido reflexivo, donde los vértices rojos representan los nodos donde están las aristas reflexivas

4. Grafo simple dirigido acíclico

Encontraos entre las aplicaciones de los grafos dirigidos o dígrafos acíclico las siguientes:

- Una red bayesiana queda especificada formalmente por una dupla $B = (G, O)$, donde G es un grafo dirigido acíclico (GDA) y O es el conjunto de distribuciones de probabilidad. Definimos un grafo como un par $G = (V, E)$, donde V es un conjunto finito de vértices nodos o variables y E es un subconjunto del producto cartesiano $V \times V$ de pares ordenados de nodos que llamamos enlaces o aristas[5]. *IS times \$*
- Los arboles dirigidos son un ejemplo clásico de grafos dirigidos acíclico, estos tienen múltiples en la cotidianidad como pueden ser los arboles genealógicos, los organigramas de una empresa (referido a las jerarquías entre los empleados) y el árbol de directorios de Windows.

En la figura 4 se muestra un ejemplo de red bayesiana (topología de red para el cáncer de pulmón), donde C(cáncer), Con(Contaminación), F(Fumador), D(disnea), Rx(rayos-x) son los vértices y sus relaciones son las aristas.

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

```
A=nx.DiGraph()
```

```

A.add_edge('Con','C')
A.add_edge('F','C')
A.add_edge('C','D')
A.add_edge('C','Rx')

posicion=nx.spring_layout(A)
nx.draw_networkx_nodes(A,posicion, node_size=500, node_color=
'grey',alpha=0.9)

nx.draw_networkx_edges(A,posicion, width=2,edge_color='black')
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial',
font_color='y', font_weight='bold')

plot.axis('off')
plot.savefig("Graf4.eps")
plot.show(A)

```

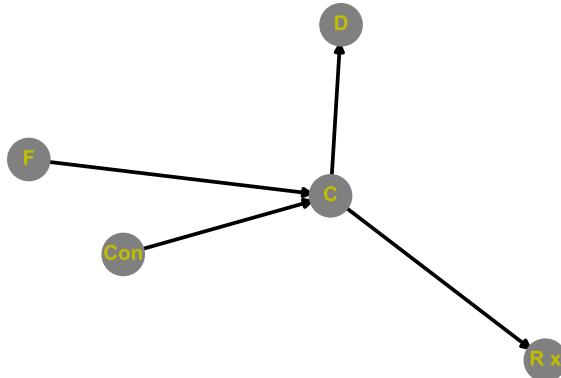


Figura 4: Grafo simple dirigido acíclico

5. Grafo simple dirigido cíclico

Aplicaciones de los ~~grafo~~ dirigido ~~cíclico~~:

- En ingeniería eléctrica se utilizan grafos dirigidos cíclicos en el análisis de circuito desde Kirchoff en los años 1850.

- En redes sociales, otro enfoque de redes sociales en su análisis puede arrojar un grafo dirigido cíclico si te tomamos como vértices a personas y como aristas el sentimiento de amistad de una persona hacia otra, este ejemplo se refleja en el grafo de la figura 5.

```

import networkx as nx
import matplotlib.pyplot as plot

A=nx.DiGraph()
A.add_edge('Ana','Felix')
A.add_edge('Ana','Alberto')
A.add_edge('Alberto','Yanet')
A.add_edge('Alberto','Fernando')
A.add_edge('Yanet','Roger')
A.add_edge('Teresa','Roger')
A.add_edge('Felix','Marta')
A.add_edge('Marta','Fernando')
A.add_edge('Yanet','Ana')
A.add_edge('Marta','Ana')

posicion=nx.spring_layout(A)

nx.draw_networkx_nodes(A,posicion,
                      node_size=500, node_color= 'y')
nx.draw_networkx_edges(A,posicion, width=2,edge_color='b')
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial',
                      font_color='Black', font_weight='bold')

plot.axis('off')
plot.savefig("Graf5.eps")
plot.show(A)

```

6. Grafo simple dirigido reflexivo

Las aplicaciones de estos grafos van desde la representación del funcionamiento de las páginas web, el modelado de una empresa de servicio que brinda el mismo tanto a sus clientes como a ella misma, hasta representar un grafo que modele la comprobación de una red de computadoras.

Por ejemplo, suponiendo que se está comprobando la conectividad entre los nodos(equipos) de una red de computadoras, es decir que un nodo puede dar ping a cualquier otro nodo, significa que dicho nodo está conectado con el resto, además se debe asegurar que el mismo nodo pueda recibir un auto-ping, si esto no ocurre existe un problema de conectividad. La figura 6 muestra dicho ejemplo.

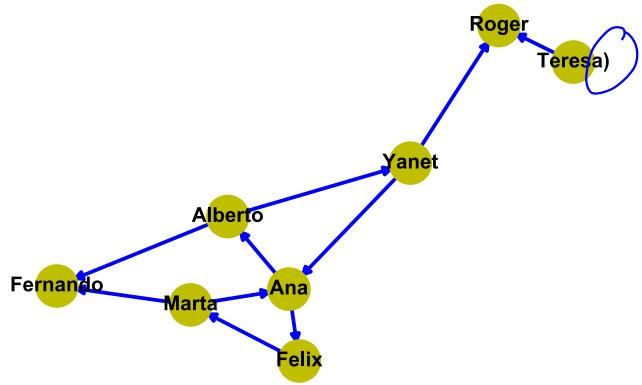


Figura 5: Grafo simple dirigido cíclico

```

import networkx as nx
import matplotlib.pyplot as plot

# Grafo simple no dirigido acclico= A
A=nx.DiGraph()
A.add_edge('Pc1','Pc1')
A.add_edge('Pc1','Pc2')
A.add_edge('Pc1','Pc3')
A.add_edge('Pc1','Pc4')
A.add_edge('Pc1','Pc5')
A.add_edge('Pc1','Pc6')
A.add_edge('Pc1','Pc7')
nodes_reflex = {'Pc1'}
nodes_no_reflex = {'Pc2', 'Pc3', 'Pc4', 'Pc5', 'Pc6', 'Pc6', 'Pc7'}
posicion=nx.spring_layout(A)
nx.draw_networkx_nodes(A,posicion,nodelist=nodes_reflex,
                      node_size=500, node_color= 'r')
nx.draw_networkx_nodes(A,posicion,nodelist=nodes_no_reflex,
                      node_size=500, node_color= 'y')
nx.draw_networkx_edges(A,posicion, width=2)
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial')

plot.axis('off')
plot.savefig("Graf6.eps")
plot.show(A)
  
```

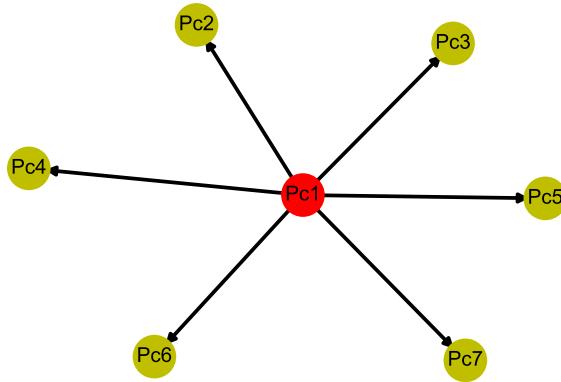


Figura 6: Grafo simple dirigido reflexivo, donde el vértice rojo representa la arista reflexiva

7. Multigrafo no dirigido acíclico

Una de las aplicaciones de este tipo de grafo es en el tasado de rutas. Por ejemplo, considerando que se quiere trazar los diferentes caminos (sin importar el sentido de estos) que comunican a varios Municipios de La Habana en un orden específico (Lisa, Marianao, Playa, Vedado, Habana Vieja, Habana del Este). Tomando como vértices los municipios y como aristas las carreteras que los unen, esto da lugar al grafo que muestra la figura 7.

```

import networkx as nx
import matplotlib.pyplot as plot

A=nx.MultiGraph()
A.add_edge('Lisa','Marianao', weight=2)
A.add_edge('Marianao','Playa', weight=2)
A.add_edge('Marianao','Playa', weight=4)
A.add_edge('Playa','Vedado', weight=2)
A.add_edge('Vedado','Habana Vieja', weight=2)
A.add_edge('Vedado','Habana Vieja', weight=4)

```

```

A.add_edge('Vedado','Habana Vieja', weight=5)
A.add_edge('Habana Vieja','Habana del Este', weight=2)
A.add_edge('Habana Vieja','Habana del Este', weight=4)

black=[('Lisa','Marianao'),('Playa','Vedado'),('Vedado','Habana Vieja'),
      ('Habana Vieja','Habana del Este'),('Marianao','Playa')]
red=[('Marianao','Playa'),('Vedado','Habana Vieja'),
     ('Habana Vieja','Habana del Este')]
bl=[('Vedado','Habana Vieja')]
posicion=nx.spring_layout(A)

nx.draw_networkx_nodes(A,posicion,
                       node_size=500, node_color= 'y')
nx.draw_networkx_edges(A, posicion, edgelist=bl, width=10, alpha=0.5,
                      edge_color='b')
nx.draw_networkx_edges(A, posicion, edgelist=red, width=5, alpha=0.5,
                      edge_color='r')
nx.draw_networkx_edges(A, posicion, edgelist=black, width=2,
                      edge_color='Black')
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial',
                       font_color='Black', font_weight='bold')

plot.axis('off')
plot.savefig("Graf7.eps")
plot.show(A)

```

8. Multigrafo no dirigido cíclico

Un ejemplo donde podemos utilizar este tipo de grafos es: se quiere representar cuantas llamadas(aristas)se realizaron entre un grupo de personas(vértices), donde nos importa saber la duración de cada llamada durante un día determinado. Este grafo se ve representado en la figura 8.

```

import networkx as nx
import matplotlib.pyplot as plot

A=nx.MultiGraph()
A.add_edge('Tel_1','Tel_2', weight=2)
A.add_edge('Tel_1','Tel_2', weight=3)
A.add_edge('Tel_1','Tel_2', weight=4)
A.add_edge('Tel_1','Tel_3', weight=2)
A.add_edge('Tel_1','Tel_4', weight=4)
A.add_edge('Tel_2','Tel_4', weight=2)
A.add_edge('Tel_2','Tel_4', weight=3)

```



Figura 7: Multigrafo no dirigido acíclico, donde los diferentes colores de los arcos representan los diferentes caminos

```

A.add_edge('Tel_2','Tel_3', weight=2)
A.add_edge('Tel_3','Tel_5', weight=2)
A.add_edge('Tel_3','Tel_5', weight=3)
A.add_edge('Tel_3','Tel_5', weight=4)
A.add_edge('Tel_5','Tel_6', weight=2)
A.add_edge('Tel_5','Tel_6', weight=3)
A.add_edge('Tel_1','Tel_6', weight=2)
A.add_edge('Tel_1','Tel_6', weight=3)
black=[('Tel_1','Tel_2'), ('Tel_1','Tel_3'), ('Tel_2','Tel_4'),
      ('Tel_2','Tel_3'), ('Tel_3','Tel_5'), ('Tel_3','Tel_5'),
      ('Tel_5','Tel_6'), ('Tel_1','Tel_6')]

red=[('Tel_1','Tel_2'), ('Tel_2','Tel_4'),
     ('Tel_1','Tel_2'), ('Tel_1','Tel_6'), ('Tel_5','Tel_6')]
bl=[('Tel_1','Tel_2'), ('Tel_3','Tel_5')]

posicion=nx.spring_layout(A)

nx.draw_networkx_nodes(A,posicion,
                       node_size=1000, node_color= 'y')
nx.draw_networkx_edges(A, posicion, edgelist=bl, width=10, alpha=0.5,
                      edge_color='b')
nx.draw_networkx_edges(A, posicion, edgelist=red, width=5, alpha=0.5,
                      edge_color='r')
nx.draw_networkx_edges(A, posicion, edgelist=black, width=2,

```

```

edge_color='Black')
nx.draw_networkx_labels(A, posicion, font_size=11, font_family='arial',
font_color='Black', font_weight='bold')

plot.axis('off')
plot.savefig("Graf8.eps")
plot.show(A)

```

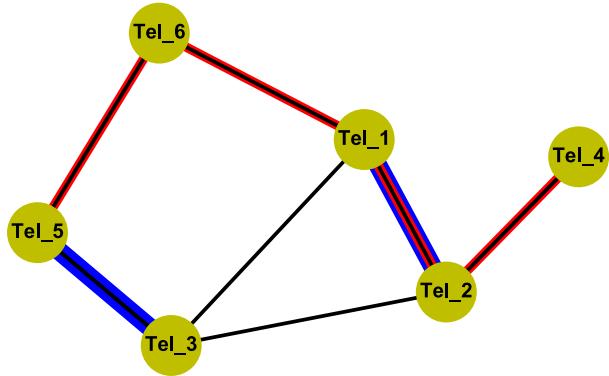


Figura 8: Multigrafo no dirigido cíclico, donde las aristas de diferentes colores representan la existencia de las llamadas con distinta duración

9. Multigrafo no dirigido reflexivo

Partiendo del ejemplo de la sección 3 se pretende construir un grafo más informativo a la hora de su análisis, por lo que se le agrega como arista (el hecho que las relaciones sexuales mantenidas fueran dentro de una relación formal). Este grafo se muestra en la figura 9.

```

import networkx as nx
import matplotlib.pyplot as plot

```

```
A=nx.MultiDiGraph()
```

```

A.add_edge('Luis(15)', 'Luis(15)', weight=2)
A.add_edge('Maria(15)', 'Yanet(17)', weight=2)
A.add_edge('Yanet(17)', 'Yanet(17)', weight=2)

A.add_edge('Maria(15)', 'Ferndo(16)', weight=2)
A.add_edge('Ferndo(16)', 'Desisy(18)', weight=3)
A.add_edge('Ferndo(16)', 'Desisy(18)', weight=2)
A.add_edge('Pedro(17)', 'Pedro(17)', weight=2)
A.add_edge('Pedro(17)', 'Julia(18)', weight=2)
A.add_edge('Pedro(17)', 'Rosi(16)', weight=3)
A.add_edge('Pedro(17)', 'Rosi(16)', weight=2)
A.add_edge('Ferndo(16)', 'Claudia(16)', weight=2)

nodes_reflex = {'Luis(15)', 'Ferndo(16)', 'Julia(18)'}
nodes_no_reflex = {'Maria(15)', 'Yanet(17)', 'Rosi(16)', 'Desisy(18)',
                   'Pedro(17)', 'Claudia(16)'}

black=[('Maria(15)', 'Yanet(17)'),
       ('Maria(15)', 'Ferndo(16)'), ('Ferndo(16)', 'Desisy(18)'),
       ('Yanet(17)', 'Julia(18)'), ('Desisy(18)', 'Pedro(17)'),
       ('Pedro(17)', 'Julia(18)'), ('Pedro(17)', 'Rosi(16)'),
       ('Ferndo(16)', 'Claudia(16)')]

bl=[('Ferndo(16)', 'Desisy(18)'), ('Pedro(17)', 'Rosi(16)')]

posicion=nx.spring_layout(A)
nx.draw_networkx_nodes(A, posicion, nodelist=nodes_reflex,
                      node_size=1800, node_color= 'r')
nx.draw_networkx_nodes(A, posicion, nodelist=nodes_no_reflex,
                      node_size=1500, node_color= 'y')
nx.draw_networkx_edges(A, posicion, edgelist=bl, width=5, alpha=0.5,
edge_color='b')
nx.draw_networkx_edges(A, posicion, edgelist=black, width=2,
edge_color='Black')
nx.draw_networkx_labels(A, posicion, font_size=11, font_family='arial',
font_color='Black', font_weight='bold')

plot.axis('off')
plot.savefig("Graf9.eps")
plot.show(A)

```

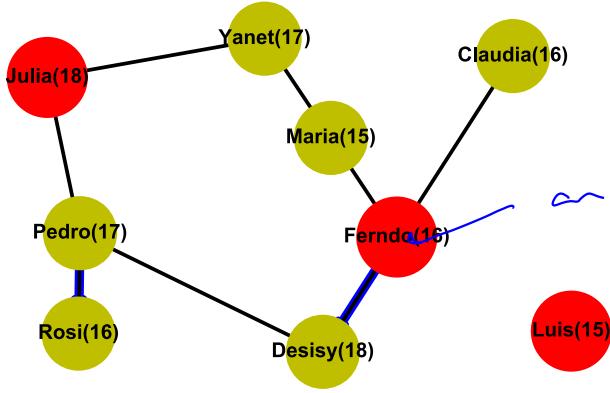


Figura 9: Multigrafo no dirigido reflexivo, donde las aristas de diferentes colores representan la existencia de una relación formal entre los individuos y los vértices de color rojo representan los vértices con aristas reflexivas .

10. Multigrafo dirigido acíclico

Considerando que se quiere trazar las diferentes rutas (teniendo en cuenta el sentido de los mismos) que llevan de una provincia a otra en Cuba (Pinar del Río, Artemisa, La Habana, Mayabeque, Matanza, Cienfuegos, Villa Clara). Tomando como vértices las provincias y como aristas las carreteras que las unen, como muestra la figura 10.

```

import networkx as nx
import matplotlib.pyplot as plot

A=nx.MultiDiGraph()
A.add_edge('PR','A', weight=2)
A.add_edge('A','LH', weight=2)
A.add_edge('A','LH', weight=4)
A.add_edge('LH','May', weight=2)
A.add_edge('May','Mat', weight=2)
A.add_edge('May','Mat', weight=4)
A.add_edge('LH','May', weight=4)
A.add_edge('LH','May', weight=5)
A.add_edge('May','Mat', weight=5)
A.add_edge('Mat','C', weight=2)
A.add_edge('Mat','C', weight=4)

```

```

black=[('PR', 'A'), ('LH', 'May'), ('May', 'Mat'),
      ('Mat', 'C'), ('A', 'LH')]
red=[('A', 'LH'), ('May', 'Mat'),
     ('Mat', 'C'), ('LH', 'May')]
bl=[('May', 'Mat'), ('LH', 'May')]
posicion=nx.spring_layout(A)

nx.draw_networkx_nodes(A,posicion,
                      node_size=500, node_color= 'y')
nx.draw_networkx_edges(A, posicion, edgelist=bl, width=10, alpha=0.5,
edge_color='b')
nx.draw_networkx_edges(A, posicion, edgelist=red, width=5, alpha=0.5,
edge_color='r')
nx.draw_networkx_edges(A, posicion, edgelist=black, width=2,
edge_color='Black')
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial',
font_color='Black', font_weight='bold')

plot.axis('off')
plot.savefig("Graf10.eps")
plot.show(A)

```

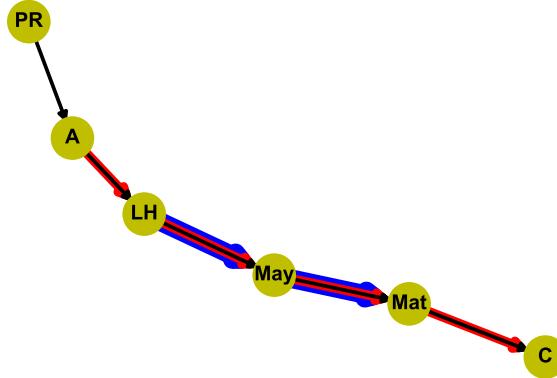


Figura 10: Multigrafo dirigido acíclico, donde las aristas de diferentes colores representan la existencia de más de una ruta entre las diferentes provincias .

11. Multigrafo dirigido cíclico

. En el caso particular de que las redes reflejen una realidad social, los nodos pueden representar personas o entidades relacionadas con sus contextos, y las conexiones representarán relaciones sociales existentes entre ellos (amistad, parentesco, membresía, afinidad, etc.). A pesar de que intuitivamente las redes sociales se asemejan a los grafos matemáticos, es más habitual que en ellas se trabaje con distintos tipos de relaciones”[3] por lo que es necesario la utilización de multígrafos, que es la herramienta que contempla más de una relación entre dos nodos, con esto ganamos mayor riqueza en los datos a analizar. Por ejemplo, tenemos un grupo de personas que laboran en un departamento de Informática y se hace una encuesta donde se les pide que marque cuales de tres sentimientos (respeto, afinidad, rechazo) sienten por sus compañeros de trabajo, como muestra la figura 11.

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

A=nx.MultiDiGraph()
A.add_edge('Persona1','Persona2', weight=2)
A.add_edge('Persona1','Persona2', weight=3)
A.add_edge('Persona1','Persona2', weight=4)
A.add_edge('Persona1','Persona3', weight=2)
A.add_edge('Persona1','Persona4', weight=4)
A.add_edge('Persona2','Persona4', weight=2)
A.add_edge('Persona2','Persona4', weight=3)
A.add_edge('Persona4','Persona3', weight=3)
A.add_edge('Persona3','Persona2', weight=2)
A.add_edge('Persona3','Persona5', weight=2)
A.add_edge('Persona3','Persona5', weight=3)
A.add_edge('Persona3','Persona5', weight=4)
A.add_edge('Persona5','Persona6', weight=2)
A.add_edge('Persona5','Persona6', weight=3)
A.add_edge('Persona1','Persona6', weight=2)
A.add_edge('Persona1','Persona6', weight=3)
black=[('Persona1','Persona2',
       ),('Persona1','Persona3'),('Persona2','Persona4'),
       ('Persona3','Persona2'),('Persona3','Persona5'),('Persona3','Persona5'),
       ('Persona5','Persona6'),('Persona1','Persona6'),('Persona4','Persona3')]

red=[('Persona1','Persona2'),('Persona2','Persona4'),
     ('Persona1','Persona2'),('Persona1','Persona6'),('Persona5','Persona6')]
bl=[('Persona1','Persona2'),('Persona3','Persona5')]

posicion=nx.spring_layout(A)

nx.draw_networkx_nodes(A,posicion,
```

```

    node_size=500
    , node_color= 'y')
nx.draw_networkx_edges(A, posicion, edgelist=bl, width=10, alpha=0.5,
edge_color='b')
nx.draw_networkx_edges(A, posicion, edgelist=red, width=5, alpha=0.5,
edge_color='r')
nx.draw_networkx_edges(A, posicion, edgelist=black, width=2,
edge_color='Black')
nx.draw_networkx_labels(A, posicion, font_size=11, font_family='arial',
font_color='Black', font_weight='bold')

plot.axis('off')
plot.savefig("Graf11.eps")
plot.show(A)

```

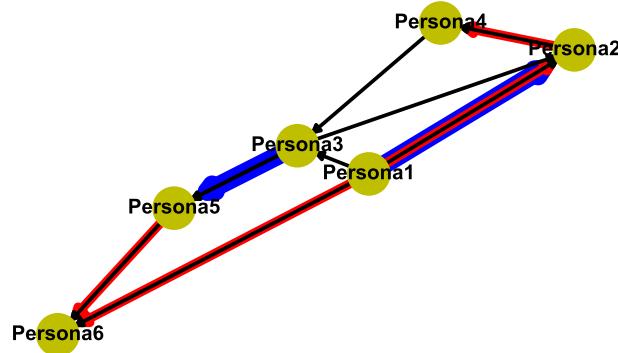
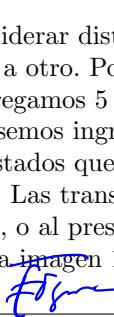


Figura 11: Multigrafo dirigido cíclico, donde las aristas de diferentes colores representan la existencia de más de una opinión sobre sus compañeros.

12. Multigrafo dirigido reflexivo

Una de las aplicaciones de estos grafos es en el modelado del funcionamiento de una máquina de estados. Por ejemplo, se desea modelar un grafo que represente el funcionamiento de una máquina muy sencilla de golosinas, que responde a las siguientes reglas:

- 1 Cada golosina vale \$ 0.25.
- 2 La máquina acepta sólo monedas de \$0.10 y de \$ 0.05.
- 3 La máquina NO da vuelto.

Para modelar el grafo, debemos considerar distintos “estados de dinero ingresado”, y como se va pasando de uno a otro. Por ejemplo, si en un momento tenemos ingresados 5 centavos, y agregamos 5 centavos más, pasamos a otro estado, que es el mismo que si hubiésemos ingresado 10 centavos al principio. Llamemos A, B, C, D, E y F a los estados que representan 0, 5, 10, 15, 20, 25 centavos ingresados respectivamente. Las transiciones de un estado a otro se harán por ingreso de 5 o 10 centavos, o al presionar el botón para obtener los caramelos (G). Como se muestra en la  imagen 12.

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

```
A=nx.MultiDiGraph()
A.add_edge('A','A', weight=2)
A.add_edge('A','B', weight=2)
A.add_edge('A','C', weight=2)
A.add_edge('B','B', weight=2)
A.add_edge('B','C', weight=2)
A.add_edge('B','D', weight=2)
A.add_edge('C','C', weight=2)
A.add_edge('C','D', weight=2)
A.add_edge('C','E', weight=2)
A.add_edge('D','D', weight=2)
A.add_edge('D','E', weight=2)
A.add_edge('D','F', weight=2)
A.add_edge('E','E', weight=2)
A.add_edge('E','F', weight=2)
A.add_edge('E','E', weight=4)
A.add_edge('F','F', weight=2)
A.add_edge('F','A', weight=2)

black=[('A','A'), ('A','B'), ('A','C'),
       ('B','B'), ('B','C'), ('B','D'), ('C','C'), ('C','D'), ('C','E'),
       ('D','D'), ('D','E'), ('D','F'), ('E','E'), ('E','F'), ('E','F'), ('F','A')]

bl=[('E','F')]
posicion=nx.spring_layout(A)

nx.draw_networkx_nodes(A,posicion,
                      node_size=500, node_color= 'R')
nx.draw_networkx_edges(A, posicion, edgelist=bl, width=10, alpha=0.5,
```

```

edge_color='b')

nx.draw_networkx_edges(A, posicion, edgelist=black, width=2,
edge_color='Black')
nx.draw_networkx_labels(A, posicion, font_size=11, font_family='arial',
font_color='Black', font_weight='bold')

plot.axis('off')
plot.savefig("Graf12.eps")
plot.show(A)

```

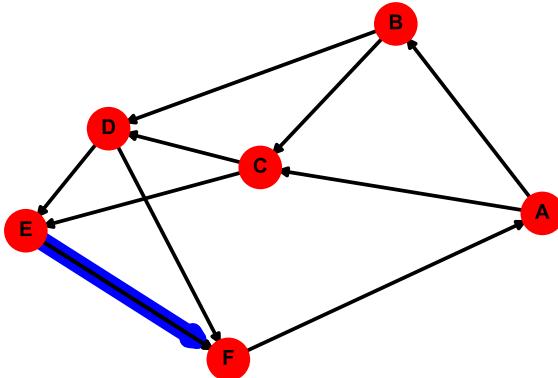


Figura 12: Multigrafo dirigido reflexivo, donde las aristas de diferentes colores representan la existencia de mas de una forma de pasar de un estado a otro.

Referencias

- [1] Amalia Duch Brown. Grafos. <https://www.cs.upc.edu/~duch/home/duch/grafos.pdf>, Octubre 2007.
- [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.

- [3] R. A. Hanneman and M. Riddle. Introduction to social network methods.
<http://faculty.ucr.edu/~hanneman/>. Consultado, September 2013.
- [4] Kimball Martin. *Graph Theory and Social Networks*. Spring, Abril 2014.
- [5] Ángel Muñoz Alamillos Mauricio Beltrán Pascuala,
Azahara Muñoz Martínez. Redes bayesianas aplicadas a problemas de credit scoring. una aplicación práctica.
<http://www.elsevier.es/en-revista-cuadernos-\economia-329-articulo-redes-bayesianas-aplicadas-problemas-credit-S0210026613000083?referer=buscador>, Octubre 2013.

Optimización de flujo en redes

Tarea 1: Representación de redes a través de la teoría de grafos

5271

26 de mayo de 2019

1. Grafo simple no dirigido acíclico

Se pueden encontrar varias aplicaciones a la modelación de un grafo simple no dirigido acíclico, una de las más claras son los árboles, “el árbol (árbol libre) que es un grafo no dirigido, conexo y acíclico. Un árbol también puede definirse como un grafo no dirigido en el que hay exactamente un camino entre todo par de vértices”[1].

Un ejemplo de usos de árboles es en topología de red la de árbol, en esta topología los nodos de la red están ubicados en forma de árbol. Esta conexión es similar a muchas redes en estrella interconectadas con la diferencia de no poseer un nodo central, en cambio posee un nodo troncal desde el cual se ramifican el resto de los nodos como se muestra en la figura 1 que es una pequeña representación con solo ocho nodos de la red que posee la UEB Rolando Pérez Gollanes entidad dedicada al sacrificio y procesamiento de aves.

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

A=nx.Graph()
A.add_edges_from([('R1','Sw1'),('Sw1','Sw2'),
                  ('Sw1','Sw3'),('Sw2','Pc2'),('Sw2','Pc3'),
                  ('Sw3','Pc4'),('Sw3','Pc5')])

posicion = nx.spring_layout(A)
nx.draw_networkx_nodes(A, posicion, node_size=800)
nx.draw_networkx_edges(A, posicion, width=2)
nx.draw_networkx_labels(A, posicion, font_size=11, font_family='arial')

plot.axis('off')
plot.savefig("Graf1.eps")
plot.show(A)
```

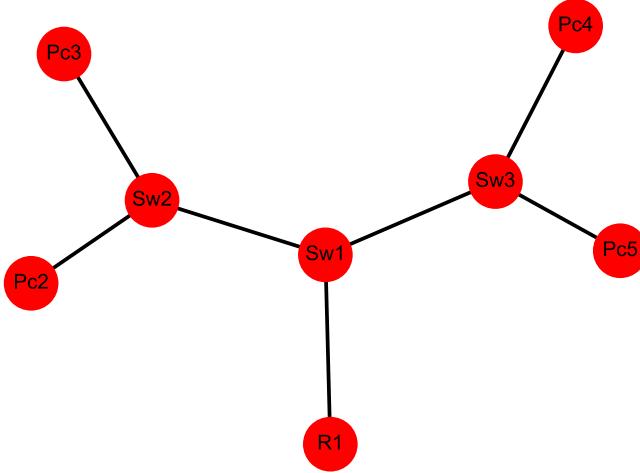


Figura 1: Grafo simple no dirigido acíclico

2. Grafo simple no dirigido cíclico

Un grafo simple no dirigido cíclico puede usarse áreas como geografía. Si se considera un mapa, digamos de Europa, que cada país sea un vértice y conecte dos vértices con una arista si esos países comparten una frontera. Un problema famoso que quedó sin resolver durante más de cien años fue el problema de los cuatro colores. Aproximadamente esto indica que cualquier mapa puede ser coloreado con a lo sumo cuatro colores de tal manera que los países adyacentes no tengan el mismo color. Este problema motivó muchos desarrollos en la teoría de grafos y finalmente se demostró con la ayuda de una computadora en 1976 [4].

Otra aplicación es en la representación de redes sociales, una red social se conceptualiza como un grafo, es decir, un conjunto de vértices (o nodos, unidades, puntos) que representan entidades u objetos sociales y un conjunto de líneas que representan una o más relaciones sociales entre ellos [2].

Por ejemplo, si consideramos un grupo de nueve doctores del núcleo académico de PISIS (Posgrado en Ingeniería de Sistemas) y construimos una red, tomando como vértices a los doctores y la colaboración de ellos en artículos publicados como aristas dará lugar a un grafo simple no dirigido cíclico como se muestra en la figura 2.

```

import networkx as nx
import matplotlib.pyplot as plot

A=nx.Graph()
A.add_edges_from([('Dr.FL','Dra.AA'),('Dr.FL','Dra.YR'),
                  ('Dr.FL','Dr.RS'), ('Dr.FL','Dra.ES'),
                  ('Dra.YR','Dr.VB'), ('Dra.YR','Dr.RR'),

```

```

('Dra.YR','Dr.RS'),('Dra.AA','Dra.IM'),
('Dra.AA','Dr.RR'),('Dra.IM','Dr.VB'),
('Dra.IM','Dra.AS'),('Dra.IM','Dr.RS'),
('Dra.IM','Dr.VB'),('Dr.VB','Dr.RS'),
('Dra.AS','Dr.VB'),('Dra.AS','Dr.RR'),
('Dra.AS','Dr.RS'),('Dra.ES','Dr.RR'))]

posicion=nx.spring_layout(A)
nx.draw_networkx_nodes(A,posicion, node_size=1700, node_color= 'grey')
nx.draw_networkx_edges(A,posicion, width=2,edge_color='b')
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial',
font_color='y', font_weight='bold')

plot.axis('off')
plot.savefig("Graf2.eps")
plot.show(A)

```

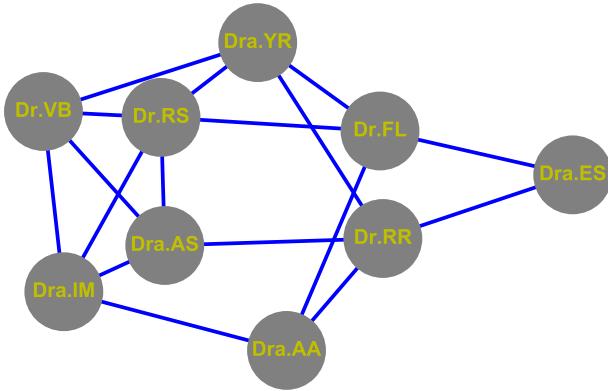


Figura 2: Grafo simple no dirigido cíclico

3. Grafo simple no dirigido reflexivo

Una de las aplicaciones de este tipo de grafo es en la modelación de comportamientos de elementos sociales en la vida real. Por ejemplo, en un estudio sobre el comportamiento sexual de un grupo de adolescentes con edades comprendida entre 15 y 18 años, se representarán las relaciones sexuales consentidas (aristas) que existen entre los individuos (vértices) del grupo, así como la satisfacción, para así saber tendencias por edades, posibles esquemas de propagación de enfermedades y promiscuidad entre otros factores de interés de los sexólogos. Este ejemplo se muestra en el grafo de la figura 3.

```

import networkx as nx
import matplotlib.pyplot as plot

A=nx.Graph()
A.add_edge('Luis(15)', 'Maria(15)')
A.add_edge('Luis(15)', 'Luis(15)')
A.add_edge('Maria(15)', 'Yanet(17)')
A.add_edge('Yanet(17)', 'Yanet(17)')
A.add_edge('Maria(15)', 'Ferndo(16)')
A.add_edge('Ferndo(16)', 'Desisy(18)')
A.add_edge('Desisy(18)', 'Pedro(17)')
A.add_edge('Pedro(17)', 'Pedro(17)')
A.add_edge('Pedro(17)', 'Julia(18)')
A.add_edge('Pedro(17)', 'Rosi(16)')
A.add_edge('Ferndo(16)', 'Claudia(16)')
nodes_reflex = {'Luis(15)', 'Ferndo(16)', 'Julia(18)'}
nodes_no_reflex = {'Maria(15)', 'Yanet(17)', 'Rosi(16)', 'Desisy(18)',
                   'Pedro(17)', 'Claudia(16)'}

posicion=nx.spring_layout(A)
nx.draw_networkx_nodes(A,posicion,nodelist=nodes_reflex,
                      node_size=1500, node_color= 'r')
nx.draw_networkx_nodes(A,posicion,nodelist=nodes_no_reflex,
                      node_size=1500, node_color= 'y')
nx.draw_networkx_edges(A,posicion, width=2,edge_color='b')
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial',
                       font_color='Black', font_weight='bold')

plot.axis('off')
plot.savefig("Graf3.eps")
plot.show(A)

```

4. Grafo simple dirigido acíclico

Encontraos entre las aplicaciones de los grafos dirigidos o dígrafos acíclico las siguientes:

- Una red bayesiana queda especificada formalmente por una dupla $B = (G, O)$, donde G es un grafo dirigido acíclico (GDA) y O es el conjunto de distribuciones de probabilidad. Definimos un grafo como un par $G = (V, E)$, donde V es un conjunto finito de vértices nodos o variables y E es un subconjunto del producto cartesiano $V \times V$ de pares ordenados de nodos que llamamos enlaces o aristas [5].
- Los arboles dirigidos son un ejemplo clásico de grafos dirigidos acíclico, estos tienen múltiples en la cotidianidad como pueden ser los arboles genealógicos, los organigramas de una empresa (referido a las jerarquías entre los empleados) y el árbol de directorios de Windows.

En la figura 4 se muestra un ejemplo de red bayesiana (topología de red para el cáncer de pulmón),

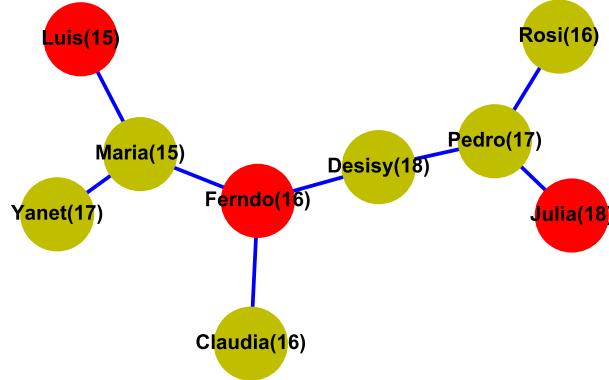


Figura 3: Grafo simple no dirigido reflexivo, donde los vértices rojos representan los nodos donde están las aristas reflexivas

donde C (cáncer), Con (Contaminación), F (Fumador), D (disnea), Rx (rayos-x) son los vértices y sus relaciones son las aristas.

```

import networkx as nx
import matplotlib.pyplot as plot
import numpy as nup
from time import time
A=nx.DiGraph()
A.add_edge('Con','C')
A.add_edge('F','C')
A.add_edge('C','D')
A.add_edge('C','R x')

tiempo=[]

for i in range(30):
    tiempo_inicial = time()
    for j in range(1600):
        d= list(nx.topological_sort(A))
    tiempo_final = time()
    tiempo_ejecucion = tiempo_final - tiempo_inicial
    tiempo.append(tiempo_ejecucion)
media=nup.mean(tiempo)
desv=nup.std(tiempo)
print (tiempo, d, media,desv)#En segundos
posicion=nx.spring_layout(A)
nx.draw_networkx_nodes(A,posicion, node_size=500, node_color= 'grey',alpha=0.9)

nx.draw_networkx_edges(A,posicion, width=2,edge_color='black')
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial',
font_color='y', font_weight='bold')

```

```

plot.axis('off')
plot.savefig("Graf4.eps")
plot.show(A)

```

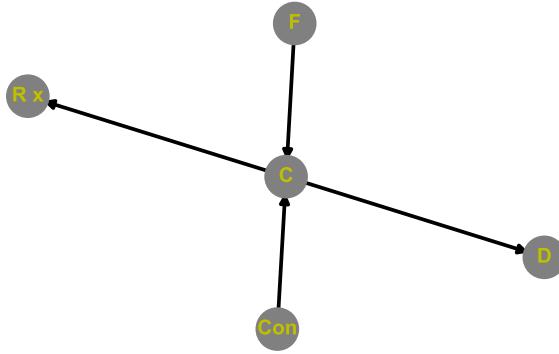


Figura 4: Grafo simple dirigido acíclico

5. Grafo simple dirigido cíclico

Aplicaciones de los grafos dirigidos cíclicos:

- En ingeniería eléctrica se utilizan grafos dirigidos cíclicos en el análisis de circuito desde Kirchoff en los años 1850.
- En redes sociales, otro enfoque de redes sociales en su análisis puede arrojar un grafo dirigido cíclico si te tomamos como vértices a personas y como aristas el sentimiento de amistad de una persona hacia otra, este ejemplo se refleja en el grafo de la figura 5.

```

import networkx as nx
import matplotlib.pyplot as plot

A=nx.DiGraph()
A.add_edge('Ana','Felix')
A.add_edge('Ana','Alberto')
A.add_edge('Alberto','Yanet')
A.add_edge('Alberto','Fernando')
A.add_edge('Yanet','Roger')
A.add_edge('Teresa','Roger')
A.add_edge('Felix','Marta')

```

```

A.add_edge('Marta','Fernando')
A.add_edge('Yanet','Ana')
A.add_edge('Marta','Ana')

posicion=nx.spring_layout(A)

nx.draw_networkx_nodes(A,posicion,
                       node_size=500, node_color= 'y')
nx.draw_networkx_edges(A,posicion, width=2,edge_color='b')
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial',
                       font_color='Black', font_weight='bold')

plot.axis('off')
plot.savefig("Graf5.eps")
plot.show(A)

```

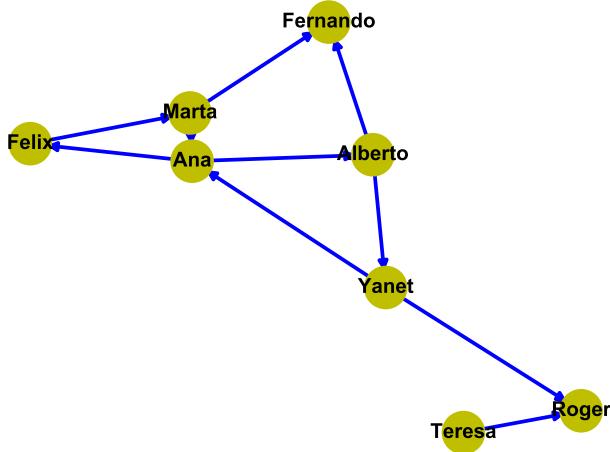


Figura 5: Grafo simple dirigido cíclico

6. Grafo simple dirigido reflexivo

Las aplicaciones de estos grafos van desde la representación del funcionamiento de las páginas web, el modelado de una empresa de servicio que brinda el mismo tanto a sus clientes como a ella misma, hasta representar un grafo que modele la comprobación de una red de computadoras. Por ejemplo, suponiendo que se está comprobando la conectividad entre los nodos (equipos) de una red de computadoras, es

decir que un nodo puede dar ping a cualquier otro nodo, significa que dicho nodo está conectado con el resto, además se debe asegurar que el mismo nodo pueda recibir un auto-ping, si esto no ocurre existe un problema de conectividad. La figura 6 muestra dicho ejemplo.

```

import networkx as nx
import matplotlib.pyplot as plot

A=nx.DiGraph()
A.add_edge('Pc1','Pc1')
A.add_edge('Pc1','Pc2')
A.add_edge('Pc1','Pc3')
A.add_edge('Pc1','Pc4')
A.add_edge('Pc1','Pc5')
A.add_edge('Pc1','Pc6')
A.add_edge('Pc1','Pc7')
nodes_reflex = {'Pc1'}
nodes_no_reflex = {'Pc2', 'Pc3', 'Pc4', 'Pc5', 'Pc6', 'Pc7', 'Pc8'}
posicion=nx.spring_layout(A)
nx.draw_networkx_nodes(A,posicion,nodelist=nodes_reflex,
                      node_size=500, node_color= 'r')
nx.draw_networkx_nodes(A,posicion,nodelist=nodes_no_reflex,
                      node_size=500, node_color= 'y')
nx.draw_networkx_edges(A,posicion, width=2)
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial')

plot.axis('off')
plot.savefig("Graf6.eps")
plot.show(A)

```

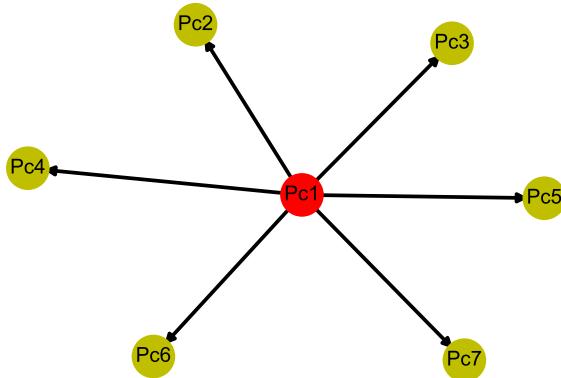


Figura 6: Grafo simple dirigido reflexivo, donde el vértice rojo representa la arista reflexiva

7. Multigrafo no dirigido acíclico

Una de las aplicaciones de este tipo de grafo es en el tasado de rutas. Por ejemplo, considerando que se quiere trazar los diferentes caminos (sin importar el sentido de estos) que comunican a varios Municipios de La Habana en un orden específico (Lisa, Marianao, Playa, Vedado, Habana Vieja, Habana del Este). Tomando como vértices los municipios y como aristas las carreteras que los unen, esto da lugar al grafo que muestra la figura 7.

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

A=nx.MultiGraph()
A.add_edge('Lisa','Marianao', weight=2)
A.add_edge('Marianao','Playa', weight=2)
A.add_edge('Marianao','Playa', weight=4)
A.add_edge('Playa','Vedado', weight=2)
A.add_edge('Vedado','Habana_Vieja', weight=2)
A.add_edge('Vedado','Habana_Vieja', weight=4)
A.add_edge('Vedado','Habana_Vieja', weight=5)
A.add_edge('Habana_Vieja','Habana_del_Este', weight=2)
A.add_edge('Habana_Vieja','Habana_del_Este', weight=4)

black=[('Lisa','Marianao'), ('Playa','Vedado'), ('Vedado','Habana_Vieja'),
      ('Habana_Vieja','Habana_del_Este'), ('Marianao','Playa')]
red=[('Marianao','Playa'), ('Vedado','Habana_Vieja'),
     ('Habana_Vieja','Habana_del_Este')]
bl=[('Vedado','Habana_Vieja')]
posicion=nx.spring_layout(A)

nx.draw_networkx_nodes(A,posicion,
                      node_size=500, node_color= 'y')
nx.draw_networkx_edges(A, posicion, edgelist=bl, width=10, alpha=0.5,
edge_color='b')
nx.draw_networkx_edges(A, posicion, edgelist=red, width=5, alpha=0.5,
edge_color='r')
nx.draw_networkx_edges(A, posicion, edgelist=black, width=2,
edge_color='Black')
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial',
                      font_color='Black', font_weight='bold')

plot.axis('off')
plot.savefig("Graf7.eps")
plot.show(A)
```

8. Multigrafo no dirigido cíclico

Un ejemplo donde podemos utilizar este tipo de grafos es: se quiere representar cuantas llamadas (aristas)se realizaron entre un grupo de personas (vértices), donde nos importa saber la duración de

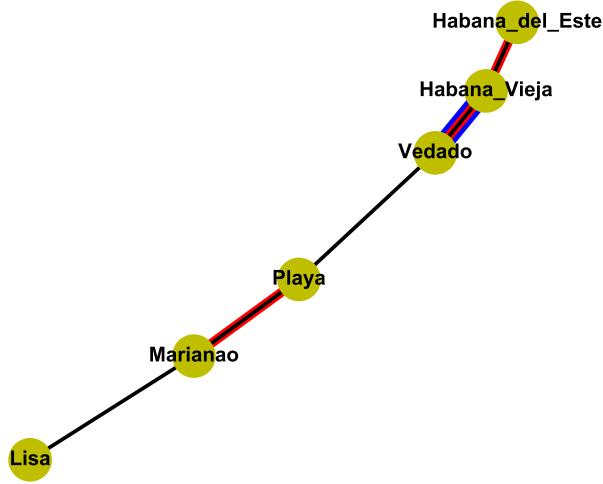


Figura 7: Multigrafo no dirigido acíclico, donde los diferentes colores de los arcos representan los diferentes caminos

cada llamada durante un día determinado. Este grafo se ve representado en la figura 8.

```

import networkx as nx
import matplotlib.pyplot as plot

A=nx.MultiGraph()
A.add_edge('Tel_1','Tel_2', weight=2)
A.add_edge('Tel_1','Tel_2', weight=3)
A.add_edge('Tel_1','Tel_2', weight=4)
A.add_edge('Tel_1','Tel_3', weight=2)
A.add_edge('Tel_1','Tel_4', weight=4)
A.add_edge('Tel_2','Tel_4', weight=2)
A.add_edge('Tel_2','Tel_4', weight=3)
A.add_edge('Tel_2','Tel_3', weight=2)
A.add_edge('Tel_3','Tel_5', weight=2)
A.add_edge('Tel_3','Tel_5', weight=3)
A.add_edge('Tel_3','Tel_5', weight=4)
A.add_edge('Tel_5','Tel_6', weight=2)
A.add_edge('Tel_5','Tel_6', weight=3)
A.add_edge('Tel_1','Tel_6', weight=2)
A.add_edge('Tel_1','Tel_6', weight=3)
black=[('Tel_1','Tel_2'), ('Tel_1','Tel_3'), ('Tel_2','Tel_4'),
      ('Tel_2','Tel_3'), ('Tel_3','Tel_5'), ('Tel_3','Tel_5'),
      ('Tel_5','Tel_6'), ('Tel_1','Tel_6')]

red=[('Tel_1','Tel_2'), ('Tel_2','Tel_4'),
     ('Tel_1','Tel_2'), ('Tel_1','Tel_6'), ('Tel_5','Tel_6')]

```

```

bl=[('Tel_1','Tel_2'),('Tel_3','Tel_5')]

posicion=nx.spring_layout(A)

nx.draw_networkx_nodes(A,posicion,
                      node_size=1000, node_color= 'y')
nx.draw_networkx_edges(A, posicion, edgelist=bl, width=10, alpha=0.5,
edge_color='b')
nx.draw_networkx_edges(A, posicion, edgelist=red, width=5, alpha=0.5,
edge_color='r')
nx.draw_networkx_edges(A, posicion, edgelist=black, width=2,
edge_color='Black')
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial',
font_color='Black', font_weight='bold')

plot.axis('off')
plot.savefig("Graf8.eps")
plot.show(A)

```

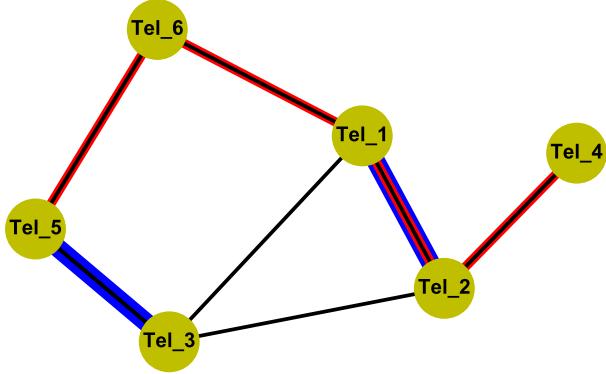


Figura 8: Multigrafo no dirigido cíclico, donde las aristas de diferentes colores representan la existencia de las llamadas con distinta duración

9. Multigrafo no dirigido reflexivo

Partiendo del ejemplo de la sección 3 se pretende construir un grafo más informativo a la hora de su análisis, por lo que se le agrega como arista (el hecho que las relaciones sexuales mantenidas fueran dentro de una relación formal). Este grafo se muestra en la figura 9.

```

import networkx as nx
import matplotlib.pyplot as plot

```

```

A=nx.MultiDiGraph()

A.add_edge('Luis(15)', 'Luis(15)', weight=2)
A.add_edge('Maria(15)', 'Yanet(17)', weight=2)
A.add_edge('Yanet(17)', 'Yanet(17)', weight=2)

A.add_edge('Maria(15)', 'Fernando(16)', weight=2)
A.add_edge('Fernando(16)', 'Desisy(18)', weight=3)
A.add_edge('Fernando(16)', 'Desisy(18)', weight=2)
A.add_edge('Pedro(17)', 'Pedro(17)', weight=2)
A.add_edge('Pedro(17)', 'Julia(18)', weight=2)
A.add_edge('Pedro(17)', 'Rosi(16)', weight=3)
A.add_edge('Pedro(17)', 'Rosi(16)', weight=2)
A.add_edge('Fernando(16)', 'Claudia(16)', weight=2)

nodes_reflex = {'Luis(15)', 'Fernando(16)', 'Julia(18)'}
nodes_no_reflex = {'Maria(15)', 'Yanet(17)', 'Rosi(16)', 'Desisy(18)', 'Pedro(17)', 'Claudia(16)'}

black=[('Maria(15)', 'Yanet(17)'),
       ('Maria(15)', 'Fernando(16)'), ('Fernando(16)', 'Desisy(18)'),
       ('Yanet(17)', 'Julia(18)'), ('Desisy(18)', 'Pedro(17)'),
       ('Pedro(17)', 'Julia(18)'), ('Pedro(17)', 'Rosi(16)'),
       ('Fernando(16)', 'Claudia(16)')]

bl=[('Fernando(16)', 'Desisy(18)'), ('Pedro(17)', 'Rosi(16)')]

posicion=nx.spring_layout(A)
nx.draw_networkx_nodes(A, posicion, nodelist=nodes_reflex,
                      node_size=1800, node_color= 'r')
nx.draw_networkx_nodes(A, posicion, nodelist=nodes_no_reflex,
                      node_size=1500, node_color= 'y')
nx.draw_networkx_edges(A, posicion, edgelist=bl, width=5, alpha=0.5,
edge_color='b')
nx.draw_networkx_edges(A, posicion, edgelist=black, width=2,
edge_color='Black')
nx.draw_networkx_labels(A, posicion, font_size=11, font_family='arial',
font_color='Black', font_weight='bold')

plot.axis('off')
plot.savefig("Graf9.eps")
plot.show(A)

```

10. Multigrafo dirigido acíclico

Considerando que se quiere trazar las diferentes rutas (teniendo en cuenta el sentido de los mismas) que llevan de una provincia a otra en Cuba (Pinar del Río, Artemisa, La Habana, Mayabeque, Matanza,

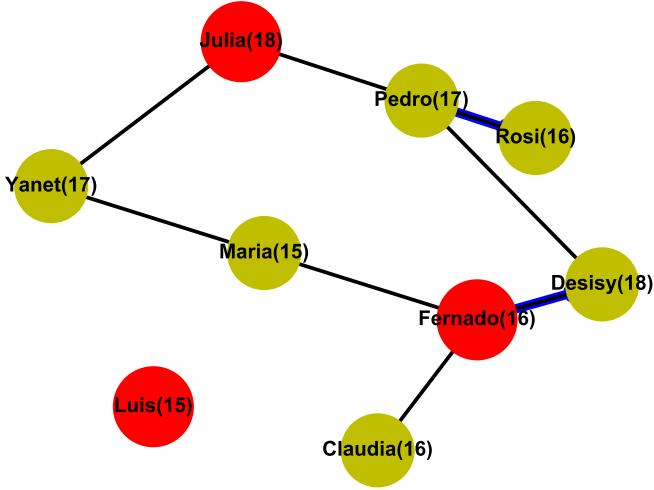


Figura 9: Multigrafo no dirigido reflexivo, donde las aristas de diferentes colores representan la existencia de una relación formal entre los individuos y los vértices de color rojo representa los vértices con aristas reflexivas

Cienfuegos, Villa Clara). Tomando como vértices las provincias y como aristas las carreteras que las unen, como muestra la figura 10.

```

import networkx as nx
import matplotlib.pyplot as plot

A=nx.MultiDiGraph()
A.add_edge('PR','A', weight=2)
A.add_edge('A','LH', weight=2)
A.add_edge('A','LH', weight=4)
A.add_edge('LH','May', weight=2)
A.add_edge('May','Mat', weight=2)
A.add_edge('May','Mat', weight=4)
A.add_edge('LH','May', weight=4)
A.add_edge('LH','May', weight=5)
A.add_edge('May','Mat', weight=5)
A.add_edge('Mat','C', weight=2)
A.add_edge('Mat','C', weight=4)

black=[('PR','A'), ('LH','May'), ('May','Mat'),
      ('Mat','C'), ('A','LH')]
red=[('A','LH'), ('May','Mat'),
     ('Mat','C'), ('LH','May')]
bl=[('May','Mat'), ('LH','May')]
posicion=nx.spring_layout(A)

```

```

nx.draw_networkx_nodes(A, posicion,
                       node_size=500, node_color= 'y')
nx.draw_networkx_edges(A, posicion, edgelist=bl, width=10, alpha=0.5,
edge_color='b')
nx.draw_networkx_edges(A, posicion, edgelist=red, width=5, alpha=0.5,
edge_color='r')
nx.draw_networkx_edges(A, posicion, edgelist=black, width=2,
edge_color='Black')
nx.draw_networkx_labels(A, posicion, font_size=11,font_family='arial',
font_color='Black', font_weight='bold')

plot.axis('off')
plot.savefig("Graf10.eps")
plot.show(A)

```

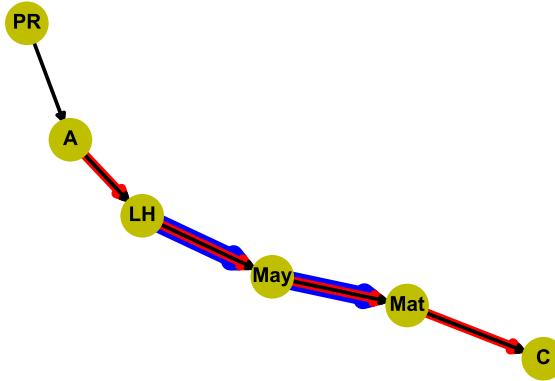


Figura 10: Multigrafo dirigido acíclico, donde las aristas de diferentes colores representan las existencia de más de una ruta entre las diferente provincias

11. Mltigrafo dirigido cíclico

“En el caso particular de que las redes reflejen una realidad social, los nodos pueden representar personas o entidades relacionadas con sus contextos, y las conexiones representarán relaciones sociales existentes entre ellos (amistad, parentesco, membresía, afinidad, etc.). A pesar de que intuitivamente las redes sociales se asemejan a los grafos matemáticos, es más habitual que en ellas se trabaje con distintos tipos de relaciones” [3] por lo que es necesario la utilización de multígrafos, que es la herramienta que contempla más de una relación entre dos nodos, con esto ganamos mayor riqueza en los datos a analizar. Por ejemplo, tenemos un grupo de personas que laboran en un departamento de Informática y se hace una encuesta donde se les pide que marque cuales de tres sentimientos (respeto, afinidad, rechazo) sienten por sus compañeros de trabajo, como muestra la figura 11.

```

import networkx as nx
import matplotlib.pyplot as plot

A=nx.MultiDiGraph()
A.add_edge('Persona1','Persona2', weight=2)
A.add_edge('Persona1','Persona2', weight=3)
A.add_edge('Persona1','Persona2', weight=4)
A.add_edge('Persona1','Persona3', weight=2)
A.add_edge('Persona1','Persona4', weight=4)
A.add_edge('Persona2','Persona4', weight=2)
A.add_edge('Persona2','Persona4', weight=3)
A.add_edge('Persona4','Persona3', weight=3)
A.add_edge('Persona3','Persona2', weight=2)
A.add_edge('Persona3','Persona5', weight=2)
A.add_edge('Persona3','Persona5', weight=3)
A.add_edge('Persona3','Persona5', weight=4)
A.add_edge('Persona5','Persona6', weight=2)
A.add_edge('Persona5','Persona6', weight=3)
A.add_edge('Persona1','Persona6', weight=2)
A.add_edge('Persona1','Persona6', weight=3)
black=[('Persona1','Persona2'), ('Persona1','Persona3'), ('Persona2','Persona4'),
      ('Persona3','Persona2'), ('Persona3','Persona5'), ('Persona3','Persona5'),
      ('Persona5','Persona6'), ('Persona1','Persona6'), ('Persona4','Persona3')]

red=[('Persona1','Persona2'), ('Persona2','Persona4'),
     ('Persona1','Persona2'), ('Persona1','Persona6'), ('Persona5','Persona6')]
bl=[('Persona1','Persona2'), ('Persona3','Persona5')]

posicion=nx.spring_layout(A)

nx.draw_networkx_nodes(A,posicion,
                      node_size=500
                      , node_color= 'y')
nx.draw_networkx_edges(A, posicion, edgelist=bl, width=10, alpha=0.5,
edge_color='b')
nx.draw_networkx_edges(A, posicion, edgelist=red, width=5, alpha=0.5,
edge_color='r')
nx.draw_networkx_edges(A, posicion, edgelist=black, width=2,
edge_color='Black')
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial',
font_color='Black', font_weight='bold')

plot.axis('off')
plot.savefig("Grafi11.eps")
plot.show(A)

```

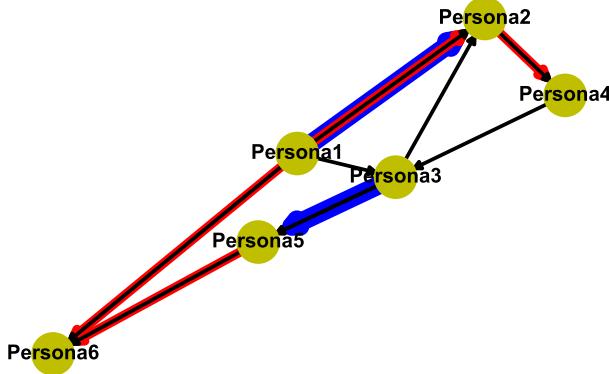


Figura 11: Multigrafo dirigido cíclico, donde las aristas de diferentes colores representan la existencia de más de una opinión sobre sus compañeros

12. Multigrafo dirigido reflexivo

Una de las aplicaciones de estos grafos es en el modelado del funcionamiento de una máquina de estados. Por ejemplo, se desea modelar un grafo que represente el funcionamiento de una máquina muy sencilla de golosinas, que responde a las siguientes reglas:

- 1 Cada golosina vale \$ 0.25.
- 2 La máquina acepta sólo monedas de \$ 0.10 y de \$ 0.05.
- 3 La máquina NO da vuelto.

Para modelar el grafo, debemos considerar distintos “estados de dinero ingresado”, y como se va pasando de uno a otro. Por ejemplo, si en un momento tenemos ingresados 5 centavos, y agregamos 5 centavos más, pasamos a otro estado, que es el mismo que si hubiésemos ingresado 10 centavos al principio. Llámemos A, B, C, D, E y F a los estados que representan 0, 5, 10, 15, 20, 25 centavos ingresados respectivamente. Las transiciones de un estado a otro se harán por ingreso de 5 o 10 centavos, o al presionar el botón para obtener los caramelos (G). Como se muestra en la figura 12.

```

import networkx as nx
import matplotlib.pyplot as plot

A=nx.MultiDiGraph()
A.add_edge('A','A', weight=2)
A.add_edge('A','B', weight=2)
A.add_edge('A','C', weight=2)
A.add_edge('B','B', weight=2)
A.add_edge('B','C', weight=2)

```

```

A.add_edge('B','D', weight=2)
A.add_edge('C','C', weight=2)
A.add_edge('C','D', weight=2)
A.add_edge('C','E', weight=2)
A.add_edge('D','D', weight=2)
A.add_edge('D','E', weight=2)
A.add_edge('D','F', weight=2)
A.add_edge('E','E', weight=2)
A.add_edge('E','F', weight=2)
A.add_edge('E','F', weight=4)
A.add_edge('F','F', weight=2)
A.add_edge('F','A', weight=2)

black=[('A','A'), ('A','B'), ('A','C'),
      ('B','B'), ('B','C'), ('B','D'), ('C','C'), ('C','D'), ('C','E'),
      ('D','D'), ('D','E'), ('D','F'), ('E','E'), ('E','F'), ('E','F'), ('F','A')]

bl=[('E','F')]
posicion=nx.spring_layout(A)

nx.draw_networkx_nodes(A,posicion,
                      node_size=500, node_color= 'R')
nx.draw_networkx_edges(A, posicion, edgelist=bl, width=10, alpha=0.5,
edge_color='b')

nx.draw_networkx_edges(A, posicion, edgelist=black, width=2,
edge_color='Black')
nx.draw_networkx_labels(A,posicion, font_size=11,font_family='arial',
font_color='Black', font_weight='bold')

plot.axis('off')
plot.savefig("Graf12.eps")
plot.show(A)

```

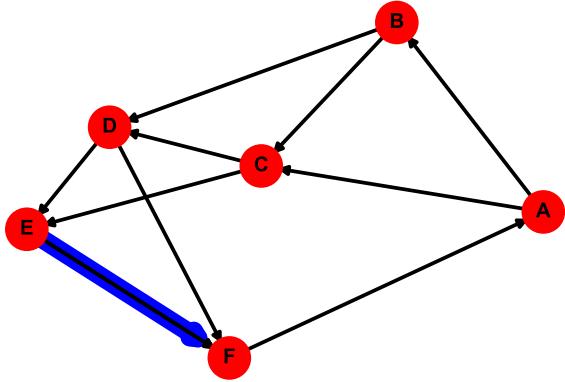
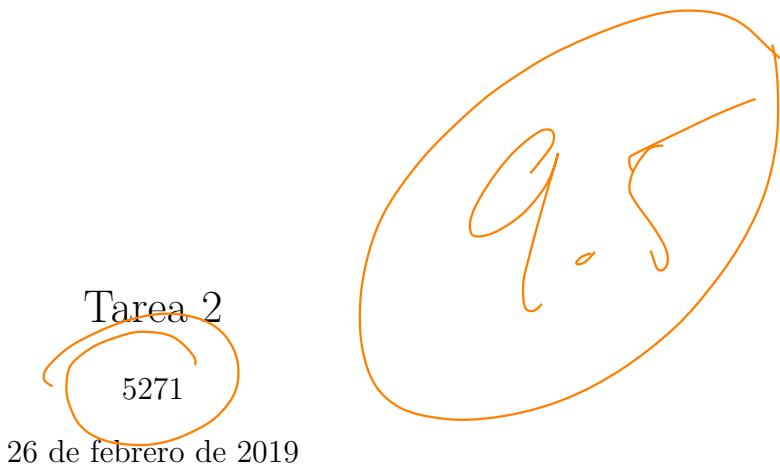


Figura 12: Multigrafo dirigido reflexivo, donde las aristas de diferentes colores representan la existencia de mas de una forma de pasar de un estado a otro

Referencias

- [1] Amalia Duch Brown. Grafos. <https://www.cs.upc.edu/~duch/home/duch/grafos.pdf>, Octubre 2007.
- [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.
- [3] R. A. Hanneman and M. Riddle. Introduction to social network methods. <http://faculty.ucr.edu/~hanneman/>. Consultado, Septiembre 2013.
- [4] Kimball Martin. *Graph Theory and Social Networks*. Spring, Abril 2014.
- [5] Azahara Muñoz Martínez Mauricio Beltrán Pascuala and Ángel Muñoz Alamillos. Redes bayesianas aplicadas a problemas de credit scoring. una aplicación práctica. <https://www.sciencedirect.com/science/article/pii/S0210026613000083>, Octubre 2013.



1. Introducción

Cuando una información se manifiesta mediante objetos y sus relaciones, visualizarla puede ser el primer paso para resolver problemas de diversa índole. La modelación y teoría de grafos constituyen la forma más común de modelar información relacional, y su visualización una forma de representar la información [1].

Sin embargo el modelado y acomodo de un grafo que refleje situaciones de la vida real resulta complejo, es por ello que se han desarrollado algoritmos dedicados a perfeccionar el trazado de los mismos según el tipo problema y grafo que representen, llamados algoritmos de diseño o *layout*.

Los algoritmos de diseño son los que devuelven una lista de posiciones para los nodos según diversos parámetros definidos para cada algoritmo, buscando cumplir con ciertos criterios estéticos como son: minimizar cruces entre aristas, maximizar el ángulo entre aristas adyacentes, maximizar el ángulo entre aristas que se cruzan, mostrar simetría, distribución uniforme de los vértices y longitud uniforme de aristas [1].

Entre los algoritmos de diseño destacan los siguientes:

- *Bipartite layout* (posiciona los nodos en dos líneas rectas, es decir que divide el conjunto de nodos en dos subconjuntos X e Y donde no existe adyacencia entre los elementos de un mismo subconjunto) [4].
- *Circular layout* (ubica los nodos en forma circular) [4].
- *Kamada kawai layout* (posiciona los nodos utilizando la función de costo de longitud de camino Kamada-Kawai) [4].
- *Random layout* (posiciona los nodos de forma aleatoria) [4].
- *Rescale layout* (reescala dado una matriz de posiciones) [4].
- *Shell layout* (posiciona los nodos en círculos concéntricos) [4].
- *Spring layout* (posiciona los nodos utilizando el algoritmo dirigido por fuerza de Fruchterman-Reingold) [4].
- *Spectral layout* (posiciona nodos utilizando los vectores propios del gráfico laplaciano) [4].

2. Grafo simple no dirigido acíclico

Se pueden encontrar varias aplicaciones a la modelación de un grafo simple no dirigido acíclico, una de las más claras son los árboles, “el árbol (árbol libre) que es un grafo no dirigido, conexo y acíclico. Un árbol también puede definirse como un grafo no dirigido en el que hay exactamente un camino entre todo par de vértices” [2].

Un ejemplo de usos de árboles es en topología de red la de árbol, en esta topología los nodos de la red están ubicados en forma de árbol. Esta conexión es similar a muchas redes en estrella interconectadas con la diferencia de no poseer un nodo central, en cambio posee un nodo troncal desde el cual se ramifican el resto de los nodos como se muestra en la figura 1 que es una pequeña representación con solo ocho nodos de la red que posee la UEB Roldano Pérez Gollanes entidad dedicada al sacrificio y procesamiento de aves.

Algoritmo de diseño.

Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 1 el algoritmo *spring layout*, dado que fue el que cumplió con la mayor cantidad de los criterios estéticos para este ejemplo.

```
1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4 A=nx.Graph()
5 A.add_edges_from([( 'R1' , 'Sw1') , ( 'Sw1' , 'Sw2') ,
6                   ( 'Sw1' , 'Sw3') , ( 'Sw2' , 'Pc2') , ( 'Sw2' , 'Pc3') ,
7                   ( 'Sw3' , 'Pc4') , ( 'Sw3' , 'Pc5')])
8
9
10 sw=[ 'R1' , 'Sw1' , 'Sw2' , 'Sw3' ]
11 pc=[ 'Pc2' , 'Pc3' , 'Pc4' , 'Pc5' ]
12 swc=[( 'R1' , 'Sw1') , ( 'Sw1' , 'Sw2') , ( 'Sw1' , 'Sw3') ]
13 pcc=[( 'Sw2' , 'Pc2') , ( 'Sw2' , 'Pc3') , ( 'Sw3' , 'Pc4') , ( 'Sw3' , 'Pc5') ]
14 lista=[ 'Sw2' , 'Pc2' , 'Pc3' , 'Sw3' , 'Pc4' , 'Pc5' , 'R1' , [ 'Sw1' ]
15 #posicion=nx.shell_layout(A, nlist=lista, scale=0.8, center=None,
16 #                           dim=2)
16 #
17 #posicion=nx.random_layout(A, center=None, dim=2)
18 #
19 #posicion=nx.circular_layout(A, scale=0.8, center=None, dim=2)
20 #
21 #posicion=nx.spectral_layout(A, weight='weight', scale=1, center=
22 #                           None, dim=2)
22
23 #posicion=nx.kamada_kawai_layout(A, dist=None, pos=None, weight='
24 #                                 weight', scale=0.5, center=None, dim=2)
24
25 posicion=nx.spring_layout(A, k=1, iterations=300, threshold=0.0001,
26 #                           weight='distans', scale=0.5)
```

```

27 nx.draw_networkx_nodes(A, posicion , node_size=800, node_shape='s' ,
28     nodelist=sw, node_color='#c9b323')
29 nx.draw_networkx_nodes(A, posicion , node_size=800, node_shape='o' ,
30     nodelist=pc, node_color='#c68282')
31 nx.draw_networkx_edges(A, posicion , width=4, edgelist=swc, style='
32     dashed',edge_vmax=1, edge_vmin=1)
33 nx.draw_networkx_edges(A, posicion , width=2, edgelist=pcc)
34 nx.draw_networkx_labels(A, posicion , font_size=11,font_family='arial
35     ')
36 plot.axis('off')
37 plot.savefig("Graf1_spring_layout.eps")
plot.show(A)

```

Graf1.py

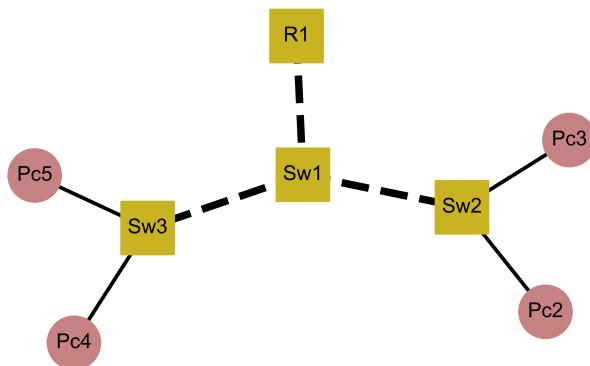


Figura 1: Grafo simple no dirigido acíclico donde las líneas discontinuas representan los cables de fibra optica y los líneas continuas los cables UTP.

3. Grafo simple no dirigido cíclico

Un grafo simple no dirigido cíclico puede usarse áreas como geografía. Si se considera un mapa, digamos de Europa: que cada país sea un vértice y conecte dos vértices con una arista si esos países comparten una frontera. Un problema famoso que quedó sin resolver durante más de cien años fue el problema de los cuatro colores. Aproximadamente esto indica que cualquier mapa puede ser coloreado con a lo sumo cuatro colores de tal manera que los países adyacentes no tengan el mismo color. Este problema motivó muchos desarrollos en la teoría de grafos y finalmente se demostró con la ayuda de una computadora en 1976 [6].

Otra aplicación es en la representación de redes sociales, una red social se conceptualiza como un grafo, es decir, un conjunto de vértices (o nodos, unidades, puntos) que representan entidades u objetos sociales y un conjunto de líneas que representan una o más relaciones sociales entre ellos [3].

Por ejemplo, si consideramos un grupo de nueve doctores del núcleo académico de PISIS (Posgrado en Ingeniería de Sistemas) y construimos una red, tomando como vértices a los doctores y la colaboración de ellos en artículos publicados como aristas dará lugar a un grafo simple no dirigido cíclico como se muestra en la figura 2.

Algoritmo de diseño.

Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 2 el algoritmo *circular layout*, ya que el mismo permite una mejor compresión del grafo del ejemplo en cuestión.

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4
5 A=nx.Graph()
6 A.add_edges_from([( 'Dr.FL' , 'Dra.AA') , ( 'Dr.FL' , 'Dra.YR') ,
7                   ( 'Dr.FL' , 'Dr.RS') , ( 'Dr.FL' , 'Dra.ES') ,
8                   ( 'Dra.YR' , 'Dr.VB') , ( 'Dra.YR' , 'Dr.RR') ,
9                   ( 'Dra.YR' , 'Dr.RS') , ( 'Dra.AA' , 'Dra.IM') ,
10                  ( 'Dra.AA' , 'Dr.RR') , ( 'Dra.IM' , 'Dr.VB') ,
11                  ( 'Dra.IM' , 'Dra.AS') , ( 'Dra.IM' , 'Dr.RS') ,
12                  ( 'Dra.IM' , 'Dr.VB') , ( 'Dr.VB' , 'Dr.RS') ,
13                  ( 'Dra.AS' , 'Dr.VB') , ( 'Dra.AS' , 'Dr.RR') ,
14                  ( 'Dra.AS' , 'Dr.RS') , ( 'Dra.ES' , 'Dr.RR') ,
15                  ( 'Dra.AA' , 'Dra.ES') , ( 'Dra.IM' , 'Dra.ES')])
16 lista=[ 'Dr.FL' , 'Dra.AA' , 'Dra.YR' , 'Dr.RR' , 'Dra.IM' , 'Dra.AS' , 'Dr.VB' ,
17        'Dra.ES' ] , [ 'Dr.RS' ]
18 #
19 #posicion=nx.shell_layout(A, nlist=lista , scale=0.8, center=None,
20 #                           dim=2)
21 #
22 #posicion=nx.random_layout(A, center=None, dim=2)
```

```

21 #
22 posicion=nx.circular_layout(A, scale=0.8, center=None, dim=2)
23 #
24 #posicion=nx.spectral_layout(A, weight='weight', scale=1, center=
25 None, dim=2)
26 #posicion=nx.kamada_kawai_layout(A, dist=None, pos=None, weight='
27 weight', scale=0.5, center=None, dim=2)
28 #posicion=nx.spring_layout(A, k=1, iterations=300, threshold
29 =0.0001, weight='distans', scale=0.5)
30 nx.draw_networkx_nodes(A, posicion, node_size=2000, node_color=range
31 (9),cmap=plot.cm.Blues)
32 nx.draw_networkx_edges(A, posicion, width=2,edge_color='#46267d')
33 nx.draw_networkx_labels(A, posicion, font_size=13,font_family='arial
34 ',
35 font_color='#846c16', font_weight='bold')
36 plot.axis('off')
37 plot.savefig("Graf2_circular_layout.eps")
38 plot.show(A)

```

Graf2.py

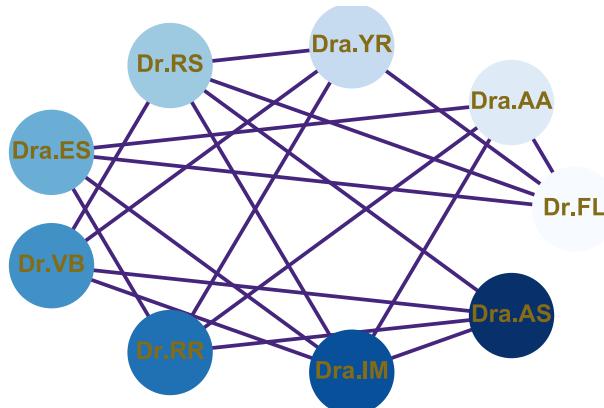


Figura 2: Grafo simple no dirigido cíclico, donde los vértices representan los diferentes doctores y las aristas la colaboración entre ellos.

4. Grafo simple no dirigido reflexivo

Una de las aplicaciones de este tipo de grafo es en la modelación de comportamientos de elementos sociales en la vida real.

Por ejemplo, en un estudio sobre el comportamiento sexual de un grupo de adolescentes con edades comprendida entre 15 y 18 años, se representar las relaciones sexuales consentidas (aristas) que existen entre los individuos (vértices) del grupo, así como la satisfacción, para así saber tendencias por edades, posibles esquemas de propagación de enfermedades y promiscuidad entre otros factores del interés de los sexólogos. Este ejemplo se muestra en el grafo de la figura 3.

Algoritmo de diseño.

Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 3 el algoritmo *kamada kawai layout*, dado que este algoritmo es uno de los más usados para la representación de grafos conexos y no dirigidos, por los resultados de acomodo tan buenos que arroja en estos tipos de grafos.

```
1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4
5 A=nx.Graph()
6 A.add_edge('Luis(15)', 'Maria(15)')
7 A.add_edge('Luis(15)', 'Luis(15)')
8 A.add_edge('Maria(15)', 'Yanet(17)')
9 A.add_edge('Yanet(17)', 'Yanet(17)')
10 A.add_edge('Maria(15)', 'Ferndo(16)')
11 A.add_edge('Ferndo(16)', 'Desisy(18)')
12 A.add_edge('Desisy(18)', 'Pedro(17)')
13 A.add_edge('Pedro(17)', 'Pedro(17)')
14 A.add_edge('Pedro(17)', 'Julia(18)')
15 A.add_edge('Pedro(17)', 'Rosi(16)')
16 A.add_edge('Ferndo(16)', 'Claudia(16)')
17 nodes_reflex = {'Luis(15)', 'Ferndo(16)', 'Julia(18)'}
18 nodes_no_reflex = {'Maria(15)', 'Yanet(17)', 'Rosi(16)', 'Desisy(18)', 'Pedro(17)', 'Claudia(16)'}
19
20 lista =[ 'Luis(15)', 'Julia(18)', 'Yanet(17)', 'Rosi(16)', 'Desisy(18)', 'Claudia(16)' ], [ 'Maria(15)', 'Ferndo(16)', 'Pedro(17)' ]
21
22 #posicion=nx.shell_layout(A, nlist=lista, scale=0.5, center=None, dim=2)
23 #posicion=nx.random_layout(A, center=None, dim=2)
24 #posicion=nx.circular_layout(A, scale=0.8, center=None, dim=2)
25 #posicion=nx.spectral_layout(A, weight='distans', scale=1, center=None, dim=2)
```

```

31 posicion=nx.kamada_kawai_layout(A, dist=None, pos=None, weight='
32     weight', scale=0.5, center=None, dim=2)
33 #posicion=nx.spring_layout(A, k=1, iterations=200, threshold
34     =0.0001, weight='weight', scale=0.50)
35 #posicion=nx.rescale_layout( pos , escala = 1 )
36 #posicion=nx.bipartite_layout(A, nodes, align='vertical', scale=1,
37     center=None, aspect_ratio=1.333333333333333)
38
39
40 nx.draw_networkx_nodes(A, posicion , nodelist=nodes_reflex ,
41     node_size=1500, node_color= '#ea6767')
42 nx.draw_networkx_nodes(A, posicion , nodelist=nodes_no_reflex ,
43     node_size=1500, node_color= 'y')
44 nx.draw_networkx_edges(A, posicion , width=4,edge_color='#85d9ce')
45
46 nx.draw_networkx_labels(A, posicion , font_size=11,font_family='arial
47     ',
48     font_color='Black', font_weight='bold')
49
50
51 plot.axis('off')
52 plot.savefig("Graf3_kamada_kawai_layout.eps")
53 plot.show(A)

```

Graf3.py

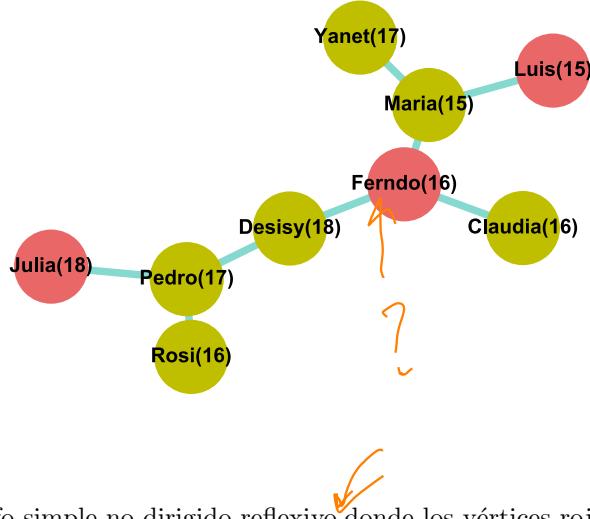


Figura 3: Grafo simple no dirigido reflexivo, donde los vértices rojos representan los nodos donde están las aristas reflexivas

5. Grafo simple dirigido acíclico

Encontraos entre las aplicaciones de los grafos dirigidos o dígrafos acíclico las siguientes:

- Una red bayesiana queda especificada formalmente por una dupla $B = (G, O)$, donde G es un grafo dirigido acíclico (GDA) y O es el conjunto de distribuciones de probabilidad. Definimos un grafo como un par $G = (V, E)$, donde V es un conjunto finito de vértices nodos o variables y E es un subconjunto del producto cartesiano $V \times V$ de pares ordenados de nodos que llamamos enlaces o aristas [7].
- Los arboles dirigidos son un ejemplo clásico de grafos dirigidos acíclico, estos tienen múltiples en la cotidianidad como pueden ser los arboles genealógicos, los organigramas de una empresa (referido a las jerarquías entre los empleados) y el árbol de directorios de Windows.

En la figura 4 se muestra un ejemplo de red bayesiana (topología de red para el cáncer de pulmón), donde C (cáncer), Con (Contaminación), F (Fumador), D (disnea), Rx (rayos-x) son los vértices y sus relaciones son las aristas.

Algoritmo de diseño.

Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 4 el algoritmo *shell layout*, dado que este algoritmo ubica los nodos en círculos concéntricos a un nodo dado, es la mejor opción para representar este ejemplo.

```

1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4 A=nx.DiGraph()
5 A.add_edge('Con','C', weight=0.8)
6 A.add_edge('F','C', weight=0.8)
7 A.add_edge('C','D', weight=3)
8 A.add_edge('C','Rx', weight=3)
9 lista=['Con','F','D','Rx'],[ 'C']
10 nodes=['Con','F','D','Rx']
11 posicion=nx.shell_layout(A, nlist=lista , scale=0.8, center=None,
    dim=2)
12 #posicion=nx.random_layout(A, center=None, dim=2)
13 #posicion=nx.circular_layout(A, scale=0.8, center=None, dim=2)
14 #posicion=nx.spectral_layout(A, weight='distans' , scale=1, center=
    None, dim=2)
15 #posicion=nx.kamada_kawai_layout(A, dist=None, pos=None, weight='
    weight' , scale=0.5, center=None, dim=2)
16 #posicion=nx.spring_layout(A, k=3, iterations=200, threshold
    =0.0001, weight='weight' , scale=1)
17 #posicion=nx.rescale_layout( pos , escala = 1 )
18
19 #posicion=nx.bipartite_layout(A, nodes, align='vertical' , scale=1,
    center=None, aspect_ratio=1.333333333333333)
20
21 nx.draw_networkx_nodes(A,posicion , node_size=700, nodelist=[ 'C'],
    node_color= '#bbf1f0' ,alpha=1)
22 nx.draw_networkx_nodes(A,posicion , nodelist= [ 'Con','F','D','Rx'],
    node_size=600, node_color= '#6ed3c5' ,alpha=1)
23 nx.draw_networkx_edges(A,posicion , width=4,edge_color=' #35665f ')
24
25 nx.draw_networkx_labels(A,posicion , font_size=13,font_family='arial
    ' ,
    font_color=' #147645' , font_weight='bold')
26
27
28
29
30
31
32
33
34
35 plot.axis('off')
36 plot.savefig("Graf4_shell_layout.eps")
37 plot.show(A)

```

Graf4.py

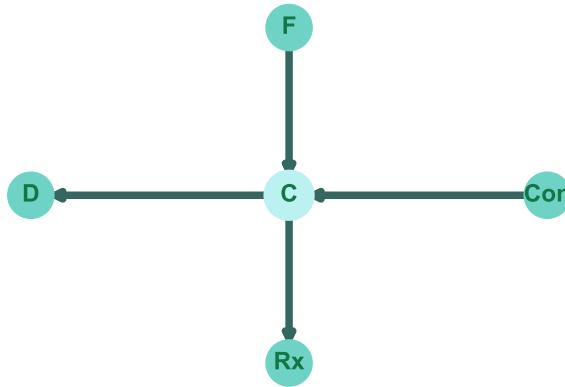


Figura 4: Grafo simple dirigido acíclico

6. Grafo simple dirigido cíclico

Aplicaciones de los grafos dirigidos cíclicos:

- En ingeniería eléctrica se utilizan grafos dirigidos cíclicos en el análisis de circuito desde Kirchoff en los años 1850.
- En redes sociales, otro enfoque de redes sociales en su análisis puede arrojar un grafo dirigido cíclico si te tomamos como vértices a personas y como aristas el sentimiento de amistad de una persona hacia otra, este ejemplo se refleja en el grafo de la figura 5.

Algoritmo de diseño.

Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 5 el algoritmo *kamada kawai layout*, a pesar de no ser este ejemplo precisamente el más adecuado para este algoritmo fue el de mejor resultado estético.

```

1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4 A=nx.DiGraph()
5 A.add_edge('Ana','Felix')
6 A.add_edge('Ana','Alberto')
7 A.add_edge('Alberto','Yanet')
8

```

```

9| A.add_edge('Alberto','Fernando')
10| A.add_edge('Yanet','Roger')
11| A.add_edge('Teresa','Roger')
12| A.add_edge('Felix','Marta')
13| A.add_edge('Marta','Fernando')
14| A.add_edge('Yanet','Ana')
15| A.add_edge('Marta','Ana')
16| lista=['Ana','Felix','Alberto','Yanet','Fernando'],[ 'Roger','Teresa',
17|   , 'Marta',]
18| #posicion=nx.shell_layout(A, nlist=lista, scale=0.50, center=None,
19|   dim=2)
20| #posicion=nx.random_layout(A, center=None, dim=2)
21| #posicion=nx.circular_layout(A, scale=0.5, center=None, dim=2)
22| #posicion=nx.spectral_layout(A, weight='distans', scale=0.50,
23|   center=None, dim=2)
24| posicion=nx.kamada_kawai_layout(A, dist=None, pos=None, weight='
25|   weight', scale=0.5, center=None, dim=2)
26| #posicion=nx.spring_layout(A, k=1, iterations=200, threshold
27|   =0.0001, weight='weight', scale=1)
28| #posicion=nx.rescale_layout( pos , escala = 1 )
29| #posicion=nx.bipartite_layout(A, nodes, align='vertical', scale=1,
30|   center=None, aspect_ratio=1.333333333333333)
31| nx.draw_networkx_nodes(A,posicion ,
32|   node_size=500, node_color= 'y')
33| nx.draw_networkx_edges(A,posicion , width=2,edge_color='b')
34| for p in posicion:
35|   posicion[p][1] += 0.05
36| nx.draw_networkx_labels(A,posicion , font_size=11,font_family='arial
37|   ',
38|     font_color='Black', font_weight='bold')
39|
40| plot.axis('off')
41| plot.savefig("Graf5_kamada_kawai_layout.eps")
42| plot.show(A)

```

Graf5.py

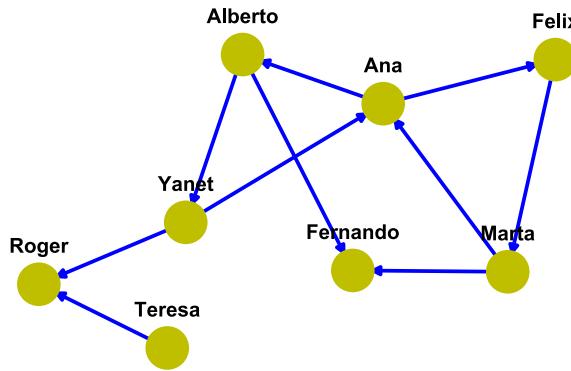


Figura 5: Grafo simple dirigido cíclico, donde los vértices representan las personas y las aristas la relación de amistad entre ellas.

7. Grafo simple dirigido reflexivo

Las aplicaciones de estos grafos van desde la representación del funcionamiento de las páginas web, el modelado de una empresa de servicio que brinda el mismo tanto a sus clientes como a ella misma, hasta representar un grafo que modele la comprobación de una red de computadoras.

Por ejemplo, suponiendo que se está comprobando la conectividad entre los nodos (equipos) de una red de computadoras, es decir que un nodo puede dar ping a cualquier otro nodo, significa que dicho nodo está conectado con el resto, además se debe asegurar que el mismo nodo pueda recibir un auto ping, si esto no ocurre existe un problema de conectividad. La figura 6 muestra dicho ejemplo.

Algoritmo de diseño.

Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 6 el algoritmo *random layout*, en este ejemplo se pudo usar este algoritmo por su simpleza ya que con grafos más complejos no es aconsejable el uso del mismo por sus malos resultados estéticamente hablando.

```

1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4

```

```

5 A=nx.DiGraph()
6 A.add_edge('Pc1','Pc1')
7 A.add_edge('Pc1','Pc2')
8 A.add_edge('Pc1','Pc3')
9 A.add_edge('Pc1','Pc4')
10 A.add_edge('Pc1','Pc5')
11 A.add_edge('Pc1','Pc6')
12 A.add_edge('Pc1','Pc7')
13 nodes_reflex = {'Pc1'}
14 nodes_no_reflex = {'Pc2', 'Pc3', 'Pc4', 'Pc5', 'Pc6', 'Pc6', 'Pc7'}
15 lista=['Pc2', 'Pc3', 'Pc4', 'Pc5', 'Pc6', 'Pc6', 'Pc7'], ['Pc1']
16
17 #posicion=nx.shell_layout(A, nlist=lista, scale=0.50, center=None,
18   dim=2)
19 posicion=nx.random_layout(A, center=None, dim=2)
20
21 #posicion=nx.circular_layout(A, scale=0.5, center=None, dim=2)
22
23 #posicion=nx.spectral_layout(A, weight='distans', scale=0.50,
24   center=None, dim=2)
25
26 #posicion=nx.kamada_kawai_layout(A, dist=None, pos=None, weight=
27   'weight', scale=0.5, center=None, dim=2)
28
29 #posicion=nx.spring_layout(A, k=1, iterations=200, threshold
30   =0.0001, weight='weight', scale=1)
31
32 #posicion=nx.rescale_layout( pos , escala = 1 )
33
34 #posicion=nx.bipartite_layout(A, {'Pc1'}, align='vertical', scale
35   =1, center=None, aspect_ratio=1.333333333333333)
36 nx.draw_networkx_nodes(A, posicion, nodelist=nodes_reflex,
37   node_size=500, node_color= 'r')
38 nx.draw_networkx_nodes(A, posicion, nodelist=nodes_no_reflex,
39   node_size=500, node_color= 'y')
40 nx.draw_networkx_edges(A, posicion, width=2)
41 nx.draw_networkx_labels(A, posicion, font_size=11,font_family='arial',
42   )
43
44 plot.axis('off')
45 plot.savefig("Graf6_random_layout.eps")
46 plot.show(A)

```

Graff6.py

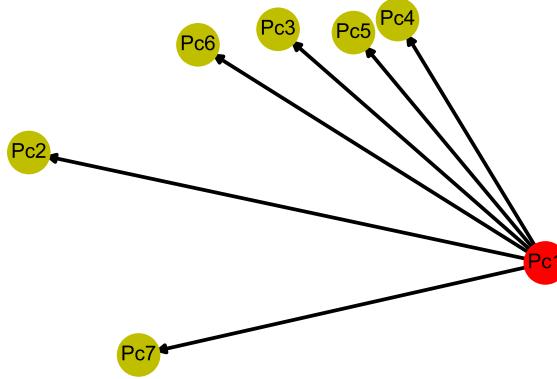


Figura 6: Grafo simple dirigido reflexivo, donde el vértice rojo representa la arista reflexiva.

8. Multigrafo no dirigido acíclico

Una de las aplicaciones de este tipo de grafo es en el tasado de rutas. Por ejemplo, considerando que se quiere trazar los diferentes caminos (sin importar el sentido de estos) que comunican a varios Municipios de La Habana en un orden específico (Lisa, Marianao, Playa, Vedado, Habana Vieja, Habana del Este). Tomando como vértices los municipios y como aristas las carreteras que los unen, esto da lugar al grafo que muestra la figura 7.

Algoritmo de diseño.

Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 7 el algoritmo *spring layout*, este acomoda los nodos con el algoritmo dirigido por fuerza de Fruchterman-Reingold que minimiza la distancia entre los vértices conectados y de mayor peso.

```

1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4
5 A=nx.MultiGraph()
6 A.add_edge('Lisa','Marianao', weight=2)
7 A.add_edge('Marianao','Playa', weight=2)
8 A.add_edge('Marianao','Playa', weight=4)
9 A.add_edge('Playa','Vedado', weight=2)
10 A.add_edge('Vedado','Habana Vieja', weight=2)

```

```

11 A.add_edge( 'Vedado' , 'Habana Vieja' , weight=4)
12 A.add_edge( 'Vedado' , 'Habana Vieja' , weight=5)
13 A.add_edge( 'Habana Vieja' , 'Habana del Este' , weight=2)
14 A.add_edge( 'Habana Vieja' , 'Habana del Este' , weight=4)
15
16 black=[( 'Lisa' , 'Marianao' ) ,( 'Playa' , 'Vedado' ) ,( 'Vedado' , 'Habana
17 Vieja' ),
18 ( 'Habana Vieja' , 'Habana del Este' ),( 'Marianao' , 'Playa' )]
19 red=[( 'Marianao' , 'Playa' ),( 'Vedado' , 'Habana Vieja' ),
20 ( 'Habana Vieja' , 'Habana del Este' )]
21 bl=[( 'Vedado' , 'Habana Vieja' )]
22
23 lista=[ 'Lisa' , 'Marianao' , 'Playa' , 'Vedado' , 'Habana Vieja' ],[ 'Habana
24 del Este' ]
25
26 #posicion=nx.shell_layout(A, nlist=lista , scale=0.50, center=None,
27 dim=2)
28
29 #posicion=nx.random_layout(A, center=None, dim=2)
30
31 #posicion=nx.circular_layout(A, scale=0.5, center=None, dim=2)
32
33 #posicion=nx.spectral_layout(A, weight='distans' , scale=0.40,
34 center=None, dim=2)
35
36 #posicion=nx.kamada_kawai_layout(A, dist=None, pos=None, weight=
37 'weight' , scale=0.30, center=None, dim=2)
38
39 posicion=nx.spring_layout(A, k=1, iterations=200, threshold=0.0001,
40 weight='weight' , scale=0.5)
41
42 #posicion=nx.rescale_layout( pos , escala = 1 )
43
44 #posicion=nx.bipartite_layout(A, { 'Marianao' , 'Vedado' , 'Habana Vieja
45 '}, align='vertical' , scale=1, center=None, aspect_ratio
46 =1.333333333333333)
47
48 nx.draw_networkx_nodes(A, posicion ,
49 node_size=800, node_color= 'y')
50 nx.draw_networkx_edges(A, posicion , edgelist=bl , width=10, alpha
51 =0.5,
52 edge_color='b')
53 nx.draw_networkx_edges(A, posicion , edgelist=red , width=5, alpha
54 =0.5,
55 edge_color='r')
56 nx.draw_networkx_edges(A, posicion , edgelist=black , width=2,
57 edge_color='Black')
58 for p in posicion :
59     posicion[p][1] -= 0.01
60 nx.draw_networkx_labels(A, posicion , font_size=11,font_family='arial
61 ' ,
62 font_color='Black' , font_weight='bold')
63
64 plot.axis( 'off' )
65 plot.savefig("Graf7_spring_layout.eps")
66 plot.show(A)

```

Graf7.py

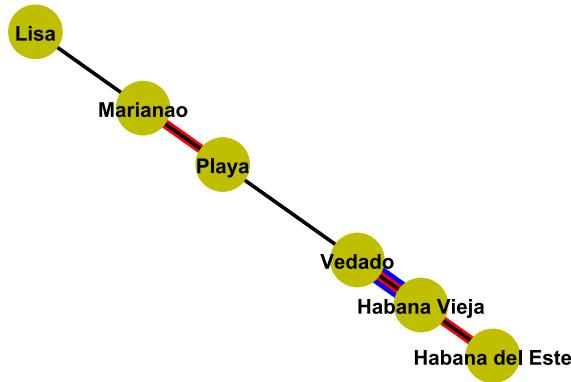


Figura 7: Multigrafo no dirigido acíclico, donde la diferencia de colores de los arcos representan los diferentes caminos.

9. Multigrafo no dirigido cíclico

Un ejemplo donde podemos utilizar este tipo de grafos es: se quiere representar cuantas llamadas (aristas) se realizaron entre un grupo de personas (vértices), donde nos importa saber la duración de cada llamada durante un día determinado. Este grafo se ve representado en la figura 8.

Algoritmo de diseño.

Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 8 el algoritmo *spectral layout*, dando buenos resultados estéticos.

```
1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4
5 A=nx.MultiGraph()
6 A.add_edge('Tel_1','Tel_2', weight=2)
7 A.add_edge('Tel_1','Tel_2', weight=3)
```

```

8 A.add_edge('Tel_1','Tel_2', weight=4)
9 A.add_edge('Tel_1','Tel_3', weight=2)
10 A.add_edge('Tel_1','Tel_4', weight=4)
11 A.add_edge('Tel_2','Tel_4', weight=2)
12 A.add_edge('Tel_2','Tel_4', weight=3)
13 A.add_edge('Tel_2','Tel_3', weight=2)
14 A.add_edge('Tel_3','Tel_5', weight=2)
15 A.add_edge('Tel_3','Tel_5', weight=3)
16 A.add_edge('Tel_3','Tel_5', weight=4)
17 A.add_edge('Tel_5','Tel_6', weight=2)
18 A.add_edge('Tel_5','Tel_6', weight=3)
19 A.add_edge('Tel_1','Tel_6', weight=2)
20 A.add_edge('Tel_1','Tel_6', weight=3)
21 black=[('Tel_1','Tel_2'), ('Tel_1','Tel_3'), ('Tel_2','Tel_4'),
22   ('Tel_2','Tel_3'), ('Tel_3','Tel_5'), ('Tel_3','Tel_5'),
23   ('Tel_5','Tel_6'), ('Tel_1','Tel_6')]
24
25 red=[('Tel_1','Tel_2'), ('Tel_2','Tel_4'),
26   ('Tel_1','Tel_2'), ('Tel_1','Tel_6'), ('Tel_5','Tel_6')]
27 bl=[('Tel_1','Tel_2'), ('Tel_3','Tel_5')]
28
29 lista=['Tel_1','Tel_2','Tel_3','Tel_4','Tel_5'], ['Tel_6']
30 #posicion=nx.shell_layout(A, nlist=lista, scale=0.50, center=None,
31   dim=2)
32 #posicion=nx.random_layout(A, center=None, dim=2)
33 #posicion=nx.circular_layout(A, scale=0.5, center=None, dim=2)
34
35 posicion=nx.spectral_layout(A, weight='weight', scale=0.50, center=
36   None, dim=2)
37
38 #posicion=nx.kamada_kawai_layout(A, dist=None, pos=None, weight='
39   weight', scale=0.30, center=None, dim=2)
40
41 #posicion=nx.spring_layout(A, k=1, iterations=200, threshold
42   =0.0001, weight='weight', scale=0.5)
43
44 #posicion=nx.rescale_layout( pos , escala = 1 )
45
46 posicion=nx.bipartite_layout(A, {'Marianao','Vedado','Habana Vieja
47   '}, align='vertical', scale=1, center=None, aspect_ratio
48   =1.333333333333333)
49
50 nx.draw_networkx_nodes(A, posicion,
51   node_size=1000, node_color= 'y')
52 nx.draw_networkx_edges(A, posicion, edgelist=bl, width=10, alpha
53   =0.5,
54   edge_color='b')
55 nx.draw_networkx_edges(A, posicion, edgelist=red, width=5, alpha
56   =0.5,
57   edge_color='r')
58 nx.draw_networkx_edges(A, posicion, edgelist=black, width=2,
59   edge_color='Black')
60
61 nx.draw_networkx_labels(A, posicion, font_size=11,font_family='arial
62   ',
63   )

```

```

56                               font_color='Black', font_weight='bold')
57
58 plot.axis('off')
59 plot.savefig("Graf8_spectral_layout.eps")
60 plot.show(A)

```

Graf8.py

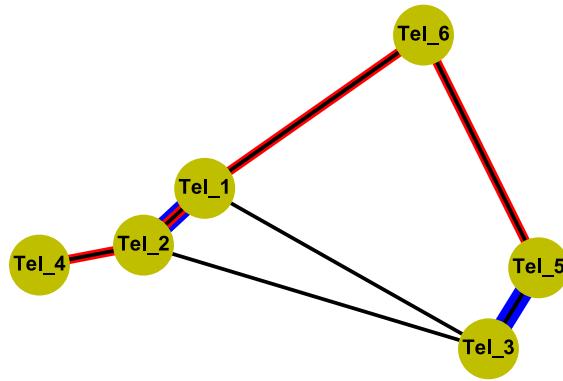


Figura 8: Multigrafo no dirigido cíclico, donde las aristas de diferentes colores representan la existencia de las llamadas con distinta duración

10. Multigrafo no dirigido reflexivo

Partiendo del ejemplo de la sección 3 se pretende construir un grafo más informativo a la hora de su análisis, por lo que se le agrega como arista (el hecho que las relaciones sexuales mantenidas fueran dentro de una relación formal). Este grafo se muestra en la figura 9.

Algoritmo de diseño.

Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 9 el algoritmo *circular layout*, ya que el mismo permite una mejor compresión de este ejemplo mediante la ubicación circular de los nodos.

```

1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4
5 A=nx.MultiDiGraph()
6
7
8 A.add_edge( 'Luis(15)' , 'Luis(15)' , weight=2)
9 A.add_edge( 'Maria(15)' , 'Yanet(17)' , weight=2)
10 A.add_edge( 'Yanet(17)' , 'Yanet(17)' , weight=2)
11
12
13 A.add_edge( 'Maria(15)' , 'Fernando(16)' , weight=2)
14 A.add_edge( 'Fernando(16)' , 'Desisy(18)' , weight=3)
15 A.add_edge( 'Fernando(16)' , 'Desisy(18)' , weight=2)
16 A.add_edge( 'Pedro(17)' , 'Pedro(17)' , weight=2)
17 A.add_edge( 'Pedro(17)' , 'Julia(18)' , weight=2)
18 A.add_edge( 'Pedro(17)' , 'Rosi(16)' , weight=3)
19 A.add_edge( 'Pedro(17)' , 'Rosi(16)' , weight=2)
20 A.add_edge( 'Fernando(16)' , 'Claudia(16)' , weight=2)
21
22 nodes_reflex = { 'Luis(15)' , 'Fernando(16)' , 'Julia(18)' }
23 nodes_no_reflex = { 'Maria(15)' , 'Yanet(17)' , 'Rosi(16)' , 'Desisy(18)' ,
24 'Pedro(17)' , 'Claudia(16)' }
25
26 black=[( 'Maria(15)' , 'Yanet(17)' ),
27 ( 'Maria(15)' , 'Fernando(16)' ),( 'Fernando(16)' , 'Desisy(18)' ),
28 ( 'Yanet(17)' , 'Julia(18)' ),( 'Desisy(18)' , 'Pedro(17)' ),
29 ( 'Pedro(17)' , 'Julia(18)' ), ( 'Pedro(17)' , 'Rosi(16)' ),
30 ( 'Fernando(16)' , 'Claudia(16)' )]
31
32 bl=[( 'Fernando(16)' , 'Desisy(18)' ),( 'Pedro(17)' , 'Rosi(16)' )]
33
34 lista=[ 'Luis(15)' , 'Julia(18)' , 'Yanet(17)' , 'Rosi(16)' , 'Desisy(18)' ,
35 'Claudia(16)' ], [ 'Maria(15)' , 'Fernando(16)' , 'Pedro(17)' ]
36 #posicion=nx.shell_layout(A, nlist=lista , scale=0.50, center=None,
37 dim=2)
38 #posicion=nx.random_layout(A, center=None, dim=2)
39
40 posicion=nx.circular_layout(A, scale=0.5, center=None, dim=2)
41
42 #posicion=nx.spectral_layout(A, weight='distans' , scale=0.50,
43 center=None, dim=2)
44 #posicion=nx.kamada_kawai_layout(A, dist=None, pos=None, weight=
45 'weight' , scale=0.30, center=None, dim=2)
46 #posicion=nx.spring_layout(A, k=1, iterations=200, threshold
47 =0.0001, weight='weight' , scale=0.5)
48 #posicion=nx.rescale_layout( pos , escala = 1 )
49
50 #posicion=nx.bipartite_layout(A, { 'Marianao' , 'Vedado' , 'Habana Vieja'
51 '}' , align='vertical' , scale=1, center=None, aspect_ratio
52 =1.333333333333333)

```

```

51 nx.draw_networkx_nodes(A, posicion , nodelist=nodes_reflex ,
52                               node_size=1800, node_color= '#ea6d6d')
53 nx.draw_networkx_nodes(A, posicion , nodelist=nodes_no_reflex ,
54                               node_size=1500, node_color= '#cde95b')
55 nx.draw_networkx_edges(A, posicion , edgelist=bl , width=5, alpha
56 =0.5,
57 edge_color='b')
58 nx.draw_networkx_edges(A, posicion , edgelist=black , width=2,
59 edge_color='Black')
60 nx.draw_networkx_labels(A, posicion , font_size=11,font_family='arial
61 ',
62 font_color='#4f5148' , font_weight='bold')
63 plot.axis('off')
64 plot.savefig("Graf9_circular_layout.eps")
65 plot.show(A)

```

Graf9.py

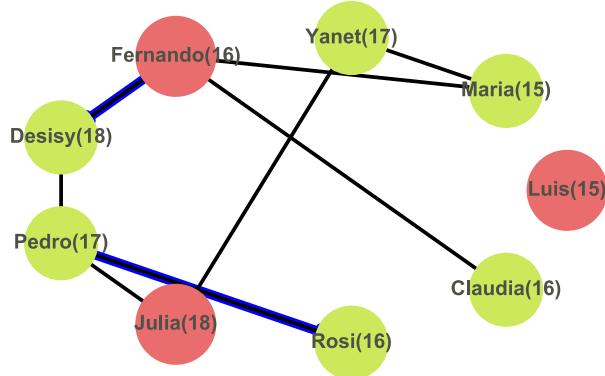


Figura 9: Multigrafo no dirigido reflexivo, donde las aristas de diferentes colores representan la existencia de una relación formal entre los individuos y los vértices de color rojo representan los vértices con aristas reflexivas.

11. Multigrafo dirigido acíclico

Considerando que se quiere trazar las diferentes rutas (teniendo en cuenta el sentido de los mismos) que llevan de una provincia a otra en Cuba (Pinar del

Río, Artemisa, La Habana, Mayabeque, Matanza, Cienfuegos, Villa Clara). Tomando como vértices las provincias y como aristas las carreteras que las unen, como muestra la figura 10.

Algoritmo de diseño.

Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 10 el algoritmo *spectral layout*, ya que el mismo logró un buen resultado en la ubicación de los nodos facilitando la compresión del grafo.

```

1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4
5 A=nx.MultiDiGraph()
6 A.add_edge( 'PR' , 'A' , weight=2)
7 A.add_edge( 'A' , 'LH' , weight=2)
8 A.add_edge( 'A' , 'LH' , weight=4)
9 A.add_edge( 'LH' , 'May' , weight=2)
10 A.add_edge( 'May' , 'Mat' , weight=2)
11 A.add_edge( 'May' , 'Mat' , weight=4)
12 A.add_edge( 'LH' , 'May' , weight=4)
13 A.add_edge( 'LH' , 'May' , weight=5)
14 A.add_edge( 'May' , 'Mat' , weight=5)
15 A.add_edge( 'Mat' , 'C' , weight=2)
16 A.add_edge( 'Mat' , 'C' , weight=4)
17
18 black=[( 'PR' , 'A' ) ,( 'LH' , 'May' ) ,( 'May' , 'Mat' ) ,
19 ( 'Mat' , 'C' ) ,( 'A' , 'LH' ) ]
20 red=[( 'A' , 'LH' ) ,( 'May' , 'Mat' ) ,
21 ( 'Mat' , 'C' ) ,( 'LH' , 'May' ) ]
22 bl=[( 'May' , 'Mat' ) ,( 'LH' , 'May' ) ]
23 lista=[ 'PR' , 'A' , 'Mat' , 'May' , 'LH' ] ,[ 'C' ]
24 #posicion=nx.shell_layout(A, nlist=lista , scale=0.50, center=None,
25 dim=2)
26
27 #posicion=nx.random_layout(A, center=None, dim=2)
28
29 #posicion=nx.circular_layout(A, scale=0.5, center=None, dim=2)
30
31 posicion=nx.spectral_layout(A, weight='distans' , scale=0.50, center
32 =None, dim=2)
33
34 #posicion=nx.kamada_kawai_layout(A, dist=None, pos=None, weight=
35 weight , scale=0.30, center=None, dim=2)
36
37 #posicion=nx.spring_layout(A, k=1, iterations=200, threshold
38 =0.0001, weight='weight' , scale=0.5)
39
40 #posicion=nx.rescale_layout( pos , escala = 1 )
41
42 #posicion=nx.bipartite_layout(A, { 'A' , 'May' , 'C' } , align='vertical'
43 scale=0.5, center=None, aspect_ratio=1.333333333333333)
44
45 nx.draw_networkx_nodes(A, posicion ,

```

```

41             node_size=500, node_color= 'y')
42 nx.draw_networkx_edges(A, posicion , edgelist=bl , width=10, alpha
43 =0.5,
44 edge_color='b')
45 nx.draw_networkx_edges(A, posicion , edgelist=red , width=5, alpha
46 =0.5,
47 edge_color='r')
48 nx.draw_networkx_edges(A, posicion , edgelist=black , width=2,
49 edge_color='Black')
50 nx.draw_networkx_labels(A, posicion , font_size=11,font_family='arial
51 ',
52 font_color='Black' , font_weight='bold')
53
54 plot.axis('off')
plot.savefig("Graf10_spectral_layout.eps")
plot.show(A)

```

Graf10.py

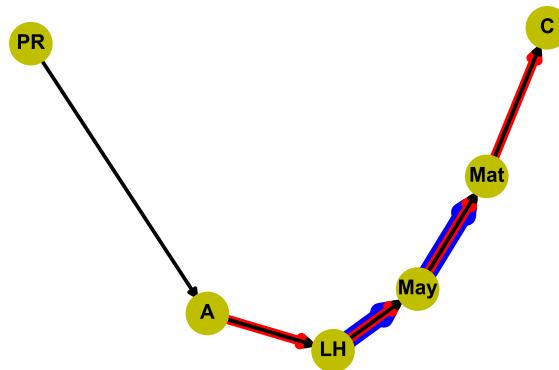


Figura 10: Multigrafo dirigido acíclico, donde las aristas de diferentes colores representan la existencia de más de una ruta entre las diferentes provincias.

12. Multigrafo dirigido cíclico

“En el caso particular de que las redes reflejen una realidad social, los nodos pueden representar personas o entidades relacionadas con sus contextos, y las conexiones representarán relaciones sociales existentes entre ellos (amistad, parentesco, membresía, afinidad, etc.). A pesar de que intuitivamente las redes sociales se asemejan a los grafos matemáticos, es más habitual que en ellas se trabaje con distintos tipos de relaciones”[5] por lo que es necesario la utilización de multígrafos, que es la herramienta que contempla más de una relación entre dos nodos, con esto ganamos mayor riqueza en los datos a analizar. Por ejemplo, tenemos un grupo de personas que laboran en un departamento de Informática y se hace una encuesta donde se les pide que marque cuales de tres sentimientos (respeto, afinidad, rechazo) sienten por sus compañeros de trabajo, como muestra la figura 11.

Algoritmo de diseño.

Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 11 el algoritmo *kamada kawai layout*, ya que el mismo logró un buen resultado estético.

```
1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4
5 A=nx.MultiDiGraph()
6 A.add_edge('Personal1','Persona2', weight=2)
7 A.add_edge('Personal1','Persona2', weight=3)
8 A.add_edge('Personal1','Persona2', weight=4)
9 A.add_edge('Personal1','Persona3', weight=2)
10 A.add_edge('Personal1','Persona4', weight=4)
11 A.add_edge('Persona2','Persona4', weight=2)
12 A.add_edge('Persona2','Persona4', weight=3)
13 A.add_edge('Persona4','Persona3', weight=3)
14 A.add_edge('Persona3','Persona2', weight=2)
15 A.add_edge('Persona3','Persona5', weight=2)
16 A.add_edge('Persona3','Persona5', weight=3)
17 A.add_edge('Persona3','Persona5', weight=4)
18 A.add_edge('Persona5','Persona6', weight=2)
19 A.add_edge('Persona5','Persona6', weight=3)
20 A.add_edge('Personal1','Persona6', weight=2)
21 A.add_edge('Personal1','Persona6', weight=3)
22 black=[('Personal1','Persona2'), ('Personal1','Persona3'), ('Persona2',
   'Persona4'),
   ('Persona3','Persona2'), ('Persona3','Persona5'), ('Persona3',
   'Persona5'),
   ('Persona5','Persona6'), ('Personal1','Persona6'), ('Persona4',
   'Persona3')]
23
24
25
26 red=[('Personal1','Persona2'), ('Persona2','Persona4'),
   ('Personal1','Persona2'), ('Personal1','Persona6'), ('Persona5',
   'Persona6')]
27
28 bl=[('Personal1','Persona2'), ('Persona3','Persona5')]
```

```

29 lista=['Persona1','Persona2','Persona3','Persona4','Persona5'],[''
30     'Persona6']
31 #posicion=nx.shell_layout(A, nlist=lista, scale=0.50, center=None,
32     dim=2)
33 #posicion=nx.random_layout(A, center=None, dim=2)
34 #posicion=nx.circular_layout(A, scale=0.5, center=None, dim=2)
35 #posicion=nx.spectral_layout(A, weight='distans', scale=0.45,
36     center=None, dim=2)
37 posicion=nx.kamada_kawai_layout(A, dist=None, pos=None, weight='
38     weight', scale=0.30, center=None, dim=2)
39 #posicion=nx.spring_layout(A, k=1, iterations=200, threshold
40     =0.0001, weight='weight', scale=0.5)
41 #posicion=nx.rescale_layout( pos , escala = 1 )
42 #posicion=nx.bipartite_layout(A, {'Marianao','Vedado','Habana Vieja
43     '}, align='vertical', scale=1, center=None, aspect_ratio
44     =1.333333333333333)
45
46 nx.draw_networkx_nodes(A, posicion ,
47                         node_size=500
48                         , node_color= 'y')
49 nx.draw_networkx_edges(A, posicion , edgelist=bl , width=10, alpha
50                         =0.5,
51                         edge_color='b')
52 nx.draw_networkx_edges(A, posicion , edgelist=red , width=5, alpha
53                         =0.5,
54                         edge_color='r')
55 nx.draw_networkx_edges(A, posicion , edgelist=black , width=2,
56                         edge_color='#297033')
57 for p in posicion:
58     posicion[p][1]+= 0.07
59 nx.draw_networkx_labels(A, posicion , font_size=11,font_family='arial
60     ',
61                         font_color='Black', font_weight='bold')
62
63 plot.axis('off')
64 plot.savefig("Graf11_kamada_kawai_layout.eps")
65 plot.show(A)

```

Graf11.py

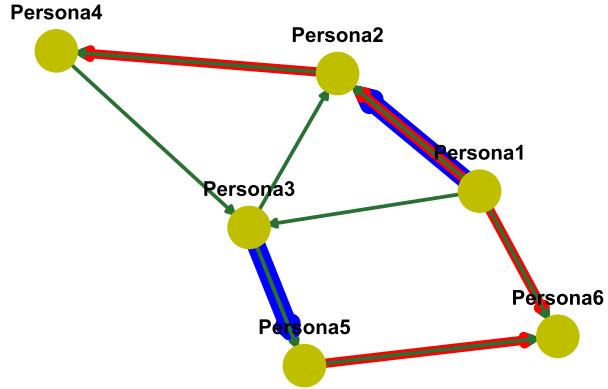


Figura 11: Multigrafo dirigido cíclico, donde las aristas de diferentes colores representan la existencia de más de una opinión sobre sus compañeros.

13. Multigrafo dirigido reflexivo

Una de las aplicaciones de estos grafos es en el modelado del funcionamiento de una máquina de estados. Por ejemplo, se desea modelar un grafo que represente el funcionamiento de una máquina muy sencilla de golosinas, que responde a las siguientes reglas:

- 1 Cada golosina vale \$ 0.25.
- 2 La máquina acepta sólo monedas de \$ 0.10 y de \$ 0.05.
- 3 La máquina NO da vuelto. *Siempre* *sí*

Para modelar el grafo, debemos considerar distintos “estados de dinero ingresado”, y como se va pasando de uno a otro. Por ejemplo, si en un momento tenemos ingresados 5 centavos, y agregamos 5 centavos más, pasamos a otro estado, que es el mismo que si hubiésemos ingresado 10 centavos al principio. Llamemos A, B, C, D, E y F a los estados que representan 0, 5, 10, 15, 20, 25 centavos ingresados respectivamente. Las transiciones de un estado a otro se harán por ingreso de 5 o 10 centavos, o al presionar el botón para obtener los caramelos (G). Como se muestra en la figura 12.

Algoritmo de diseño.

Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 12 el algoritmo *spectral*

layout, ya que este algoritmo mostro el mejor de los resultados a la hora de representar este ejemplo.

```

1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4
5 A=nx.MultiDiGraph()
6 A.add_edge('A','A', weight=2)
7 A.add_edge('A','B', weight=2)
8 A.add_edge('A','C', weight=2)
9 A.add_edge('B','B', weight=2)
10 A.add_edge('B','C', weight=2)
11 A.add_edge('B','D', weight=2)
12 A.add_edge('C','C', weight=2)
13 A.add_edge('C','D', weight=2)
14 A.add_edge('C','E', weight=2)
15 A.add_edge('D','D', weight=2)
16 A.add_edge('D','E', weight=2)
17 A.add_edge('D','F', weight=2)
18 A.add_edge('E','E', weight=2)
19 A.add_edge('E','F', weight=2)
20 A.add_edge('E','F', weight=4)
21 A.add_edge('F','F', weight=2)
22 A.add_edge('F','A', weight=2)
23
24
25 black=[( 'A' , 'A' ) ,( 'A' , 'B' ) ,( 'A' , 'C' ) ,
26 ( 'B' , 'B' ) ,( 'B' , 'C' ) ,( 'B' , 'D' ) ,( 'C' , 'C' ) ,( 'C' , 'D' ) ,( 'C' , 'E' ) ,
27 ( 'D' , 'D' ) ,( 'D' , 'E' ) ,( 'D' , 'F' ) ,( 'E' , 'E' ) ,( 'E' , 'F' ) ,( 'E' , 'F' ) ,( 'F' ,
28 , 'A' )]
29 bl=[( 'E' , 'F' )]
30 lista=[ 'A' , 'B' , 'C' , 'D' , 'E' ] ,[ 'F' ]
31 #posicion=nx.shell_layout(A, nlist=lista , scale=0.50, center=None,
32 dim=2)
33 #posicion=nx.random_layout(A, center=None, dim=2)
34 #posicion=nx.circular_layout(A, scale=0.5, center=None, dim=2)
35 posicion=nx.spectral_layout(A, weight='distans', scale=0.50, center
36 =None, dim=2)
37 #posicion=nx.kamada_kawai_layout(A, dist=None, pos=None, weight=
38 weight', scale=0.30, center=None, dim=2)
39 #posicion=nx.spring_layout(A, k=1, iterations=200, threshold
40 =0.0001, weight='weight', scale=0.5)
41 #posicion=nx.rescale_layout( pos , escala = 1 )
42 #posicion=nx.bipartite_layout(A, { 'Marianao ' , 'Vedado ' , 'Habana Vieja
43 ' } , align='vertical', scale=1, center=None, aspect_ratio
44 =1.333333333333333)
45 nx.draw_networkx_nodes(A, posicion ,
46
47

```

```

48          node_size=600, node_color= '#48b457')
49 nx.draw_networkx_edges(A, posicion , edgelist=bl , width=10, alpha
50      =0.5,
51 edge_color='b' ,arrowsize=20)
52 nx.draw_networkx_edges(A, posicion , edgelist=black , width=2,
53 edge_color='Black' ,arrowsize=20)
54 nx.draw_networkx_labels(A, posicion , font_size=14,font_family='arial
55      ,
56      font_color='Black' , font_weight='bold')
57
58 plot.axis('off')
59 plot.savefig("Graf12_spectral_layout.eps")
60 plot.show(A)

```

Graf12.py

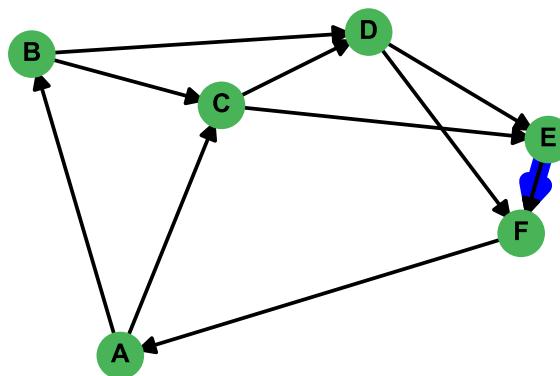


Figura 12: Multigrafo dirigido reflexivo, donde las aristas de diferentes colores representan la existencia de más de una forma de pasar de un estado a otro.

Referencias

- [1] Andrés Aiello and Rodrigo Ignacio Silveira. Trazado de grafos mediante métodos dirigidos por fuerzas: revisión del estado del arte y presentación de algoritmos para grafos donde los vértices son regiones geográficas. Technical report, Universidad de Buenos Aires, <https://dccg.upc.edu/people/rodrigo/pubs/MScThesis.pdf>, Diciembre 2004.
- [2] Amalia Duch Brown. Grafos. <https://www.cs.upc.edu/~duch/home/duch/grafos.pdf>, Octubre 2007.
- [3] Chakraborty, Anwesha, Trina Dutta, Mondal, Sushmita, Nath, and Asoke. Application of graph theory in social media. *International Journal of Computer Sciences and Engineering*, 6:722–729, 10 2018.
- [4] NetworkX Developers Copyright 2004-2019. Overview of networkx. <https://networkx.github.io/documentation/latest/index.html>, Febrero 2019.
- [5] R. A. Hanneman and M. Riddle. Introduction to social network methods. <http://faculty.ucr.edu/~hanneman/>. Consultado, Septiembre 2013.
- [6] Kimball Martin. *Graph Theory and Social Networks*. Spring, Abril 2014.
- [7] Mauricio Beltrán Pascuala, Azahara Muñoz Martínez, and Ángel Muñoz Alamillos. Redes bayesianas aplicadas a problemas de credit scoring. Una aplicación práctica. <http://www.elsevier.es/en-revista-cuadernos-economia-329-articulo-redes-bayesianas-aplicadas-problemas-credit>, Octubre 2013.

Tarea 2

5271

26 de mayo de 2019

1. Introducción

Cuando una información se manifiesta mediante objetos y sus relaciones, visualizarla puede ser el primer paso para resolver problemas de diversa índole. La modelación y teoría de grafos constituyen la forma más común de modelar información relacional, y su visualización una forma de representar la información [1].

Sin embargo el modelado y acomodo de un grafo que refleje situaciones de la vida real resulta complejo, es por ello que se han desarrollado algoritmos dedicados a perfeccionar el trazado de los mismos según el tipo problema y grafo que representen, llamados algoritmos de diseño o *layout*.

Los algoritmos de diseño son los que devuelven una lista de posiciones para los nodos según diversos parámetros definidos para cada algoritmo, buscando cumplir con ciertos criterios estéticos como son: minimizar cruces entre aristas, maximizar el ángulo entre aristas adyacentes, maximizar el ángulo entre aristas que se cruzan, mostrar simetría, distribución uniforme de los vértices y longitud uniforme de aristas [1].

Entre los algoritmos de diseño destacan los siguientes:

- *Bipartite layout* (posiciona los nodos en dos líneas rectas, es decir que divide el conjunto de nodos en dos subconjuntos X e Y donde no existe adyacencia entre los elementos de un mismo subconjunto) [4].
- *Circular layout* (ubica los nodos en forma circular) [4].
- *Kamada kawai layout* (posiciona los nodos utilizando la función de costo de longitud de camino Kamada-Kawai) [4].
- *Random layout* (posiciona los nodos de forma aleatoria) [4].

- *Rescale layout* (reescala dado una matriz de posiciones) [4].
- *Shell layout* (posiciona los nodos en círculos concéntricos) [4].
- *Spring layout* (posiciona los nodos utilizando el algoritmo dirigido por fuerza de Fruchterman-Reingold) [4].
- *Spectral layout* (posiciona nodos utilizando los vectores propios del gráfico laplaciano) [4].

2. Grafo simple no dirigido acíclico

Se pueden encontrar varias aplicaciones a la modelación de un grafo simple no dirigido acíclico, una de las más claras son los árboles, “el árbol (árbol libre) que es un grafo no dirigido, conexo y acíclico. Un árbol también puede definirse como un grafo no dirigido en el que hay exactamente un camino entre todo par de vértices”[2].

Un ejemplo de usos de árboles es en topología de red la de árbol, en esta topología los nodos de la red están ubicados en forma de árbol. Esta conexión es similar a muchas redes en estrella interconectadas con la diferencia de no poseer un nodo central, en cambio posee un nodo troncal desde el cual se ramifican el resto de los nodos como se muestra en la figura 1 que es una pequeña representación con solo ocho nodos de la red que posee la UEB Rolando Pérez Gollanes entidad dedicada al sacrificio y procesamiento de aves.

Algoritmo de diseño. Despues de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 1 el algoritmo *spring layout*, dado que cumple con la mayor cantidad de los criterios estéticos para este ejemplo.

```

1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4 A=nx.Graph()
5 A.add_edges_from([( 'R1' , 'Sw1') , ( 'Sw1' , 'Sw2') ,
6                   ( 'Sw1' , 'Sw3') , ( 'Sw2' , 'Pc2') , ( 'Sw2' , 'Pc3') ,
7                   ( 'Sw3' , 'Pc4') , ( 'Sw3' , 'Pc5')])
8
9 sw=[ 'R1' , 'Sw1' , 'Sw2' , 'Sw3']
10 pc=[ 'Pc2' , 'Pc3' , 'Pc4' , 'Pc5']
11 swc=[( 'R1' , 'Sw1') , ( 'Sw1' , 'Sw2') , ( 'Sw1' , 'Sw3')]
12 pcc=[( 'Sw2' , 'Pc2') , ( 'Sw2' , 'Pc3') , ( 'Sw3' , 'Pc4') , ( 'Sw3' , 'Pc5')]
13 lista=[ 'Sw2' , 'Pc2' , 'Pc3' , 'Sw3' , 'Pc4' , 'Pc5' , 'R1' , [ 'Sw1' ]
14
15 posicion=nx.spring_layout(A, k=1, iterations=300, threshold=0.0001,
16                           weight='distans', scale=0.5)

```

```

17 nx.draw_networkx_nodes(A, posicion , node_size=800, node_shape='s' ,
18     nodelist=sw, node_color='#c9b323')
19 nx.draw_networkx_nodes(A, posicion , node_size=800, node_shape='o' ,
20     nodelist=pc, node_color='#c68282')
21 nx.draw_networkx_edges(A, posicion , width=4, edgelist=swc, style='
22     dashed',edge_vmax=1, edge_vmin=1)
23 nx.draw_networkx_edges(A, posicion , width=2, edgelist=pcc)
24 nx.draw_networkx_labels(A, posicion , font_size=11,font_family='arial
25     ')
26
27 plot.axis('off')
28 plot.savefig("Grafo1_spring_layout.eps")
29 plot.show(A)

```

Grafo1.py

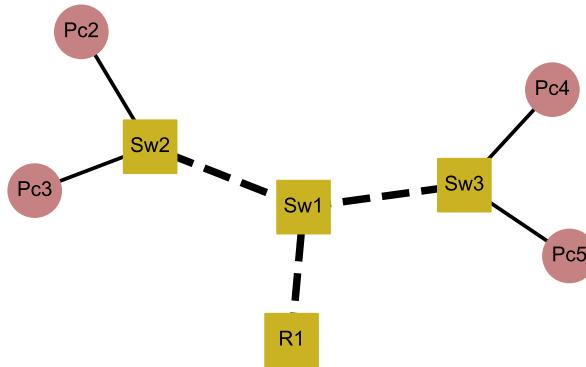


Figura 1: Grafo simple no dirigido acíclico, donde las líneas discontinuas representan los cables de fibra óptica y los líneas continuas los cables UTP.

3. Grafo simple no dirigido cíclico

Un grafo simple no dirigido cíclico puede usarse áreas como geografía. Si se considera un mapa, digamos de Europa: que cada país sea un vértice y conecte dos vértices con una arista si esos países comparten una frontera. Un problema famoso que quedó sin resolver durante más de cien años fue el problema de los cuatro colores. Aproximadamente esto indica que cualquier mapa puede ser coloreado con a lo sumo cuatro colores de tal manera que los países adyacentes no tengan el mismo color. Este problema motivó muchos desarrollos en la teoría de grafos y finalmente se demostró con la ayuda de una computadora en 1976 [6].

Otra aplicación es en la representación de redes sociales, una red social se conceptualiza como un grafo, es decir, un conjunto de vértices (o nodos, unidades, puntos) que representan entidades u objetos sociales y un conjunto de líneas que representan una o más relaciones sociales entre ellos [3]. Por ejemplo, si consideramos un grupo de nueve doctores del núcleo académico de PISIS (Posgrado en Ingeniería de Sistemas) y construimos una red, tomando como vértices a los doctores y la colaboración de ellos en artículos publicados como aristas da lugar a un grafo simple no dirigido cíclico como se muestra en la figura 2.

Algoritmo de diseño. Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 2 el algoritmo *circular layout*, ya que el mismo permite una mejor compresión del grafo del ejemplo en cuestión.

```
1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4 A=nx.Graph()
5 A.add_edges_from([( 'Dr.FL' , 'Dra.AA') , ( 'Dr.FL' , 'Dra.YR') ,
6                  ( 'Dr.FL' , 'Dr.RS') , ( 'Dr.FL' , 'Dra.ES') ,
7                  ( 'Dra.YR' , 'Dr.VB') , ( 'Dra.YR' , 'Dr.RR') ,
8                  ( 'Dra.YR' , 'Dr.RS') , ( 'Dra.AA' , 'Dra.IM') ,
9                  ( 'Dra.AA' , 'Dr.RR') , ( 'Dra.IM' , 'Dr.VB') ,
10                 ( 'Dra.IM' , 'Dra.AS') , ( 'Dra.IM' , 'Dr.RS') ,
11                 ( 'Dra.IM' , 'Dr.VB') , ( 'Dr.VB' , 'Dr.RS') ,
12                 ( 'Dra.AS' , 'Dr.VB') , ( 'Dra.AS' , 'Dr.RR') ,
13                 ( 'Dra.AS' , 'Dr.RS') , ( 'Dra.ES' , 'Dr.RR') ,
14                 ( 'Dra.AA' , 'Dra.ES') , ( 'Dra.IM' , 'Dra.ES') ])
15 lista=[ 'Dr.FL' , 'Dra.AA' , 'Dra.YR' , 'Dr.RR' , 'Dra.IM' , 'Dra.AS' , 'Dr.VB' ,
16        'Dra.ES' ] ,[ 'Dr.RS' ]
17 posicion=nx.circular_layout(A, scale=0.5, center=None, dim=2)
18 nx.draw_networkx_nodes(A, posicion , node_size=2000, node_color=range
19                        (9),cmap=plot.cm.Blues)
20 nx.draw_networkx_edges(A, posicion , width=2,edge_color=' #46267d ')
21 nx.draw_networkx_labels(A, posicion , font_size=13,font_family=' arial
, ,
```

```

22         font_color='#846c16', font_weight='bold')
23 plot.axis('off')
24 plot.savefig("Graf2_circular_layout.eps")
25 plot.show(A)

```

Grafo2.py

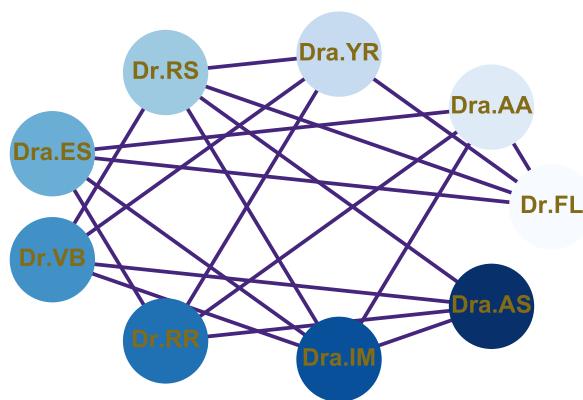


Figura 2: Grafo simple no dirigido cíclico, donde los vértices representan los diferentes doctores y las aristas la colaboración entre ellos.

4. Grafo simple no dirigido reflexivo

Una de las aplicaciones de este tipo de grafo es en la modelación de comportamientos de elementos sociales en la vida real.

Por ejemplo, en un estudio sobre el comportamiento sexual de un grupo de adolescentes con edades comprendida entre 15 y 18 años, se representarán las relaciones sexuales consentidas (aristas) que existen entre los individuos (vértices) del grupo, así como la satisfacción, para así saber tendencias por edades, posibles esquemas de propagación de enfermedades y promiscuidad entre otros factores del interés de los sexólogos. Este ejemplo se muestra en el grafo de la figura 3.

Algoritmo de diseño. Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 3 el algoritmo *kamada kawai layout*, dado que este algoritmo es uno de los más usados para la representación de grafos conexos y no dirigidos, por los resultados de acomodo tan buenos que arroja en estos tipos de grafos.

```
1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4 A=nx.Graph()
5 A.add_edge('Luis(15)', 'Maria(15)')
6 A.add_edge('Luis(15)', 'Luis(15)')
7 A.add_edge('Maria(15)', 'Yanet(17)')
8 A.add_edge('Yanet(17)', 'Yanet(17)')
9 A.add_edge('Maria(15)', 'Ferndo(16)')
10 A.add_edge('Ferndo(16)', 'Desisy(18)')
11 A.add_edge('Desisy(18)', 'Pedro(17)')
12 A.add_edge('Pedro(17)', 'Pedro(17)')
13 A.add_edge('Pedro(17)', 'Julia(18)')
14 A.add_edge('Pedro(17)', 'Rosi(16)')
15 A.add_edge('Ferndo(16)', 'Claudia(16)')
16 nodes_reflex = {'Luis(15)', 'Ferndo(16)', 'Julia(18)'}
17 nodes_no_reflex = {'Maria(15)', 'Yanet(17)', 'Rosi(16)', 'Desisy(18)', 'Pedro(17)', 'Claudia(16)'}
18
19 lista =['Luis(15)', 'Julia(18)', 'Yanet(17)', 'Rosi(16)', 'Desisy(18)', 'Claudia(16)'], ['Maria(15)', 'Ferndo(16)', 'Pedro(17)']
20
21
22 posicion=nx.kamada_kawai_layout(A, dist=None, pos=None, weight='weight', scale=0.5, center=None, dim=2)
23
24 nx.draw_networkx_nodes(A, posicion, nodelist=nodes_reflex, node_size=1500, node_color= '#ea6767')
25 nx.draw_networkx_nodes(A, posicion, nodelist=nodes_no_reflex, node_size=1500, node_color= 'y')
26 nx.draw_networkx_edges(A, posicion, width=4, edge_color='#85d9ce')
27
28 nx.draw_networkx_labels(A, posicion, font_size=11, font_family='arial',
```

```

32                               font_color='Black', font_weight='bold')
33
34 plot.axis('off')
35 plot.savefig("Graf3_kamada_kawai_layout.eps")
36 plot.show(A)

```

Grafo3.py

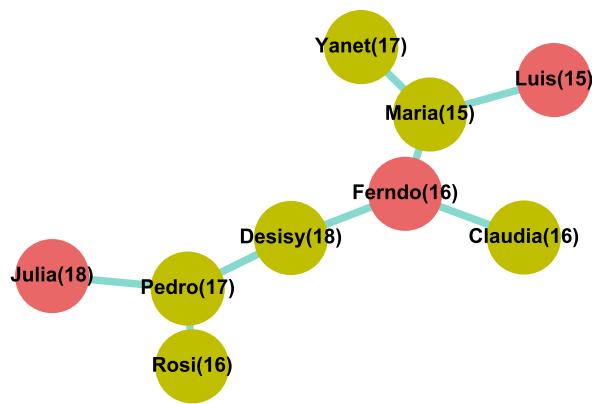


Figura 3: Grafo simple no dirigido reflexivo, donde los vértices rojos representan los nodos donde están las aristas reflexivas.

5. Grafo simple dirigido acíclico

Encontraos entre las aplicaciones de los grafos dirigidos o dígrafos acíclico las siguientes:

- Una red bayesiana queda especificada formalmente por una dupla $B = (G, O)$, donde G es un grafo dirigido acíclico (GDA) y O es el conjunto de distribuciones de probabilidad. Definimos un grafo como un par $G = (V, E)$, donde V es un conjunto finito de vértices nodos o variables y E es un subconjunto del producto cartesiano $V \times V$ de pares ordenados de nodos que llamamos enlaces o aristas [7].
- Los arboles dirigidos son un ejemplo clásico de grafos dirigidos acíclico, estos tienen múltiples en la cotidianidad como pueden ser los arboles genealógicos, los organigramas de una empresa (referido a las jerarquías entre los empleados) y el árbol de directorios de Windows.

En la figura 4 se muestra un ejemplo de red bayesiana (topología de red para el cáncer de pulmón), donde C (cáncer), Con (Contaminación), F (Fumador), D (disnea), Rx (rayos-x) son los vértices y sus relaciones son las aristas.

Algoritmo de diseño. Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 4 el algoritmo *shell layout*, dado que este algoritmo ubica los nodos en círculos concéntricos a un nodo dado, es la mejor opción para representar este ejemplo.

```
1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4 A=nx.DiGraph()
5 A.add_edge('Con', 'C', weight=0.8)
6 A.add_edge('F', 'C', weight=0.8)
7 A.add_edge('C', 'D', weight=3)
8 A.add_edge('C', 'Rx', weight=3)
9 lista=['Con', 'F', 'D', 'Rx'], [ 'C' ]
10 nodes=['Con', 'F', 'D', 'Rx']
11
12 posicion=nx.shell_layout(A, nlist=lista , scale=0.8, center=None,
13   dim=2)
14
15 nx.draw_networkx_nodes(A, posicion , node_size=700, nodelist=[ 'C' ],
16   node_color= '#bbf1f0' , alpha=1)
17 nx.draw_networkx_nodes(A, posicion , nodelist= [ 'Con', 'F', 'D', 'Rx' ],
18   node_size=600, node_color= '#6ed3c5' , alpha=1)
19 nx.draw_networkx_edges(A, posicion , width=4,edge_color='#35665f')
```

```
20  
21 plot.axis('off')  
22 plot.savefig("Graf4_shell_layout.eps")  
23 plot.show(A)
```

Grafo4.py

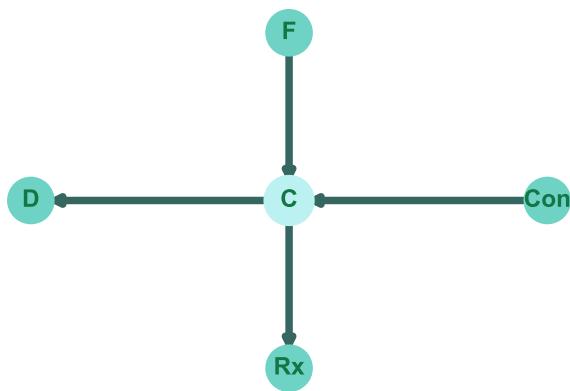


Figura 4: Grafo simple dirigido acíclico.

6. Grafo simple dirigido cíclico

Aplicaciones de los grafos dirigidos cílicos:

- En ingeniería eléctrica se utilizan grafos dirigidos cílicos en el análisis de circuito desde Kirchhoff en los años 1850.
- En redes sociales, otro enfoque de redes sociales en su análisis puede arrojar un grafo dirigido cíclico si te tomamos como vértices a personas y como aristas el sentimiento de amistad de una persona hacia otra, este ejemplo se refleja en el grafo de la figura 5.

Algoritmo de diseño. Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 5 el algoritmo *kamada kawai layout*, a pesar de no ser este ejemplo precisamente el más adecuado para este algoritmo fue el de mejor resultado estético.

```
1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4
5 A=nx.DiGraph()
6 A.add_edge('Ana','Felix')
7 A.add_edge('Ana','Alberto')
8 A.add_edge('Alberto','Yanet')
9 A.add_edge('Alberto','Fernando')
10 A.add_edge('Yanet','Roger')
11 A.add_edge('Teresa','Roger')
12 A.add_edge('Felix','Marta')
13 A.add_edge('Marta','Fernando')
14 A.add_edge('Yanet','Ana')
15 A.add_edge('Marta','Ana')
16 lista=['Ana','Felix','Alberto','Yanet','Fernando'],['Roger','Teresa',
   'Marta']
17
18 posicion=nx.kamada_kawai_layout(A, dist=None, pos=None, weight='weight',
   scale=0.5, center=None, dim=2)
19
20 nx.draw_networkx_nodes(A, posicion,
   node_size=500, node_color='y')
21 nx.draw_networkx_edges(A, posicion, width=2, edge_color='b')
22 for p in posicion:
23     posicion[p][1] += 0.05
24 nx.draw_networkx_labels(A, posicion, font_size=11, font_family='arial',
   font_color='Black', font_weight='bold')
25
26 plot.axis('off')
27 plot.savefig("Graf5_kamada_kawai_layout.eps")
28 plot.show(A)
```

Grafo5.py

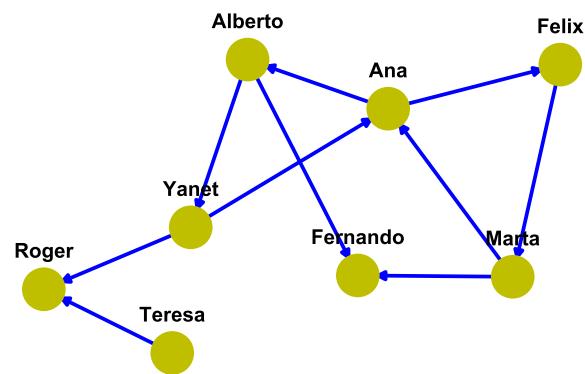


Figura 5: Grafo simple dirigido cíclico, donde los vértices representan las personas y las aristas la relación de amistad entre ellas.

7. Grafo simple dirigido reflexivo

Las aplicaciones de estos grafos van desde la representación del funcionamiento de las páginas web, el modelado de una empresa de servicio que brinda el mismo tanto a sus clientes como a ella misma, hasta representar un grafo que modele la comprobación de una red de computadoras.

Por ejemplo, suponiendo que se está comprobando la conectividad entre los nodos (equipos) de una red de computadoras, es decir que un nodo puede dar ping a cualquier otro nodo, significa que dicho nodo está conectado con el resto, además se debe asegurar que el mismo nodo pueda recibir un auto ping, si esto no ocurre existe un problema de conectividad. La figura 6 muestra dicho ejemplo.

Algoritmo de diseño. Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 6 el algoritmo *random layout*, en este ejemplo se pudo usar este algoritmo por su simpleza ya que con grafos más complejos no es aconsejable el uso del mismo por sus malos resultados estéticamente hablando.

```
 1 import networkx as nx
 2 import matplotlib.pyplot as plot
 3
 4 A=nx.DiGraph()
 5 A.add_edge('Pc1','Pc1')
 6 A.add_edge('Pc1','Pc2')
 7 A.add_edge('Pc1','Pc3')
 8 A.add_edge('Pc1','Pc4')
 9 A.add_edge('Pc1','Pc5')
10 A.add_edge('Pc1','Pc6')
11 A.add_edge('Pc1','Pc7')
12 nodes_reflex = {'Pc1'}
13 nodes_no_reflex = {'Pc2', 'Pc3', 'Pc4', 'Pc5', 'Pc6', 'Pc6', 'Pc7'}
14 lista=['Pc2', 'Pc3', 'Pc4', 'Pc5', 'Pc6', 'Pc6', 'Pc7'], [ 'Pc1']
15
16 posicion=nx.random_layout(A, center=None, dim=2)
17
18 nx.draw_networkx_nodes(A, posicion , nodelist=nodes_reflex ,
19                         node_size=500, node_color= 'r')
20 nx.draw_networkx_nodes(A, posicion , nodelist=nodes_no_reflex ,
21                         node_size=500, node_color= 'y')
22 nx.draw_networkx_edges(A, posicion , width=2)
23 nx.draw_networkx_labels(A, posicion , font_size=11,font_family='arial
   ')
24
25 plot.axis( 'off')
26 plot.savefig("Graf6_random_layout.eps")
27 plot.show(A)
```

Grafo6.py

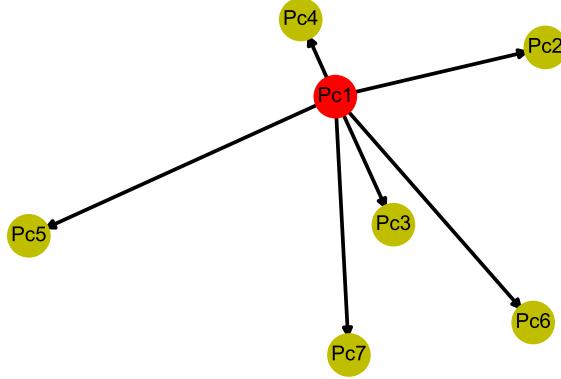


Figura 6: Grafo simple dirigido reflexivo, donde el vértice rojo representa la arista reflexiva.

8. Multigrafo no dirigido acíclico

Una de las aplicaciones de este tipo de grafo es en el tasado de rutas. Por ejemplo, considerando que se quiere trazar los diferentes caminos (sin importar el sentido de estos) que comunican a varios Municipios de La Habana en un orden específico (Lisa, Marianao, Playa, Vedado, Habana Vieja, Habana del Este). Tomando como vértices los municipios y como aristas las carreteras que los unen, esto da lugar al grafo que muestra la figura 7.

Algoritmo de diseño. Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 7 el algoritmo *spring layout*, este acomoda los nodos con el algoritmo dirigido por fuerza de Fruchterman-Reingold que minimiza la distancia entre los vértices conectados y de mayor peso.

```

1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4
5 A=nx.MultiGraph()
6 A.add_edge('Lisa','Marianao', weight=2)
7 A.add_edge('Marianao','Playa', weight=2)
8 A.add_edge('Marianao','Playa', weight=4)
9 A.add_edge('Playa','Vedado', weight=2)
10 A.add_edge('Vedado','Habana Vieja', weight=2)

```

```

11 A.add_edge( 'Vedado' , 'Habana Vieja' , weight=4)
12 A.add_edge( 'Vedado' , 'Habana Vieja' , weight=5)
13 A.add_edge( 'Habana Vieja' , 'Habana del Este' , weight=2)
14 A.add_edge( 'Habana Vieja' , 'Habana del Este' , weight=4)
15
16 black=[( 'Lisa' , 'Marianao' ),( 'Playa' , 'Vedado' ),( 'Vedado' , 'Habana
17 Vieja' ),
18 ( 'Habana Vieja' , 'Habana del Este' ),( 'Marianao' , 'Playa' )]
19 red=[( 'Marianao' , 'Playa' ),( 'Vedado' , 'Habana Vieja' ),
20 ( 'Habana Vieja' , 'Habana del Este' )]
21 bl=[( 'Vedado' , 'Habana Vieja' )]
22
23 lista=[ 'Lisa' , 'Marianao' , 'Playa' , 'Vedado' , 'Habana Vieja' ],[ 'Habana
24 del Este' ]
25
26 nx.spring_layout(A, k=1, iterations=200, threshold=0.0001,
27 weight='weight', scale=0.5)
28
29 nx.draw_networkx_nodes(A, posicion ,
30 node_size=800, node_color= 'y')
31 nx.draw_networkx_edges(A, posicion , edgelist=bl , width=10, alpha
32 =0.5,
33 edge_color='b')
34 nx.draw_networkx_edges(A, posicion , edgelist=red , width=5, alpha
35 =0.5,
36 edge_color='r')
37 nx.draw_networkx_edges(A, posicion , edgelist=black , width=2,
38 edge_color='Black')
39 for p in posicion :
40     posicion[p][1] -= 0.01
41 nx.draw_networkx_labels(A, posicion , font_size=11,font_family='arial
42 ',
43 font_color='Black' , font_weight='bold')
44
45 plot.axis( 'off' )
46 plot.savefig("Graf7_spring_layout.eps")
47 plot.show(A)

```

Grafo7.py

9. Multigrafo no dirigido cíclico

Un ejemplo donde podemos utilizar este tipo de grafos es: se quiere representar cuantas llamadas (aristas)se realizaron entre un grupo de personas (vértices), donde nos importa saber la duración de cada llamada durante un día determinado. Este grafo se ve representado en la figura 8.

Algoritmo de diseño. Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 8 el algoritmo *spectral layout*, dando buenos resultados estetico.

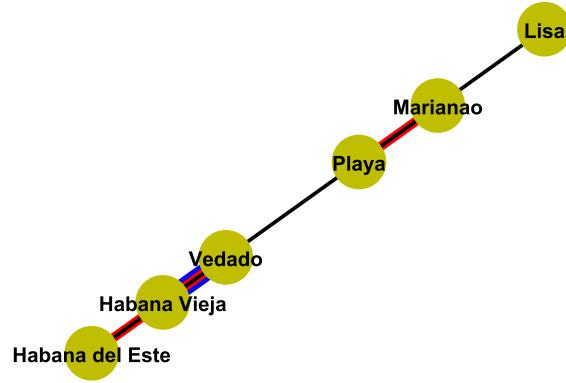


Figura 7: Multigrafo no dirigido acíclico, donde la diferencia de colores de los arcos representan los diferentes caminos.

```

1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4 A=nx.MultiGraph()
5 A.add_edge('Tel_1','Tel_2', weight=2)
6 A.add_edge('Tel_1','Tel_2', weight=3)
7 A.add_edge('Tel_1','Tel_2', weight=4)
8 A.add_edge('Tel_1','Tel_3', weight=2)
9 A.add_edge('Tel_1','Tel_4', weight=4)
10 A.add_edge('Tel_2','Tel_4', weight=2)
11 A.add_edge('Tel_2','Tel_4', weight=3)
12 A.add_edge('Tel_2','Tel_3', weight=2)
13 A.add_edge('Tel_3','Tel_5', weight=2)
14 A.add_edge('Tel_3','Tel_5', weight=3)
15 A.add_edge('Tel_3','Tel_5', weight=4)
16 A.add_edge('Tel_5','Tel_6', weight=2)
17 A.add_edge('Tel_5','Tel_6', weight=3)
18 A.add_edge('Tel_1','Tel_6', weight=2)
19 A.add_edge('Tel_1','Tel_6', weight=3)
20 black=[('Tel_1','Tel_2'), ('Tel_1','Tel_3'), ('Tel_2','Tel_4'),
21 ('Tel_2','Tel_3'), ('Tel_3','Tel_5'), ('Tel_3','Tel_5'),
22 ('Tel_5','Tel_6'), ('Tel_1','Tel_6')]
23
24 red=[('Tel_1','Tel_2'), ('Tel_2','Tel_4'),
25 ('Tel_1','Tel_2'), ('Tel_1','Tel_6'), ('Tel_5','Tel_6')]
26 bl=[('Tel_1','Tel_2'), ('Tel_3','Tel_5')]
27 lista=['Tel_1', 'Tel_2', 'Tel_3', 'Tel_4', 'Tel_5', 'Tel_6']

```

```

29
30     posicion=nx.spectral_layout(A, weight='weight', scale=0.50, center=
31         None, dim=2)
32
33     nx.draw_networkx_nodes(A, posicion,
34         node_size=1000, node_color= 'y')
35     nx.draw_networkx_edges(A, posicion , edgelist=bl, width=10, alpha
36         =0.5,
37         edge_color='b')
38     nx.draw_networkx_edges(A, posicion , edgelist=red , width=5, alpha
39         =0.5,
40         edge_color='r')
41     nx.draw_networkx_edges(A, posicion , edgelist=black , width=2,
42         edge_color='Black')
43
44     nx.draw_networkx_labels(A, posicion , font_size=11,font_family='arial
45         ',
46         font_color='Black', font_weight='bold')
47
48     plot.axis('off')
49     plot.savefig("Graf8_spectral_layout.eps")
50     plot.show(A)

```

Grafo8.py

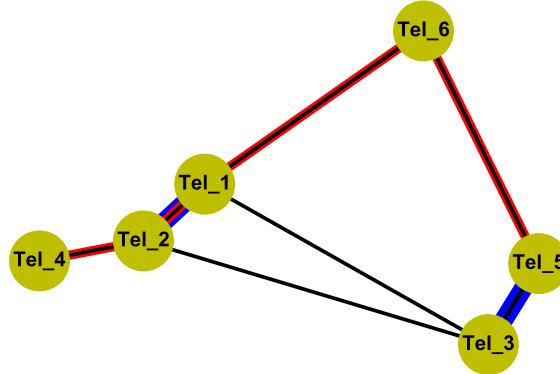


Figura 8: Multigrafo no dirigido cíclico, donde las aristas de diferentes colores representan la existencia de las llamadas con distinta duración.

10. Multigrafo no dirigido reflexivo

Partiendo del ejemplo de la sección 3 se pretende construir un grafo más informativo a la hora de su análisis, por lo que se le agrega como arista (el hecho que las relaciones sexuales mantenidas fueran dentro de una relación formal). Este grafo se muestra en la figura 9.

Algoritmo de diseño. Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 9 el algoritmo *circular layout*, ya que el mismo permite una mejor compresión de este ejemplo mediante la ubicación circular de los nodos.

```
1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4 A=nx.MultiDiGraph()
5
6 A.add_edge('Luis(15)', 'Luis(15)', weight=2)
7 A.add_edge('Maria(15)', 'Yanet(17)', weight=2)
8 A.add_edge('Yanet(17)', 'Yanet(17)', weight=2)
9
10 A.add_edge('Maria(15)', 'Fernando(16)', weight=2)
11 A.add_edge('Fernando(16)', 'Desisy(18)', weight=3)
12 A.add_edge('Fernando(16)', 'Desisy(18)', weight=2)
13 A.add_edge('Pedro(17)', 'Pedro(17)', weight=2)
14 A.add_edge('Pedro(17)', 'Julia(18)', weight=2)
15 A.add_edge('Pedro(17)', 'Rosi(16)', weight=3)
16 A.add_edge('Pedro(17)', 'Rosi(16)', weight=2)
17 A.add_edge('Fernando(16)', 'Claudia(16)', weight=2)
18
19 nodes_reflex = {'Luis(15)', 'Fernando(16)', 'Julia(18)'}
20 nodes_no_reflex = {'Maria(15)', 'Yanet(17)', 'Rosi(16)', 'Desisy(18)', 'Pedro(17)', 'Claudia(16)'}
21
22 black=[('Maria(15)', 'Yanet(17')),
23     ('Maria(15)', 'Fernando(16)'), ('Fernando(16)', 'Desisy(18)'),
24     ('Yanet(17)', 'Julia(18)'), ('Desisy(18)', 'Pedro(17)'),
25     ('Pedro(17)', 'Julia(18)'), ('Pedro(17)', 'Rosi(16)'),
26     ('Fernando(16)', 'Claudia(16)')]
27
28 bl=[('Fernando(16)', 'Desisy(18)'), ('Pedro(17)', 'Rosi(16)')]
29
30 lista=[['Luis(15)', 'Julia(18)', 'Yanet(17)', 'Rosi(16)', 'Desisy(18)', 'Claudia(16)'], ['Maria(15)', 'Fernando(16)', 'Pedro(17)']]
31
32 posicion=nx.circular_layout(A, scale=0.5, center=None, dim=2)
33
34 nx.draw_networkx_nodes(A, posicion, nodelist=nodes_reflex,
35                         node_size=1800, node_color='#ea6d6d')
36 nx.draw_networkx_nodes(A, posicion, nodelist=nodes_no_reflex,
37                         node_size=1500, node_color='#cde95b')
38 nx.draw_networkx_edges(A, posicion, edgelist=bl, width=5, alpha
39                         =0.5,
40                         edge_color='b')
```

```

41 nx.draw_networkx_edges(A, posicion , edgelist=black , width=2,
42 edge_color='Black')
43 nx.draw_networkx_labels(A, posicion , font_size=11,font_family='arial
44 ' ,
45 font_color='#4f5148' , font_weight='bold')
46 plot.axis('off')
47 plot.savefig("Graf9_circular_layout.eps")
48 plot.show(A)

```

Grafo9.py

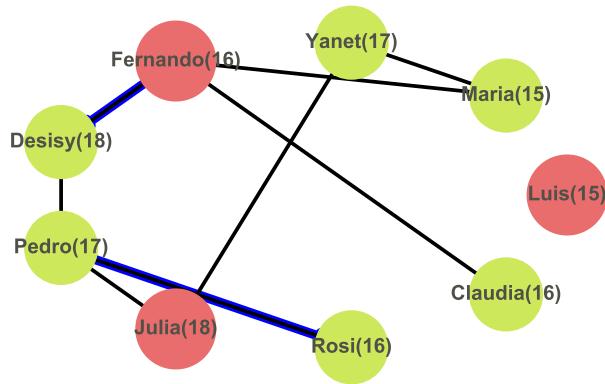


Figura 9: Multigrafo no dirigido reflexivo, donde las aristas de diferentes colores representan la existencia de una relación formal entre los individuos y los vértices de color rojo representa los vértices con aristas reflexivas.

11. Multigrafo dirigido acíclico

Considerando que se quiere trazar las diferentes rutas (teniendo en cuenta el sentido de los mimos) que llevan de una provincia a otra en Cuba (Pinar del Río, Artemisa, La Habana, Mayabeque, Matanza, Cienfuegos, Villa Clara). Tomando como vértices las provincias y como aristas las carreteras que las unen, como muestra la figura 10.

Algoritmo de diseño. Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 10 el algo-

ritmo *spectral layout*, ya que el mismo logró un buen resultado en la ubicación de los nodos facilitando la compresión del grafo.

```

1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4 A=nx.MultiDiGraph()
5 A.add_edge( 'PR' , 'A' , weight=2)
6 A.add_edge( 'A' , 'LH' , weight=2)
7 A.add_edge( 'A' , 'LH' , weight=4)
8 A.add_edge( 'LH' , 'May' , weight=2)
9 A.add_edge( 'May' , 'Mat' , weight=2)
10 A.add_edge( 'May' , 'Mat' , weight=4)
11 A.add_edge( 'LH' , 'May' , weight=4)
12 A.add_edge( 'LH' , 'May' , weight=5)
13 A.add_edge( 'May' , 'Mat' , weight=5)
14 A.add_edge( 'Mat' , 'C' , weight=2)
15 A.add_edge( 'Mat' , 'C' , weight=4)
16
17 black=[( 'PR' , 'A' ) ,( 'LH' , 'May' ) ,( 'May' , 'Mat' ) ,
18 ( 'Mat' , 'C' ) ,( 'A' , 'LH' )]
19 red=[( 'A' , 'LH' ) ,( 'May' , 'Mat' ) ,
20 ( 'Mat' , 'C' ) ,( 'LH' , 'May' )]
21 bl=[( 'May' , 'Mat' ) ,( 'LH' , 'May' )]
22 lista=[ 'PR' , 'A' , 'Mat' , 'May' , 'LH' ],[ 'C' ]
23
24 posicion=nx.spectral_layout(A, weight='distans', scale=0.50, center
25 =None, dim=2)
26
27 nx.draw_networkx_nodes(A, posicion ,
28 node_size=500, node_color= 'y')
29 nx.draw_networkx_edges(A, posicion , edgelist=bl, width=10, alpha
30 =0.5,
31 edge_color='b')
32 nx.draw_networkx_edges(A, posicion , edgelist=red, width=5, alpha
33 =0.5,
34 edge_color='r')
35 nx.draw_networkx_labels(A, posicion , font_size=11,font_family='arial
36 ,
37 font_color='Black', font_weight='bold')
38
39 plot.axis( 'off' )
plot.savefig("Graf10_spectral_layout.eps")
plot.show(A)
```

Grafo10.py

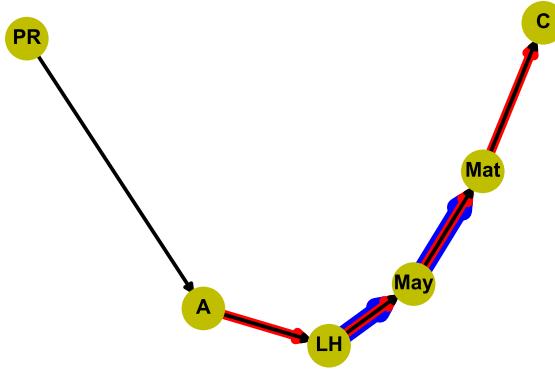


Figura 10: Multigrafo dirigido acíclico, donde las aristas de diferentes colores representan la existencia de más de una ruta entre las diferentes provincias.

12. Mltigrafo dirigido cíclico

“En el caso particular de que las redes reflejen una realidad social, los nodos pueden representar personas o entidades relacionadas con sus contextos, y las conexiones representarán relaciones sociales existentes entre ellos (amistad, parentesco, membresía, afinidad, etc.). A pesar de que intuitivamente las redes sociales se asemejan a los grafos matemáticos, es más habitual que en ellas se trabaje con distintos tipos de relaciones” [5] por lo que es necesario la utilización de multígrafos, que es la herramienta que contempla más de una relación entre dos nodos, con esto ganamos mayor riqueza en los datos a analizar. Por ejemplo, tenemos un grupo de personas que laboran en un departamento de Informática y se hace una encuesta donde se les pide que marque cuales de tres sentimientos (respeto, afinidad, rechazo) sienten por sus compañeros de trabajo, como muestra la figura 11.

Algoritmo de diseño. Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 11 el algoritmo *kamada kawai layout*, ya que el mismo logró un buen resultado estético.

```

1 import networkx as nx
2 import matplotlib.pyplot as plt
3

```

```

4 A=nx.MultiDiGraph()
5 A.add_edge('Personal','Persona2', weight=2)
6 A.add_edge('Personal','Persona2', weight=3)
7 A.add_edge('Personal','Persona2', weight=4)
8 A.add_edge('Personal','Persona3', weight=2)
9 A.add_edge('Personal','Persona4', weight=4)
10 A.add_edge('Persona2','Persona4', weight=2)
11 A.add_edge('Persona2','Persona4', weight=3)
12 A.add_edge('Persona4','Persona3', weight=3)
13 A.add_edge('Persona3','Persona2', weight=2)
14 A.add_edge('Persona3','Persona5', weight=2)
15 A.add_edge('Persona3','Persona5', weight=3)
16 A.add_edge('Persona3','Persona5', weight=4)
17 A.add_edge('Persona5','Persona6', weight=2)
18 A.add_edge('Persona5','Persona6', weight=3)
19 A.add_edge('Personal','Persona6', weight=2)
20 A.add_edge('Personal','Persona6', weight=3)
21 black=[('Personal','Persona2'), ('Personal','Persona3'), ('Persona2',
22   , 'Persona4'),
23   ('Persona3','Persona2'), ('Persona3','Persona5'), ('Persona3',
24   , 'Persona5'),
25   ('Persona5','Persona6'), ('Personal','Persona6'), ('Persona4',
26   , 'Persona3')]
27 red=[('Personal','Persona2'), ('Persona2','Persona4'),
28   ('Personal','Persona2'), ('Personal','Persona6'), ('Persona5',
29   , 'Persona6')]
30 bl=[('Personal','Persona2'), ('Persona3','Persona5')]
31 lista=['Personal','Persona2','Persona3','Persona4','Persona5'],
32   ['Persona6']
33 posicion=nx.kamada_kawai_layout(A, dist=None, pos=None, weight='
34   weight', scale=0.30, center=None, dim=2)
35 nx.draw_networkx_nodes(A, posicion,
36   node_size=500
37   , node_color='y')
38 nx.draw_networkx_edges(A, posicion, edgelist=bl, width=10, alpha
39   =0.5,
40   edge_color='b')
41 nx.draw_networkx_edges(A, posicion, edgelist=red, width=5, alpha
42   =0.5,
43   edge_color='r')
44 nx.draw_networkx_edges(A, posicion, edgelist=black, width=2,
45   edge_color='#297033')
46 for p in posicion:
47   posicion[p][1]+= 0.07
48 nx.draw_networkx_labels(A, posicion, font_size=11, font_family='arial
49   ',
50   font_color='Black', font_weight='bold')
51 plot.axis('off')
52 plot.savefig("Graf11_kamada_kawai_layout.eps")
53 plot.show(A)

```

Grafo11.py

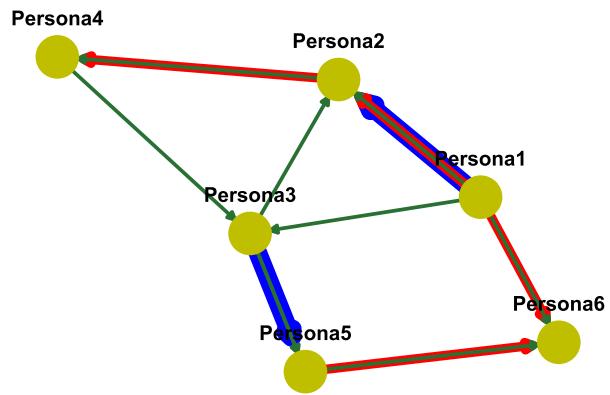


Figura 11: Multigrafo dirigido cíclico, donde las aristas de diferentes colores representan la existencia de más de una opinión sobre sus compañeros.

13. Multigrafo dirigido reflexivo

Una de las aplicaciones de estos grafos es en el modelado del funcionamiento de una máquina de estados. Por ejemplo, se desea modelar un grafo que represente el funcionamiento de una máquina muy sencilla de golosinas, que responde a las siguientes reglas:

- 1 Cada golosina vale \$ 0.25.
- 2 La máquina acepta sólo monedas de \$ 0.10 y de \$ 0.05.
- 3 La máquina *no* da vuelto.

Para modelar el grafo, debemos considerar distintos “estados de dinero ingresado”, y como se va pasando de uno a otro. Por ejemplo, si en un momento tenemos ingresados 5 centavos, y agregamos 5 centavos más, pasamos a otro estado, que es el mismo que si hubiésemos ingresado 10 centavos al principio. Llámemos A, B, C, D, E y F a los estados que representan 0, 5, 10, 15, 20, 25 centavos ingresados respectivamente. Las transiciones de un estado a otro se harán por ingreso de 5 o 10 centavos, o al presionar el botón para obtener los caramelos (G). Como se muestra en la figura 12.

Algoritmo de diseño. Después de probar con varios de los algoritmos de diseño existentes, se decide usar para la representación del grafo de la figura 12 el algoritmo *spectral layout*, ya que este algoritmo mostro el mejor de los resultados a la hora de representar este ejemplo.

```
1 import networkx as nx
2 import matplotlib.pyplot as plot
3
4 A=nx.MultiDiGraph()
5 A.add_edge('A','A', weight=2)
6 A.add_edge('A','B', weight=2)
7 A.add_edge('A','C', weight=2)
8 A.add_edge('B','B', weight=2)
9 A.add_edge('B','C', weight=2)
10 A.add_edge('B','D', weight=2)
11 A.add_edge('C','C', weight=2)
12 A.add_edge('C','D', weight=2)
13 A.add_edge('C','E', weight=2)
14 A.add_edge('D','D', weight=2)
15 A.add_edge('D','E', weight=2)
16 A.add_edge('D','F', weight=2)
17 A.add_edge('E','E', weight=2)
18 A.add_edge('E','F', weight=2)
19 A.add_edge('E','F', weight=4)
20 A.add_edge('F','F', weight=2)
21 A.add_edge('F','A', weight=2)
22
```

```

23 black=[( 'A' , 'A' ) ,( 'A' , 'B' ) ,( 'A' , 'C' ) ,
24 ( 'B' , 'B' ) ,( 'B' , 'C' ) ,( 'B' , 'D' ) ,( 'C' , 'C' ) ,( 'C' , 'D' ) ,( 'C' , 'E' ) ,
25 ( 'D' , 'D' ) ,( 'D' , 'E' ) ,( 'D' , 'F' ) ,( 'E' , 'E' ) ,( 'E' , 'F' ) ,( 'E' , 'F' ) ,( 'F'
26 , 'A' )]
27 bl=[( 'E' , 'F' )]
28 lista=['A' , 'B' , 'C' , 'D' , 'E' ] ,[ 'F' ]
29
30 posicion=nx.spectral_layout(A, weight='distans', scale=0.50, center
31 =None, dim=2)
32 nx.draw_networkx_nodes(A, posicion ,
33 node_size=600, node_color= '#48b457')
34 nx.draw_networkx_edges(A, posicion , edgelist=bl , width=10, alpha
35 =0.5,
36 edge_color='b' , arrowsize=20)
37 nx.draw_networkx_edges(A, posicion , edgelist=black , width=2,
38 edge_color='Black' , arrowsize=20)
39 nx.draw_networkx_labels(A, posicion , font_size=14,font_family='arial
40 ,
41 font_color='Black' , font_weight='bold')
42 plot.axis('off')
43 plot.savefig("Graf12_spectral_layout.eps")
44 plot.show(A)

```

Grafo12.py

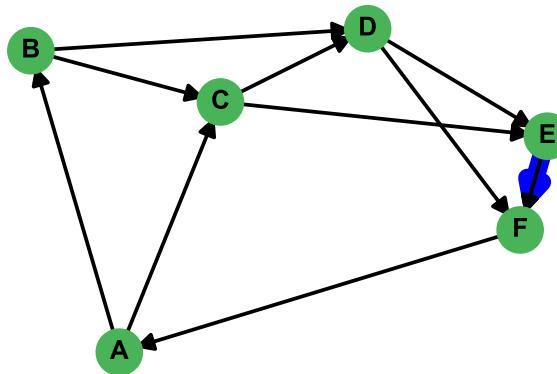


Figura 12: Multigrafo dirigido reflexivo, donde las aristas de diferentes colores representan la existencia de más de una forma de pasar de un estado a otro.

Referencias

- [1] Andrés Aiello and Rodrigo Ignacio Silveira. Trazado de grafos mediante métodos dirigidos por fuerzas: revisión del estado del arte y presentación de algoritmos para grafos donde los vértices son regiones geográficas. Technical report, Universidad de Buenos Aires, <https://dccg.upc.edu/people/rodrigo/pubs/MScThesis.pdf>, Diciembre 2004.
- [2] Amalia Duch Brown. Grafos. <https://www.cs.upc.edu/~duch/home/duch/grafos.pdf>, Octubre 2007.
- [3] Chakraborty, Anwesha, Trina Dutta, Mondal, Sushmita, Nath, and Asoke. Application of graph theory in social media. *International Journal of Computer Sciences and Engineering*, 6:722–729, 10 2018.
- [4] NetworkX Developers Copyright 2004-2019. Overview of networkx. <https://networkx.github.io/documentation/latest/index.html>, Febrero 2019.
- [5] R. A. Hanneman and M. Riddle. Introduction to social network methods. <http://faculty.ucr.edu/~hanneman/>. Consultado, September 2013.
- [6] Kimball Martin. *Graph Theory and Social Networks*. Spring, Abril 2014.
- [7] Azahara Muñoz Martínez Mauricio Beltrán Pascuala and Ángel Muñoz Alamillos. Redes bayesianas aplicadas a problemas de credit scoring, una aplicación práctica. <https://www.sciencedirect.com/science/article/pii/S0210026613000083>, Octubre 2013.

G. S

Tarea 3

5271
19 de marzo de 2019

1. Algoritmos utilizados en las mediciones

Se seleccionaron cinco de los doce algoritmos para ejecutarlos sobre cinco de los grafos de tareas anteriores, midiendo los tiempos de ejecución de cada algoritmo y realizando un análisis de los datos obtenidos. Dado los requerimientos de los algoritmos escogidos para la realización de esta tarea, se tuvo modificar los dichos grafos agregándoles más nodos y se convirtiéndolos a grafos no dirigidos, como se muestra en la figura 1.

Los algoritmos escogidos fueron los siguientes:

- *Make max clique graph* (encuentra las camarillas máximas y las trata como vértices . Los vértices están conectados si tienen miembros comunes en el grafo original) [1].
- *Betweenness centrality* (calcula la centralidad de intermediación de ruta más corta para los vértices . La centralidad de la intermediación es una forma de detectar la cantidad de influencia que un vértice tiene sobre el flujo de información en un grafo) [2].
- *Greedy color* (colorea un grafo usando varias estrategias de coloración codiciosa. Las estrategias se pueden describir como el intento de colorear un grafo con la menor cantidad de colores posibles, donde ningún vecino puede tener el mismo color) [3].
- *Maximal matching* (encuentra una cardinalidad máxima de coincidencia en el grafo. Una coincidencia es un subconjunto de aristas en las que no se produce ningún vértice más de una vez. La cardinalidad de una coincidencia es el número de arcos coincidentes. Se uso este algoritmo en lugar del *min maximal matching* ya que este daba un error en networkx)[4].
- *Dfs tree* (crea un árbol orientado hacia el retorno construido a partir de una fuente en una búsqueda en profundidad) [5].

```

1 import random as rnd
2 import networkx as nx
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 import warnings
6 warnings.filterwarnings("ignore")
7 def Crear_Grafos(nombre, size, int_distancia):
8
9     G=nx.Graph()
10    nodes = []
11    for i in range(size):
12        nodes.append(i)
13    print(nodes)
14    for i in nodes:
15        idx = nodes.index(i) + 1
16        print(idx)
17        for j in nodes[idx:len(nodes)]:
18            if rnd.randint(0,80)==1:
19                G.add_edge(i,j, distancia=rnd.randint(1,
20 int_distancia))
21    print(G.edges)
22
23    df = pd.DataFrame()
24    df = nx.to_pandas_adjacency(G, dtype=int, weight='distancia')
25    df.to_csv(nombre+'.csv', index=None, header=None)
26    plt.figure(figsize=(15,15))
27    position = nx.spring_layout(G, scale=5, iterations=200)
28    nx.draw_networkx_nodes(G, position, node_size=50, node_color="#
29    de8919", node_shape="<")
30    nx.draw_networkx_edges(G, position, width=0.5, edge_color='
31    black')
32
33    plt.axis("off")
34    plt.savefig(nombre + ".png", bbox_inches='tight')
35    plt.savefig(nombre + ".eps", bbox_inches='tight')
36    plt.show(G)
37
38 Crear_Grafos("1NoDirigido",800, 20)

```

Genera_grafos.py

2. Medición de los tiempos de ejecución

Con el objetivo de realizar las mediciones de los tiempos de ejecución de los algoritmos seleccionados se desarrolló el siguiente código.

En primer lugar, se crea una función que lee de un archivo la matriz de adyacencia de un grafo y convierte dicha matriz en un grafo.

```
1 def Leer_grafos(nomb):
2     dr = pd.read_csv( nomb , header = None)
3     A = nx.from_pandas_adjacency(dr, create_using = nx.Graph())
4     print(A.edges)
5     A.name=(i)
6     return (A)
```

Corer_Algoritmos.py

Se crea una función para cada algoritmo. En esta se le pasa el grafo leído por parámetros al algoritmo en cuestión, en este proceso se mide el tiempo de ejecución del algoritmo y se repite treinta veces la ejecución, los tiempos son almacenados en una lista. Con las mediciones acumuladas se calcula la media, la mediana y la desviación estándar.

```
1 def dfs_tree(grafo):
2     tiempo=[]
3     tiempo_inicial = dt.datetime.now()
4     for i in range(30):
5         tiempo_inicial = dt.datetime.now()
6         for j in range(1000):
7             nx.dfs_tree(grafo)
8             tiempo_final = dt.datetime.now()
9             tiempo_ejecucion = (tiempo_final - tiempo_inicial).
10            total_seconds()
11            tiempo.append(tiempo_ejecucion)
12
13            media=nup.mean(tiempo)
14            desv=nup.std(tiempo)
15            mediana=nup.median(tiempo)
16            datos[ "algoritmo" ].append("dfs_tree")
17            datos[ "grafo" ].append(grafo.name)
18            datos[ "cant_vertice" ].append(grafo.number_of_nodes())
19            datos[ "cant_arista" ].append(grafo.number_of_edges())
20            datos[ "media" ].append(media)
21            datos[ "desv" ].append(desv)
22            datos[ "mediana" ].append(mediana)
23            return datos
24
25 def Greedy_color(grafo):
26     tiempo=[]
27     for i in range(30):
28         tiempo_inicial = dt.datetime.now()
```

```

28     for j in range(1000):
29         nx.greedy_color(grafo)
30         tiempo_final = dt.datetime.now()
31         tiempo_ejecucion = (tiempo_final - tiempo_inicial).
32             total_seconds()
33         tiempo.append(tiempo_ejecucion)
34
35 media=nup.mean(tiempo)
36 desv=nup.std(tiempo)
37 mediana=nup.median(tiempo)
38 datos[ "algoritmo" ].append("greedy_color")
39 datos[ "grafo" ].append(grafo.name)
40 datos[ "cant_vertice" ].append(grafo.number_of_nodes())
41 datos[ "cant_arista" ].append(grafo.number_of_edges())
42 datos[ "media" ].append(media)
43 datos[ "desv" ].append(desv)
44 datos[ "mediana" ].append(mediana)
45 return datos

```

Corer_Algoritmos.py

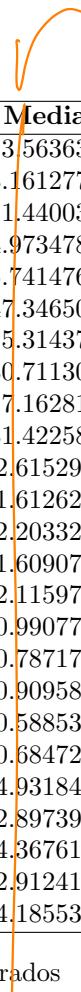
En este otro fragmento se muestra donde se ejecutan las funciones y se guarda el archivo con los datos recopilados.

```

1 listgrafoNoDi=[ "1NoDirigido.csv" , "2NoDirigido.csv" , "3NoDirigido.csv"
2   , "4NoDirigido.csv" , "5NoDirigido.csv" ]
3
4 for i in listgrafoNoDi:
5     l=Leer_grafos(i)
6     make.max_clique_graph(l)
7 for i in listgrafoNoDi:
8     l=Leer_grafos(i)
9     Betweenness_centrality(l)
10    for i in listgrafoNoDi:
11        l=Leer_grafos(i)
12        Greedy_color(l)
13        for i in listgrafoNoDi:
14            l=Leer_grafos(i)
15            Maximal_matching(l)
16            for i in listgrafoNoDi:
17                l=Leer_grafos(i)
18                dfs_tree(l)
19
20 df = pd.DataFrame(datos)
21 df.to_csv("salid.csv", index=None)

```

Corer_Algoritmos.py



Algoritmos	Nodos	Media
make max clique graph	800	13.563634
make max clique graph	400	8.1612773
make max clique graph	640	11.440034
make max clique graph	500	4.9734789
make max clique graph	710	5.7414766
betweenness centrality	800	47.346500
betweenness centrality	400	15.314375
betweenness centrality	640	30.711309
betweenness centrality	500	17.162814
betweenness centrality	710	31.422583
greedy color	800	2.615292
greedy color	400	1.612627
greedy color	640	2.203329
greedy color	500	1.609072
greedy color	710	2.115973
maximal matching	800	0.990775
maximal matching	400	0.787178
maximal matching	640	0.909580
maximal matching	500	0.588539
maximal matching	710	0.684729
dfs tree	800	4.931849
dfs tree	400	2.897395
dfs tree	640	4.367618
dfs tree	500	2.912410
dfs tree	710	4.185530

Cuadro 1: Tabla de valores capturados

3. Resultados del Análisis de los datos

Con los datos recopilado de las ejecuciones de los Algoritmos par los cinco grafos se realizó un histograma, como se muestra en la figura 2 y dos diagramas de dispersión, uno que muestra la relación de la media de los tiempos de ejecución de cada algoritmo con la cantidad de vértices y el otro con la cantidad de aristas, como se muestra en la figura 3 y 4 respectivamente. En estas figuras se puede observar que el algoritmo con los tiempos de ejecución más pequeños es el *Maximal matching*, así como que el de mayores tiempos es el *Betweenness centrality*. En sentido general se observa que cuando aumentan la cantidad de nodos y aristas aumenta el tiempo de ejecución de los algoritmos.

```

1 dr = pd.read_csv( "salid.csv" )
2 print(dr[ "media" ][10:15])
3 fig ,axes = plt.subplots(nrows=2, ncols=3, figsize=(12, 6))
4 axes [0 ,0]. hist(dr[ "media" ][:5] ,bins=5, color="#932525" , alpha=1,
      edgecolor = 'black' , linewidth=1)

```

```

5 axes[0,0].set_title(dr["algoritmo"][[1]])
6
7 axes[0,0].set_ylabel('Frecuencia de ocurrencia')
8 axes[0,1].hist(dr["media"][5:10], bins=5, color="#ee9110", alpha=1,
9   edgecolor = 'black', linewidth=1)
10 axes[0,1].set_title(dr["algoritmo"][[6]])
11
12 axes[0,1].set_ylabel('Frecuencia de ocurrencia')
13 axes[0,2].hist(dr["media"][10:15], bins=5,color="#adcb18", alpha=1,
14   edgecolor = 'black', linewidth=1)
15 axes[0,2].set_title(dr["algoritmo"][[11]])
16
17 axes[0,2].set_ylabel('Frecuencia de ocurrencia')
18 axes[1,0].hist(dr["media"][15:20], bins=5,color="#129f10", alpha=1,
19   edgecolor = 'black', linewidth=1)
20 axes[1,0].set_title(dr["algoritmo"][[16]])
21
22 axes[1,0].set_ylabel('Frecuencia de ocurrencia')
23 axes[1,1].hist(dr["media"][20:25], bins=5,color="#093ea8", alpha=1,
24   edgecolor = 'black', linewidth=1)
25 axes[1,1].set_title(dr["algoritmo"][[21]])
26
27 plt.savefig("Histograma.eps", bbox_inches='tight')
28 plt.savefig("Histograma.png", bbox_inches='tight')
29 plt.show()

```

Histograma.py

```

1 size = (25 * dr["cant_arista"][5:10] / dr["cant_vertice"][5:10])
2 color_names = ["#932525", "#129f10", "#093ea8", "#adcb18", "#ee9110"]
3
4 figure, axes = plt.subplots(figsize=(8, 8))
5 axes.scatter(dr["media"][:5], dr["cant_vertice"][:5],
6             s=size, c=color_names, marker="D",
7             label="Make max clique graph", alpha=0.8, edgecolors='black')
8 axes.scatter(dr["media"][5:10], dr["cant_vertice"][5:10],
9             s=size, c=color_names, marker="s",
10            label="Betweenness centrality", alpha=0.8, edgecolors='black')
11 axes.scatter(dr["media"][10:15], dr["cant_vertice"][10:15],
12             s=size, c=color_names, marker="8",
13            label="Greedy color algorithm", alpha=0.8, edgecolors='black')
14 axes.scatter(dr["media"][15:20], dr["cant_vertice"][15:20],
15             s=size, c=color_names, marker=">",
16            label="Maximal matching", alpha=0.8, edgecolors='black')
17 axes.scatter(dr["media"][20:25], dr["cant_vertice"][20:25],
18             s=size, c=color_names, marker="*",
19            label="Dfs_tree", alpha=0.8, edgecolors='black')
20 axes.set_ylabel("Vertices ", fontsize=12, fontfamily="arial",
21                 fontweight="bold")
22 axes.set_xlabel("Tiempo de Ejecucion", fontsize=12, fontfamily="arial",
23                  fontweight="bold")
24 plt.ylim((min(dr["cant_vertice"])) - 30, max(dr["cant_vertice"]) + 30)
25
26 axes.legend()
27 plt.savefig("DiagramVertices.eps", bbox_inches='tight')

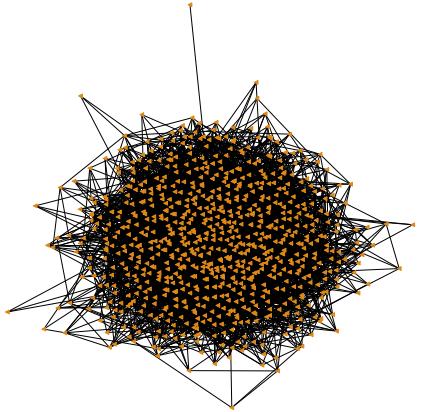
```

Histograma.py

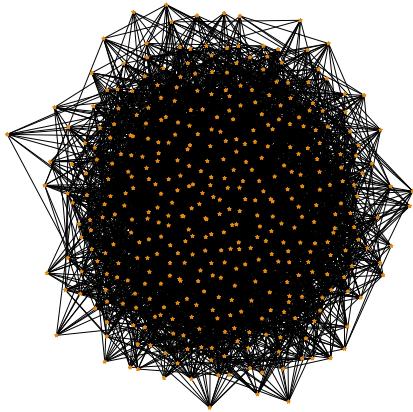


Referencias

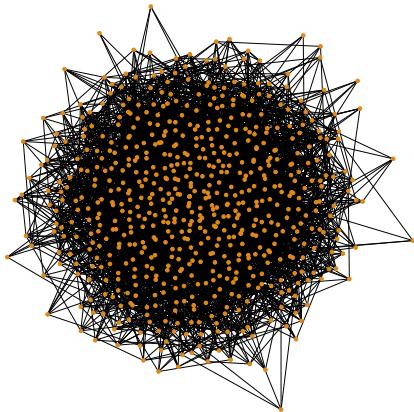
- [1] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.cliques.make_max_clique_graph.html. Accessed: 18-03-2019.
- [2] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.centrality.betweenness_centrality.html#networkx.algorithms.centrality.betweenness_centrality. Accessed: 18-03-2019.
- [3] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.coloring.greedy_color.html. Accessed: 18-03-2019.
- [4] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.matching.maximal_matching.html. Accessed: 18-03-2019.
- [5] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.traversal.depth_first_search.dfs_tree.html#networkx.algorithms.traversal.depth_first_search.dfs_tree. Accessed: 18-03-2019.



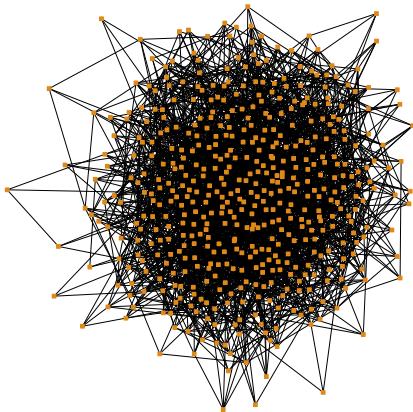
(a) grafo1, 800 vértices, 3966 aristas



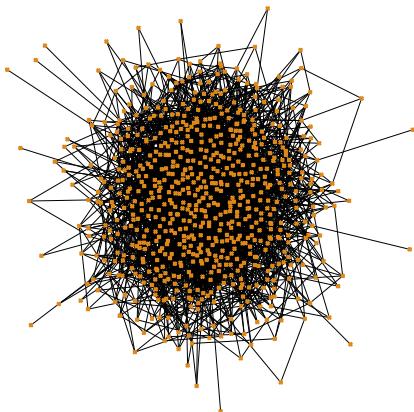
(b) grafo2, 400 vértices, 3780 aristas



(c) grafo3, 640 vértices, 3850 aristas

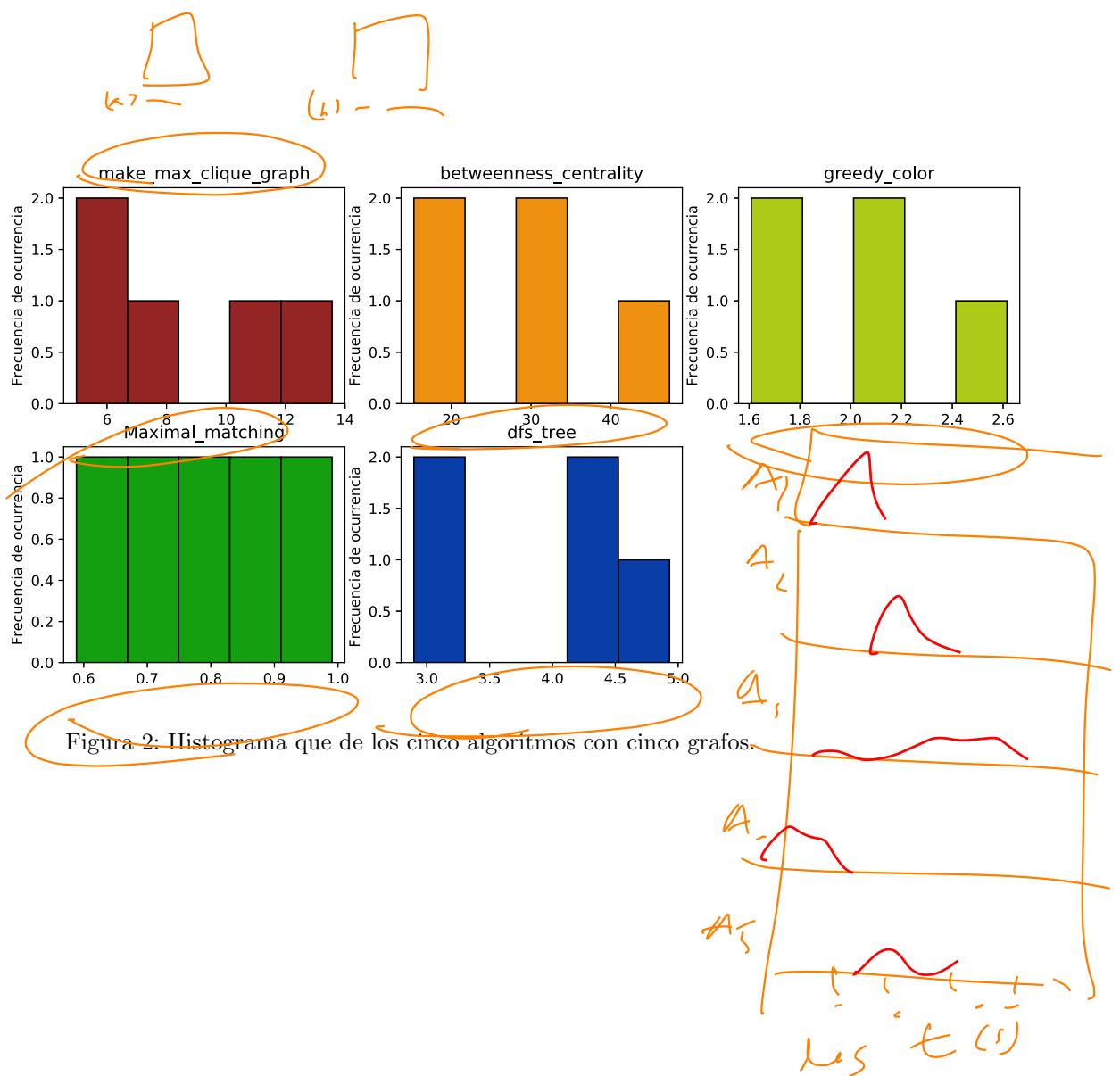


(d) grafo4, 500 vértices, 2439 aristas



(e) grafo5, 710 vértices, 2478 aristas

Figura 1: Grafos generados para la tarea 3.



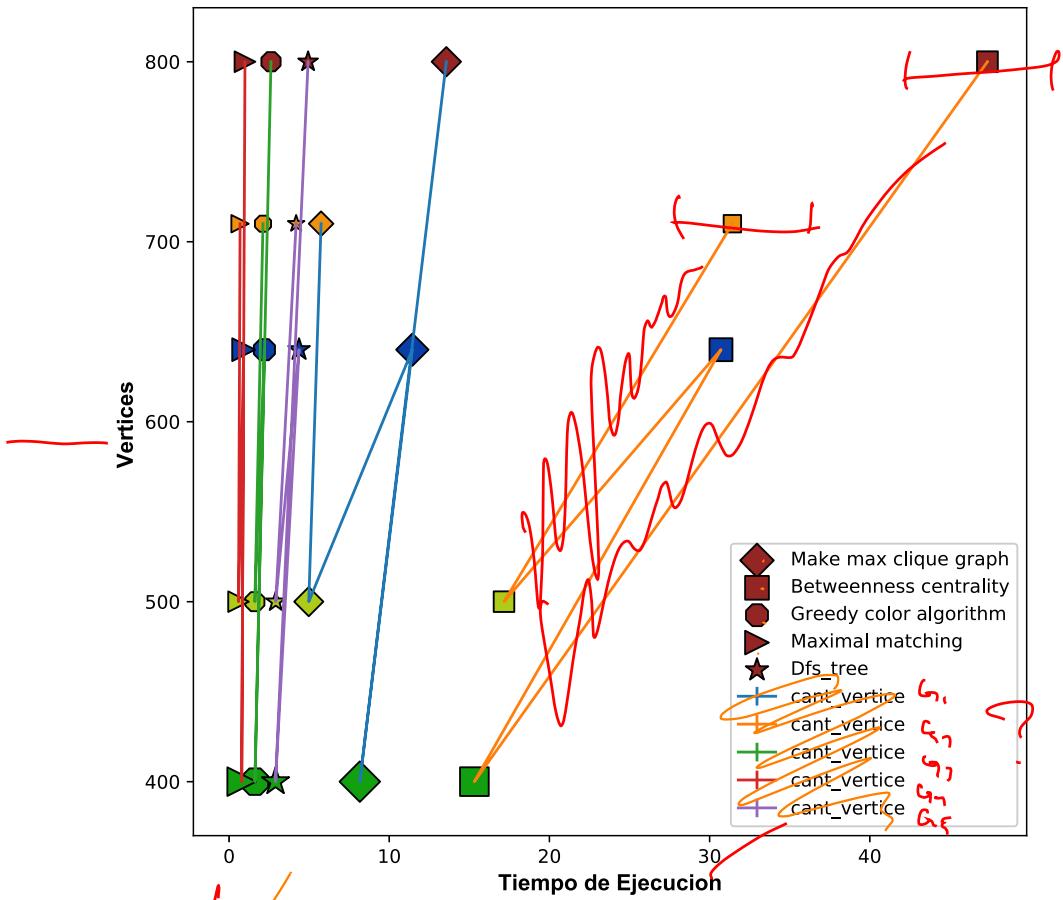


Figura 3. Diagrama de dispersión que muestra la relación entre los tiempos de ejecución(en segundos) y la cantidad de vértices, los diferentes colores indican los cinco grafos(graf01(rojo), graf02(verde), graf03(azul), graf04(amarillo), graf04(naranja)).

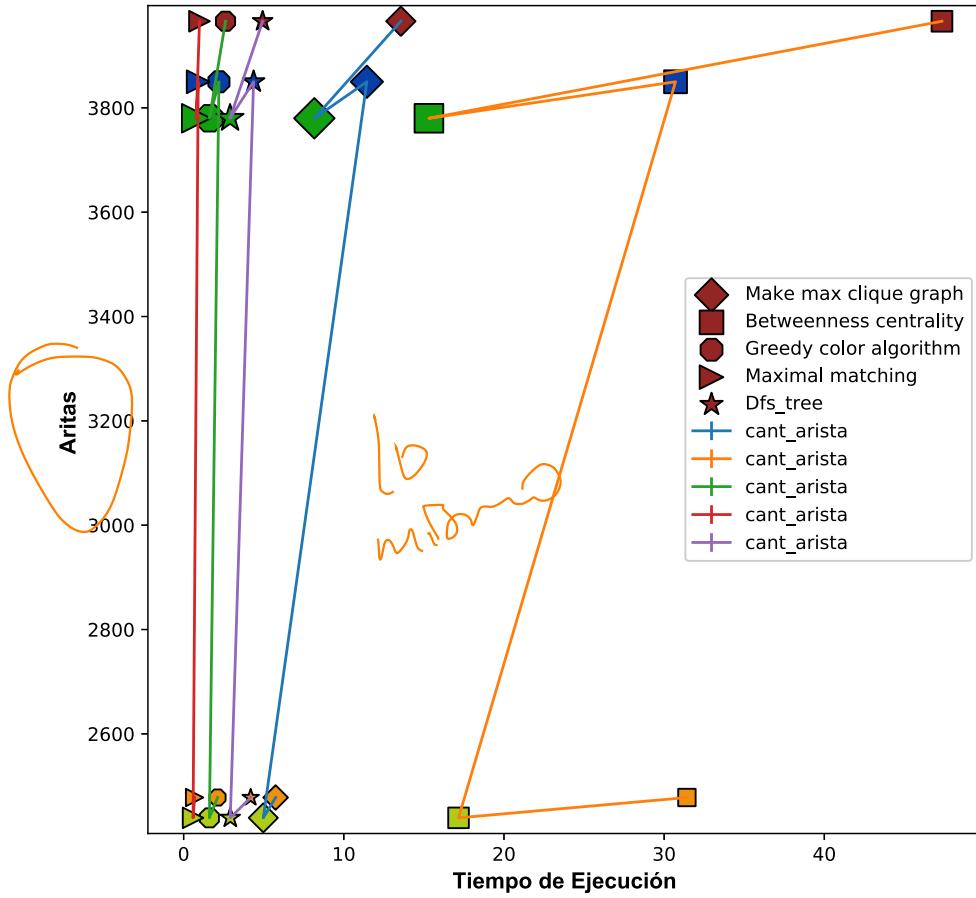


Figura 4: Diagrama de dispersión que muestra la relación entre los tiempos de ejecución(en segundos) y la cantidad de aristas,los diferentes colores indican los cinco grafos(graf01(rojo), grafo2(verde), grafo3(azul), grafo4(amarillo), grafo4(naranja)).

Tarea 3

5271

26 de mayo de 2019

1. Algoritmos utilizados en las mediciones

Se seleccionaron cinco de los doce algoritmos para ejecutarlos sobre cinco de los grafos de tareas anteriores, midiendo los tiempos de ejecución de cada algoritmo y realizando un análisis de los datos obtenidos. Dado los requerimientos de los algoritmos escogidos para la realización de esta tarea, se tuvo modificar los dichos grafos agregándoles más nodos y se convirtiéndolos a grafos no dirigidos, como se muestra en la figura 1.

Los algoritmos escogidos fueron los siguientes:

- *Make max clique graph* (encuentra las camarillas máximas y las trata como vértices. Los vértices están conectados si tienen miembros comunes en el grafo original) [1].
- *Betweenness centrality* (calcula la centralidad de intermediación de ruta más corta para los vértices. La centralidad de la intermediación es una forma de detectar la cantidad de influencia que un vértice tiene sobre el flujo de información en un grafo) [2].
- *Greedy color* (colorea un grafo usando varias estrategias de coloración codiciosa. Las estrategias se pueden describir como el intento de colorear un grafo con la menor cantidad de colores posibles, donde ningún vecino puede tener el mismo color) [3].
- *Maximal matching* (encuentra una cardinalidad máxima de coincidencia en el grafo. Una coincidencia es un subconjunto de aristas en las que no se produce ningún vértice más de una vez. La cardinalidad de una coincidencia es el número de arcos coincidentes. Se uso este algoritmo en lugar del *min maximal matching* ya que este daba un error en networkx) [4].
- *Dfs tree* (crea un árbol orientado hacia el retorno construido a partir de una fuente en una búsqueda en profundidad) [5].

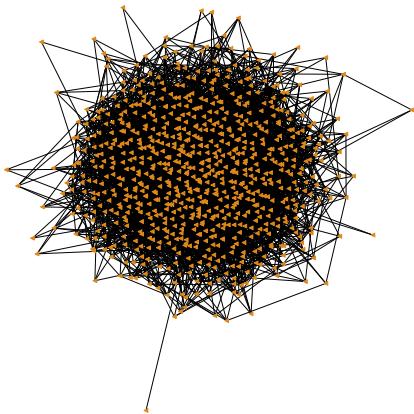
```
1 import random as rnd
2 import networkx as nx
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 import warnings
6 warnings.filterwarnings("ignore")
7 def Crear_Grafos(nombre, size, int_distancia):
8
9     G=nx.Graph()
10    nodes = []
11    for i in range(size):
```

```

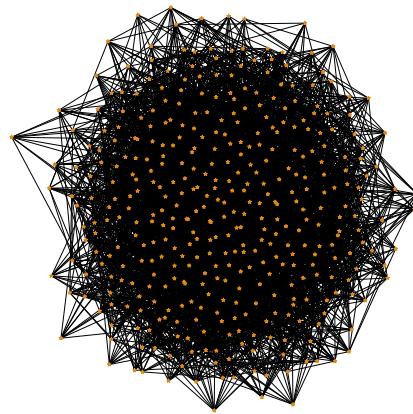
12     nodes.append(i)
13 print(nodes)
14 for i in nodes:
15     idx = nodes.index(i) + 1
16     print(idx)
17     for j in nodes[idx:len(nodes)]:
18         if rnd.randint(0,80)==1:
19             G.add_edge(i,j, distancia=rnd.randint(1, int_distancia))
20 print(G.edges)
21
22 df = pd.DataFrame()
23 df = nx.to_pandas_adjacency(G, dtype=int, weight='distancia')
24 df.to_csv(nombre+'.csv', index=None, header=None)
25 plt.figure(figsize=(15,15))
26 position = nx.spring_layout(G, scale=5, iterations=200)
27 nx.draw_networkx_nodes(G, position, node_size=50, node_color="#de8919", node_shape="<")
28 nx.draw_networkx_edges(G, position, width=0.5, edge_color='black' )
29
30
31 plt.axis("off")
32 plt.savefig(nombre + ".png", bbox_inches='tight')
33 plt.savefig(nombre + ".eps", bbox_inches='tight')
34 plt.show(G)
35
36 Crear_Grafos("1NoDirigido",800, 20)

```

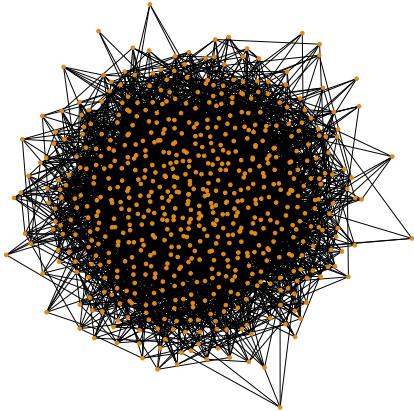
Genera_grafos.py



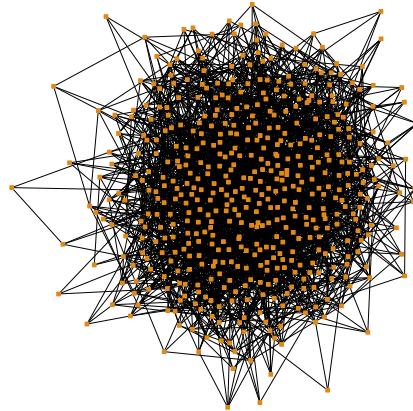
(a) grafo1, 800 vértices, 3966 aristas



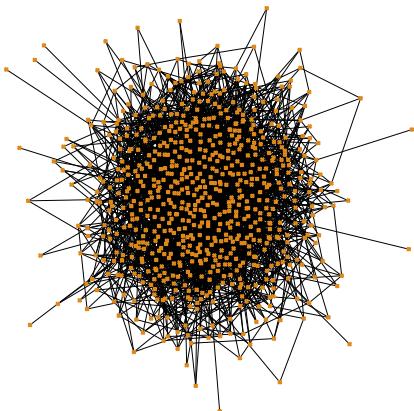
(b) grafo2, 400 vértices, 3780 aristas



(c) grafo3, 640 vértices, 3850 aristas



(d) grafo4, 500 vértices, 2439 aristas



(e) grafo5, 710 vértices, 2478 aristas

Figura 1: Grafos generados para la .

2. Medición de los tiempos de ejecución

Con el objetivo de realizar las mediciones de los tiempos de ejecución de los algoritmos seleccionados se desarrolló el siguiente código.

En primer lugar, se crea una función que lee de un archivo la matriz de adyacencia de un grafo y convierte dicha matriz en un grafo.

```
1 def Leer_grafos(nomb):
2     dr = pd.read_csv( nomb , header = None)
3     A = nx.from_pandas_adjacency(dr, create_using = nx.Graph())
4     print(A.edges)
5     A.name=(i)
6     return (A)
```

Corer_Algoritmos.py

Se crea una función para cada algoritmo. En esta se le pasa el grafo leído por parámetros al algoritmo en cuestión, en este proceso se mide el tiempo de ejecución del algoritmo y se repite treinta veces la ejecución, los tiempos son almacenados en una lista. Con las mediciones acumuladas se calcula la media, la mediana y la desviación estándar.

```
1 def dfs_tree(grafo):
2     tiempo=[]
3     tiempo_inicial = dt.datetime.now()
4     for i in range(30):
5         tiempo_inicial = dt.datetime.now()
6         for j in range(1000):
7             nx.dfs_tree(grafo)
8         tiempo_final = dt.datetime.now()
9         tiempo_ejecucion = (tiempo_final - tiempo_inicial).total_seconds()
10        tiempo.append(tiempo_ejecucion)
11
12    media=nup.mean(tiempo)
13    desv=nup.std(tiempo)
14    mediana=nup.median(tiempo)
15    datos["algoritmo"].append("dfs_tree")
16    datos["grafo"].append(grafo.name)
17    datos["cant_vertice"].append(grafo.number_of_nodes())
18    datos["cant_arista"].append(grafo.number_of_edges())
19    datos["media"].append(media)
20    datos["desv"].append(desv)
21    datos["mediana"].append(mediana)
22    return datos
23
24 def Greedy_color(grafo):
25     tiempo=[]
26     for i in range(30):
27         tiempo_inicial = dt.datetime.now()
28         for j in range(1000):
29             nx.greedy_color(grafo)
30         tiempo_final = dt.datetime.now()
31         tiempo_ejecucion = (tiempo_final - tiempo_inicial).total_seconds()
32         tiempo.append(tiempo_ejecucion)
33
34    media=nup.mean(tiempo)
35    desv=nup.std(tiempo)
36    mediana=nup.median(tiempo)
37    datos["algoritmo"].append("greedy_color")
38    datos["grafo"].append(grafo.name)
```

```

39     datos[ "cant_vertice" ].append(grafo.number_of_nodes())
40     datos[ "cant_arista" ].append(grafo.number_of_edges())
41     datos[ "media" ].append(media)
42     datos[ "desv" ].append(desv)
43     datos[ "mediana" ].append(mediana)
44
45     return datos

```

Corer_Algoritmos.py

En este otro fragmento se muestra donde se ejecutan las funciones y se guarda el archivo con los datos recopilados.

```

1 listgrafoNoDi=[ "1NoDirigido.csv" , "2NoDirigido.csv" , "3NoDirigido.csv" , "4NoDirigido.csv"
2 , "5NoDirigido.csv" ]
3
4 for i in listgrafoNoDi:
5     l=Leer_grafos(i)
6     make_max_clique_graph(l)
7 for i in listgrafoNoDi:
8     l=Leer_grafos(i)
9     Betweenness_centrality(l)
10    for i in listgrafoNoDi:
11        l=Leer_grafos(i)
12        Greedy_color(l)
13    for i in listgrafoNoDi:
14        l=Leer_grafos(i)
15        Maximal_matching(l)
16    for i in listgrafoNoDi:
17        l=Leer_grafos(i)
18        dfs_tree(l)
19
20 df = pd.DataFrame(datos)
21 df.to_csv("salid.csv", index=None)

```

Corer_Algoritmos.py

Cuadro 1: Fragmento de archivo de datos recopilados

Algoritmo	Grafo	Vertice	Arista	Media
make_max_clique_graph	1NoDirigido.csv	800	3966	13.564
make_max_clique_graph	2NoDirigido.csv	400	3780	8.161
make_max_clique_graph	3NoDirigido.csv	640	3850	11.440
make_max_clique_graph	4NoDirigido.csv	500	2439	4.973
make_max_clique_graph	5NoDirigido.csv	710	2478	5.741
betweenness_centrality	1NoDirigido.csv	800	3966	47.347
betweenness_centrality	2NoDirigido.csv	400	3780	15.314
betweenness_centrality	3NoDirigido.csv	640	3850	30.711
betweenness_centrality	4NoDirigido.csv	500	2439	17.163
betweenness_centrality	5NoDirigido.csv	710	2478	31.423
greedy_color	1NoDirigido.csv	800	3966	2.615
greedy_color	2NoDirigido.csv	400	3780	1.613
greedy_color	3NoDirigido.csv	640	3850	2.203
greedy_color	4NoDirigido.csv	500	2439	1.609
greedy_color	5NoDirigido.csv	710	2478	2.116
Maximal_matching	1NoDirigido.csv	800	3966	0.991
Maximal_matching	2NoDirigido.csv	400	3780	0.787
Maximal_matching	3NoDirigido.csv	640	3850	0.910
Maximal_matching	4NoDirigido.csv	500	2439	0.589
Maximal_matching	5NoDirigido.csv	710	2478	0.685
dfs_tree	1NoDirigido.csv	800	3966	4.932
dfs_tree	2NoDirigido.csv	400	3780	2.897
dfs_tree	3NoDirigido.csv	640	3850	4.368
dfs_tree	4NoDirigido.csv	500	2439	2.912
dfs_tree	5NoDirigido.csv	710	2478	4.186

3. Resultados del Análisis de los datos

Con los datos recopilado de las ejecuciones de los Algoritmos par los cinco grafos se realizó un histograma, como se muestra en la figura 2 y dos diagramas de dispersión, uno que muestra la relación de la media de los tiempos de ejecución de cada algoritmo con la cantidad de vértices y el otro con la cantidad de aristas, como se muestra en la figura 3 y 4 respectivamente. En estas figuras se puede observar que el algoritmo con los tiempos de ejecución más pequeños es el *Maximal matching*, así como que el de mayores tiempos es el *Betweenness centrality*. En sentido general se observa que cuando aumentan la cantidad de nodos y aristas aumenta el tiempo de ejecución de los algoritmos.

```
1 from matplotlib.lines import Line2D
2 import matplotlib.patches as mpatches
3 dr = pd.read_csv( "salid.csv" )
4 print(dr["media"])[10:15]
5
6
7 plt.hist(np.log1p(dr["media"][:5]),bins=5, color="#932525", alpha=1, edgecolor = 'black', linewidth=1)
8 plt.ylabel('Frecuencia de ocurrencia')
9 plt.xlabel('Tiempo de Ejecucion')
10 plt.savefig("Histograma1.eps",bbox_inches='tight')
11 plt.savefig("Histograma1.png",bbox_inches='tight')
12 plt.show()
13
14 plt.hist(np.log1p(dr["media"][5:10]),bins=5, color="#ee9110", alpha=1, edgecolor = 'black', linewidth=1)
15 plt.ylabel('Frecuencia de ocurrencia')
16 plt.xlabel('Tiempo de Ejecucion')
17 plt.savefig("Histograma2.eps",bbox_inches='tight')
18 plt.savefig("Histograma2.png",bbox_inches='tight')
19 plt.show()
20
21 plt.hist(np.log1p(dr["media"][10:15]),bins=5,color="#adcb18", alpha=1, edgecolor = 'black', linewidth=1)
22 plt.ylabel('Frecuencia de ocurrencia')
23 plt.xlabel('Tiempo de Ejecucion')
24 plt.savefig("Histograma3.eps",bbox_inches='tight')
25 plt.savefig("Histograma3.png",bbox_inches='tight')
26 plt.show()
27
28 plt.hist(np.log1p(dr["media"][15:20]),bins=5,color="#129f10", alpha=1, edgecolor = 'black', linewidth=1)
29 plt.ylabel('Frecuencia de ocurrencia')
30 plt.xlabel('Tiempo de Ejecucion')
31 plt.savefig("Histograma4.eps",bbox_inches='tight')
32 plt.savefig("Histograma4.png",bbox_inches='tight')
33 plt.show()
34
35 plt.hist(np.log1p(dr["media"][20:25]),bins=5,color="#093ea8", alpha=1, edgecolor = 'black', linewidth=1)
36 plt.ylabel('Frecuencia de ocurrencia')
37 plt.xlabel('Tiempo de Ejecucion')
38 plt.savefig("Histograma5.eps",bbox_inches='tight')
39 plt.savefig("Histograma5.png",bbox_inches='tight')
40 plt.show()
```

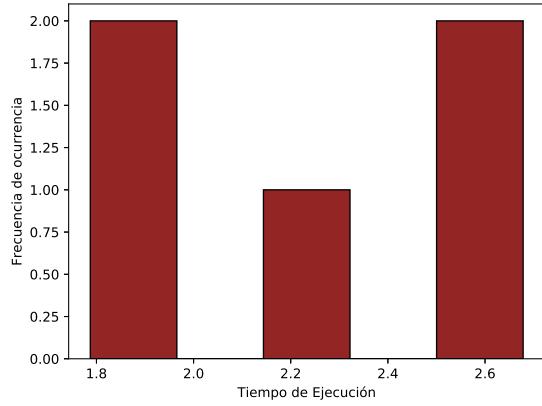
Histograma.py

```

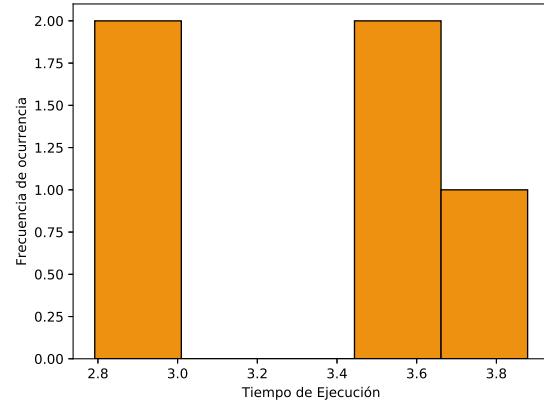
1 lastlinelastline
2 size = (25 * dr["cant_arista"][:5:10] / dr["cant_vertice"][:5:10])
3 color_names = ["#932525", "#129f10", "#093ea8", "#adcb18", "#ee9110"]
4 figure, axes = plt.subplots(figsize=(10, 10))
5 axes.errorbar(dr["media"][:5], dr["cant_vertice"][:5], xerr=(dr["desv"][:5]+1), fmt='D'
   ,color="#932525", alpha=1, label="Make max clique graph")
6
7 axes.errorbar(dr["media"][:5:10], dr["cant_vertice"][:5:10], xerr=(dr["desv"][:5:10]+1) ,
   fmt='s',color="#129f10", alpha=1,label="Betweenness centrality")
8
9 axes.errorbar(dr["media"][10:15], dr["cant_vertice"][10:15], xerr=(dr["desv"]
   [10:15]+1), fmt='8',color="#093ea8", alpha=1,label="Greedy color algorithm")
10
11 axes.errorbar(dr["media"][15:20], dr["cant_vertice"][15:20], xerr=(dr["desv"]
   [15:20]+1), fmt='>',color="#adcb18", alpha=1,label="Maximal matching")
12
13 axes.errorbar(dr["media"][20:25], dr["cant_vertice"][20:25], xerr=(dr["desv"]
   [20:25]+1), fmt='o',color="#ee9110", alpha=1,label="Dfs_tree")
14
15 axes.set_ylabel("Vertices ", fontsize=12, fontfamily="arial", fontweight="bold")
16 axes.set_xlabel("Tiempo de Ejecucion", fontsize=12, fontfamily="arial", fontweight="bold")
17 plt.ylim((min(dr["cant_vertice"]))-30, max(dr["cant_vertice"]) + 30)
18
19 axes.legend()
20 plt.savefig("DiagramVertices.eps",bbox_inches='tight')
21 plt.savefig("DiagramVertices.png",bbox_inches='tight')
22
23 size = (25 * dr["cant_arista"][:5:10] / dr["cant_arista"][:5:10])
24 color_names = ["#932525", "#129f10", "#093ea8", "#adcb18", "#ee9110"]
25 figure, axes = plt.subplots(figsize=(10, 10))
26
27 axes.errorbar(dr["media"][:5], dr["cant_arista"][:5], xerr=(dr["desv"][:5]+1), fmt='D'
   ,color="#932525", alpha=1,label="Make max clique graph")
28
29
30 axes.errorbar(dr["media"][:5:10], dr["cant_arista"][:5:10], xerr=(dr["desv"][:5:10]+1) ,
   fmt='s',color="#129f10", alpha=1,label="Betweenness centrality")
31
32 axes.errorbar(dr["media"][10:15], dr["cant_arista"][10:15], xerr=(dr["desv"][:10:15]+1)
   , fmt='8',color="#093ea8", alpha=1,label="Greedy color algorithm")
33
34 axes.errorbar(dr["media"][15:20], dr["cant_arista"][15:20], xerr=(dr["desv"][:15:20]+1)
   , fmt='>',color="#adcb18", alpha=1,label="Maximal matching")
35
36 axes.errorbar(dr["media"][20:25], dr["cant_arista"][20:25], xerr=(dr["desv"][:20:25]+1)
   , fmt='o',color="#ee9110", alpha=1,label="Dfs_tree")
37
38 axes.set_ylabel("Aritas ", fontsize=12, fontfamily="arial", fontweight="bold")
39 axes.set_xlabel("Tiempo de Ejecucion", fontsize=12, fontfamily="arial", fontweight="bold")
40 plt.ylim((min(dr["cant_arista"]))-30, max(dr["cant_arista"]) + 30)
41
42 axes.legend()
43 plt.savefig("DiagramAristas.eps",bbox_inches='tight')
44 plt.savefig("DiagramAristas.png",bbox_inches='tight')
45
46 axes.errorbar(dr["media"][:5], dr["cant_arista"][:5], xerr=dr["desv"][:5], fmt='+',
   color='k',alpha=1)
47 plt.show()

```

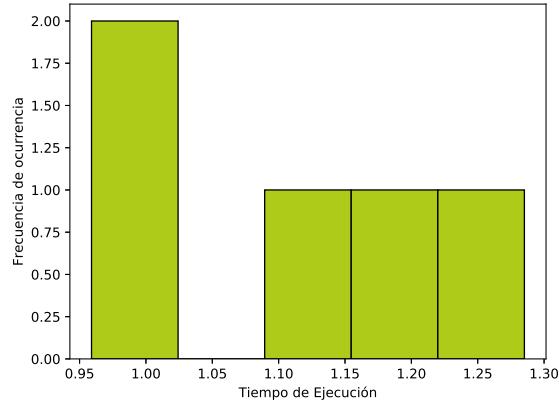
Histograma.py



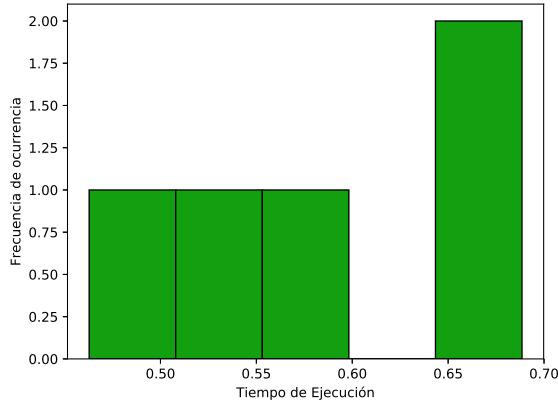
(a) *Make max clique graph*



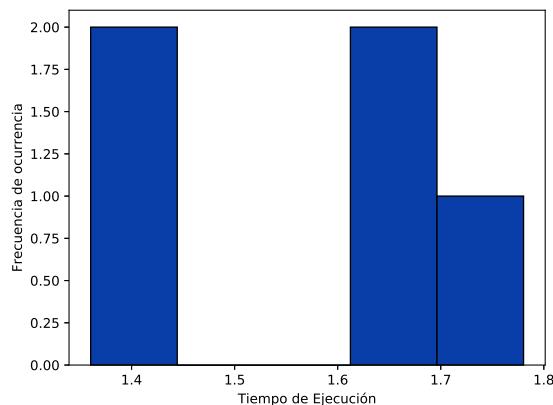
(b) *Betweenness centrality*



(c) *Greedy color algorithm*



(d) *Maximal matching*



(e) *Dfs tree*

Figura 2: Histograma de cada uno de los cinco algoritmos con los cinco grafos generados.

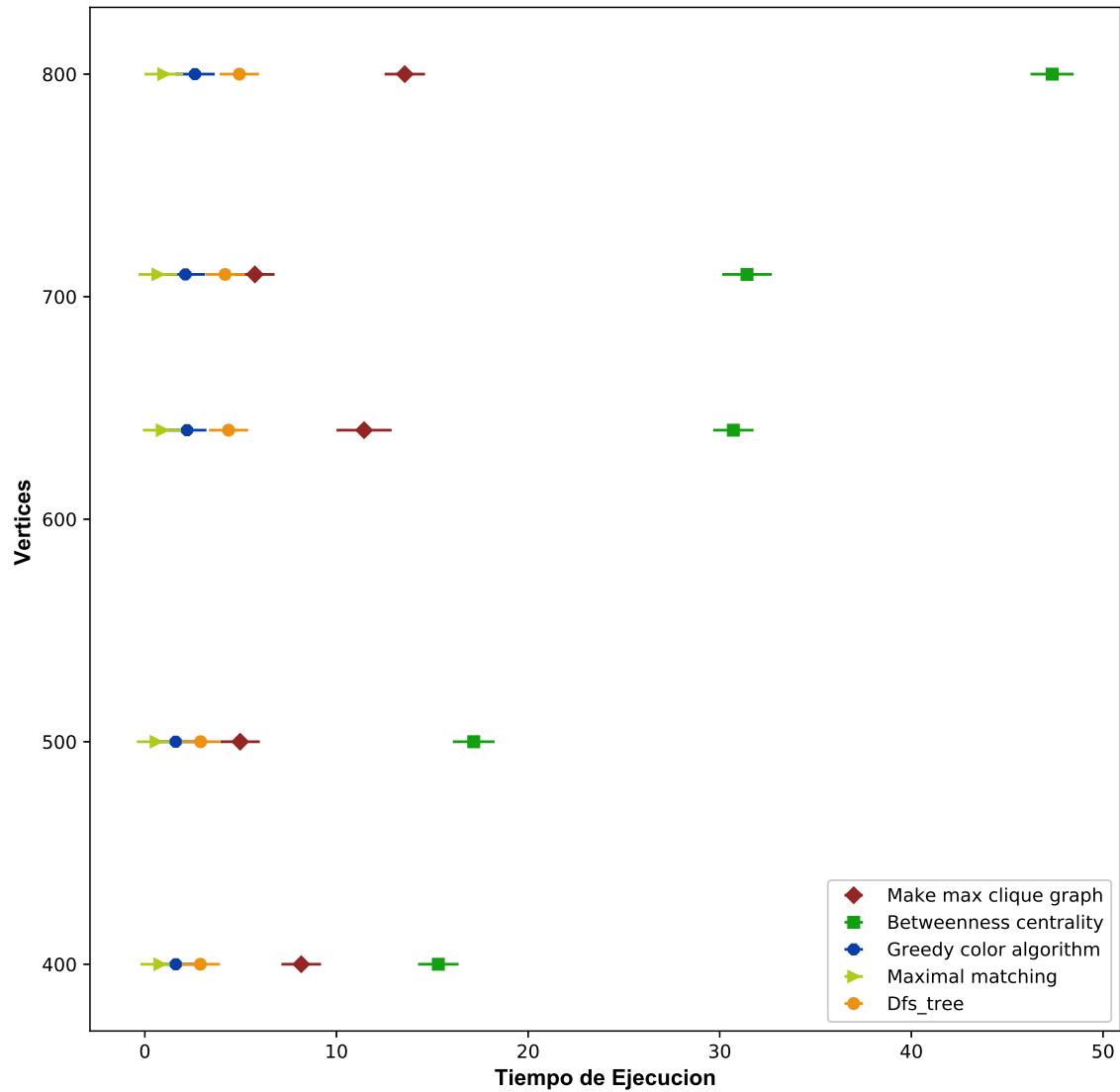


Figura 3: Diagrama de dispersión que muestra la relación entre los tiempos de ejecución(en segundos) y la cantidad de vértices.

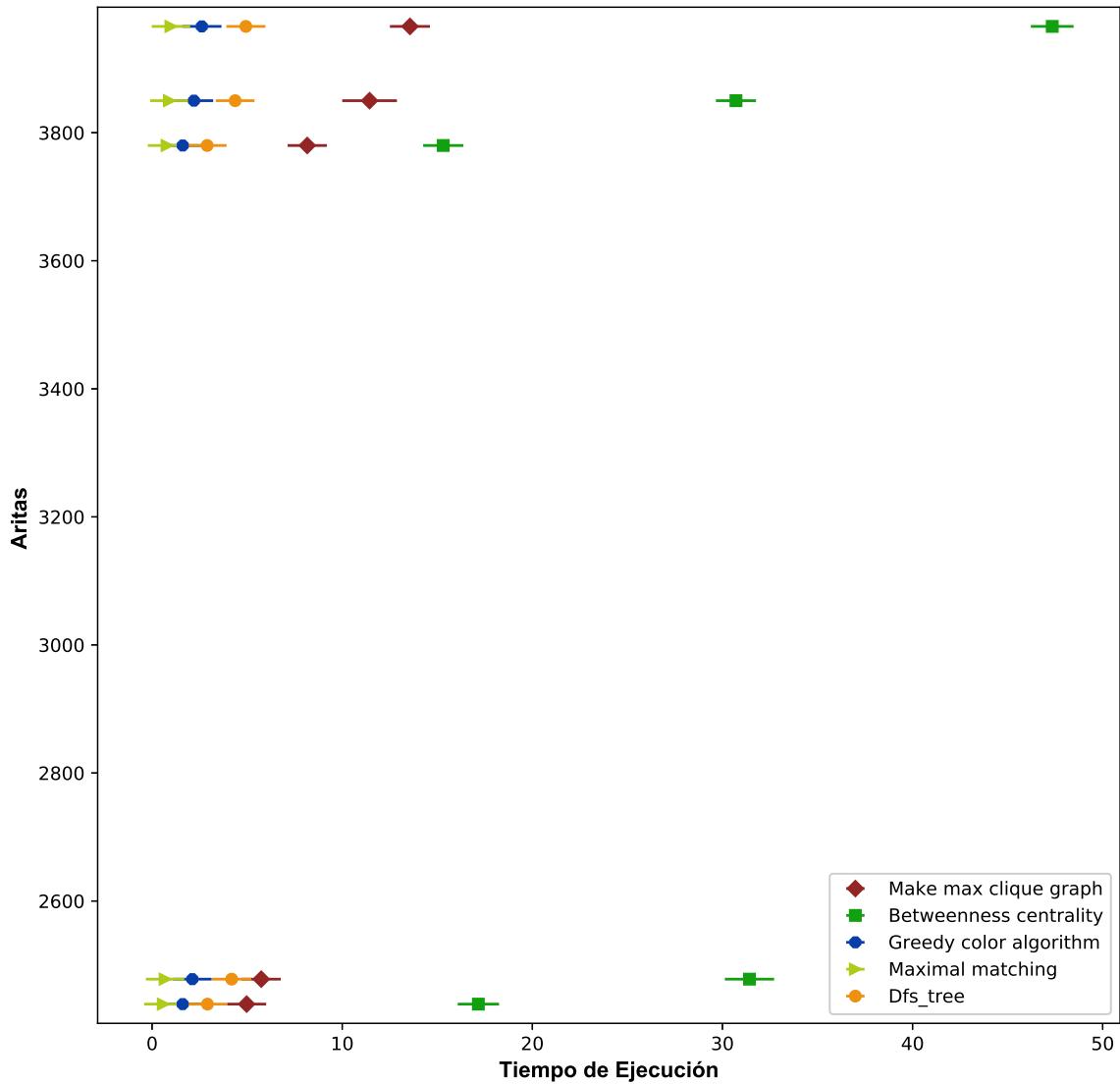
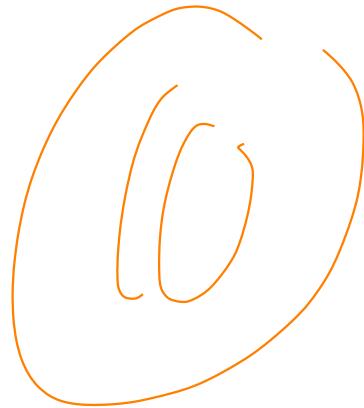


Figura 4: Diagrama de dispersión que muestra la relación entre los tiempos de ejecución(en segundos) y la cantidad de aristas.

Referencias

- [1] Desarrolladores de NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.cliques.make_max_clique_graph.html. Accessed: 18-03-2019.
- [2] Desarrolladores de NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.centrality.betweenness_centrality.html#networkx.algorithms.centrality.betweenness_centrality. Accessed: 18-03-2019.
- [3] Desarrolladores de NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.coloring.greedy_color.html. Accessed: 18-03-2019.
- [4] Desarrolladores de NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.matching.maximal_matching.html. Accessed: 18-03-2019.
- [5] Desarrolladores de NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.traversal.depth_first_search.dfs_tree.html#networkx.algorithms.traversal.depth_first_search.dfs_tree. Accessed: 18-03-2019.



Tarea 4

5271

2 de abril de 2019

1. Algoritmos generadores de grafos ~~X~~

Gracias a la versatilidad de los grafos como modelo de representación de datos, los procesos aleatorios de generación de grafos también son relevantes en aplicaciones que van desde la física, biología a la sociología [8].

Para realizar la tarea se seleccionaron tres algoritmos generadores de grafos de la biblioteca de networkx, con el objetivo de crear los grafos con pesos con distribución normal (como se muestra en la figura 1 de la página 2) a los que se le aplicaran los algoritmos de flujo máximo para realizar los experimentos requeridos. Los tres algoritmos escogidos son de generación aleatorias.

Los algoritmos escogidos fueron los siguientes:

- *Erdős-Renyi graph* (en el modelo $G(n, p)$, el grafo se construye conectando los n vértices al azar. Cada arista se incluye en el grafo con probabilidad p independiente de cualquier otro borde) [3].
- *Fast gnp random graph* (este algoritmo recibe como parámetros n números de vértices y probabilidad p de ocurrencia de aristas, el mismo devuelve un grafo aleatorio) [4].
- *Binomial graph* (este algoritmo recibe como parámetros n números de vértices y probabilidad p de ocurrencia de aristas, el mismo devuelve un grafo aleatorio, para grafos dispersos para valores pequeños de p , *Fast gnp random graph* es un algoritmo más rápido) [5].

En la figura 2 de la 3 se muestra ejemplos de grafos generados por los algoritmos antes mencionados.

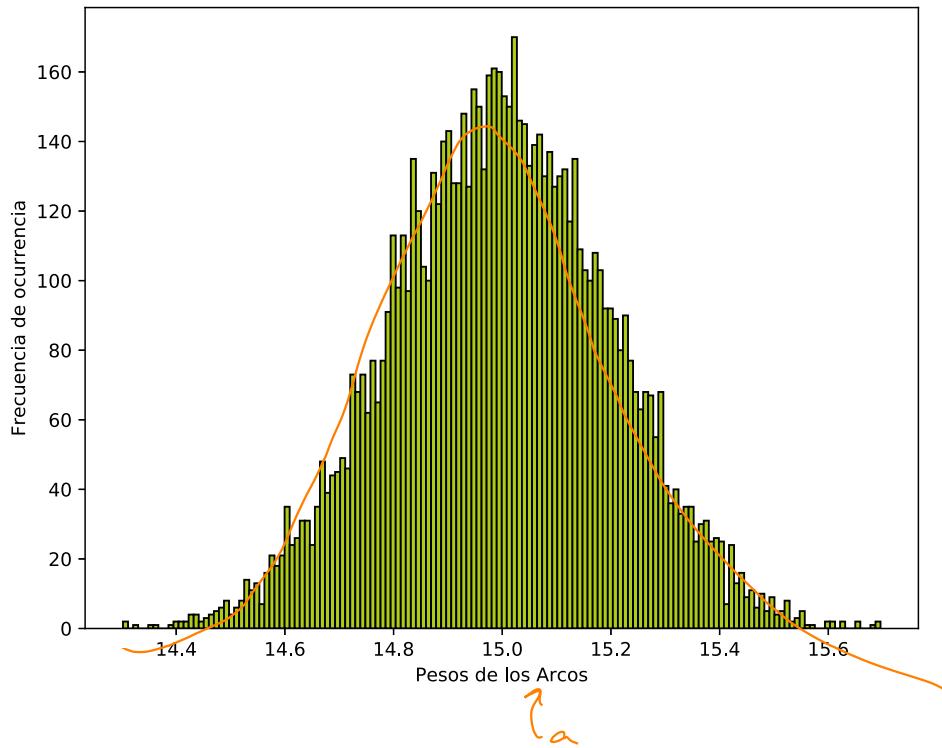


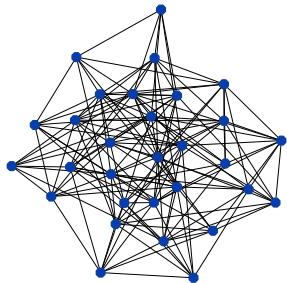
Figura 1: Histograma de distribución de los pesos.

2. Algoritmos de flujo máximo.

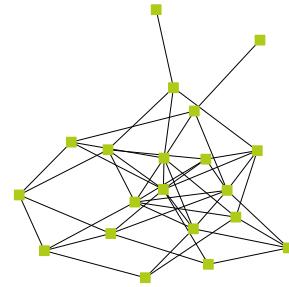
En los problemas de flujo en redes, las aristas representan vías por las que puede circular elementos: datos, agua, corriente eléctrica, entre otras. Los pesos de las aristas representan la capacidad máxima de una vía: velocidad de una conexión, volumen máximo de agua, voltaje de una línea eléctrica, entre otras; aunque es posible que la cantidad real de flujo sea menor.

El problema del flujo máximo consiste en lo siguiente: dado un grafo con pesos, $G = (V, A, W)$, que representa las capacidades máximas de los canales, un vértice fuente f y otro sumidero s en V , encontrar la cantidad máxima de flujo que puede circular desde f hasta s .

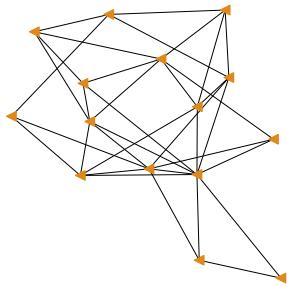
En la librería de networkx encontramos varios algoritmos con los que podemos atacar los problemas de flujo máximo, para la realización de esta tarea se escogerán tres algoritmos de dicha librería.



(a) *Binomial graph*, con 30 vértices



(b) *Erdos renyi graph*, con 20 vértices



(c) *Fast gnp random graph*, con 15 vértices

Figura 2: Ejemplo de grafos generados con los algoritmos seleccionados.

Los algoritmos escogidos fueron los siguientes:

- *Shortest augmenting path* ✓ Es uno de los enfoques más clásicos para la máxima coincidencia y los problemas de flujo máximo. Sorprendentemente, aunque esta idea es una de las técnicas más básicas, está lejos de ser completamente entendida. Es más fácil hablar de ello introduciendo el problema de emparejamiento bipartito en línea [1]. Este algoritmo encuentra el flujo máximo de un solo producto utilizando la ruta de aumento más corto y devuelve la red residual resultante después de calcular el flujo máximo. ✗
- *Maximum flow* ✗ Encuentra la ruta por la cual pasa la máxima cantidad de flujo, recibe como parámetros un grafo G , una fuente f , un sumidero s y además una capacidad que de no tenerla, se considera que el borde tiene una capacidad infinita. Se puede aplicar en grafos tanto dirigidos como no dirigidos. ✗ [6].

- Preflow push X encuentra un flujo máximo de un solo producto utilizando el algoritmo de empuje previo al flujo de la etiqueta más alta. Esta función devuelve la red residual resultante después de calcular el flujo máximo. Este algoritmo tiene un tiempo de ejecución de $O(n^2\sqrt{m})$ para n vértices y m aristas. X [7].

3. Generación de datos.

Con el objetivo de realizar las mediciones de los tiempos de ejecución de los algoritmos de flujo máximo seleccionados se desarrolló el siguiente código.

En primer lugar, se crea una función (AlgoritmoFM()) que recibe como parámetros el algoritmo de flujo máximo (algoritmoF) que se va a utilizar, el grafo al que se le aplicara el algoritmo (Graf), la fuente (fuente) y sumidero (sumidero). Al llamar esta función se genera con cada ejecución los datos que son guardados en un *data frame* del cual se muestra un fragmento en el cuadro 1 de la página 5.

```

1 def Algoritmo_FM(algoritmoF ,Graf, funte , sumidero ,):
2     tiempos_ejecucion = []
3     for medicion in range(1, mediciones + 1):
4         t_inicio = dt.datetime.now()
5         obj = algoritmos.flujo [algoritmoF](Graf, funte , sumidero ,
6                                         capacity="capacity")
6         t_fin = dt.datetime.now()
7         tiempo_consumido_segundos = (t_fin - t_inicio).
8         total_seconds()
9         tiempos_ejecucion.append(tiempo_consumido_segundos)
10        media = stats.mean(tiempos_ejecucion)
11
12        t_csv["grafo"].append("vertices" + str(inst_g_n) + "aristas" +
13                               str(aristas))
14        t_csv["generador_grafo"].append(generador_grafo)
15        t_csv["vertices"].append(inst_g_n)
16        t_csv["densidad"].append(nx.density(Grafo))
17        t_csv["aristas"].append(aristas)
18        t_csv["f"].append(f)
19        t_csv["s"].append(s)
20        t_csv["algoritmo_fm"].append(algoritmo)
21        t_csv["media"].append(round(media, 5))
22        t_csv["mediana"].append(round(stats.median(tiempos_ejecucion),
23                                  5))
23        t_csv["varianza"].append(round(stats.pvariance(
24            tiempos_ejecucion, mu=media), 5))
25        t_csv["desv"].append(round(stats.pstdev(tiempos_ejecucion, mu=
26                                  media), 5))
27
28    return t_csv

```

Generar_datos.py

Cuadro 1: Fragmento de *data frame* generado.

grafo	generador	algoritmo_fm	vertices	densidad	aristas	f	s	mediana	varianza	desv
v256a8140	fast_gnp	shortest_a_path	256	0.24939	8140	33	101	0.06247	0.00004	0.00618
v256a8141	fast_gnp	edmonds_karp	256	0.24939	8140	33	101	0.04692	0.00001	0.00318
v256a8142	fast_gnp	preflow_push	256	0.24939	8140	33	101	0.08225	0.00025	0.01596
v256a8143	fast_gnp	shortest_a_path	256	0.24939	8140	33	101	0.06251	0.00004	0.00638
v256a8144	fast_gnp	edmonds_karp	256	0.24939	8140	33	101	0.05291	0.00007	0.00834
v256a8145	fast_gnp	preflow_push	256	0.24939	8140	33	101	0.08253	0.00017	0.01296
v256a8146	fast_gnp	shortest_a_path	256	0.24939	8140	33	101	0.06903	0.00020	0.01393
v256a8147	fast_gnp	edmonds_karp	256	0.24939	8140	33	101	0.05296	0.00003	0.00566

En este otro fragmento del código es donde se generan los grafos y se llama la función *(AlgoritmoFM())*.

```

1 numero_intancias = 10
2 mediciones = 5
3 archivo_CSV = "Datost4.csv"
4 control_iteraciones = 0
5
6 generadores_grafos = {
7 "fast_gnp_random_graph": nx.fast_gnp_random_graph ,
8 "binomial_graph": nx.binomial_graph ,
9 "erdos_renyi_graph": nx.erdos_renyi_graph
10 }
11
12 algoritmos_flujo = {
13 "shortest_augmenting_path": shortest_augmenting_path ,
14 "maximum_flow": maximum_flow ,
15 "preflow_push": preflow_push
16 }
17
18 t_csv = {
19 "grafo": [] , "generador_grafo": [] ,
20 "algoritmo_fm": [] , "vertices": [] ,
21 "densidad": [] , "aristas": [] ,
22 "f": [] , "s": [] , "media": [] ,
23 "mediana": [] , "varianza": [] ,
24 "desv": []
25 }
from np.random import randint
26 for generador_grafo in generadores_grafos:
27     for inst_g_n in [round(pow(2, value + 1))
28                      for value in
29                      range(7, 11 )]:
30
31         for grafo in range(1, numero_intancias + 1):
32             f = np.random.randint(1, high=(inst_g_n - 1), dtype="int")
33             s = np.random.randint(1, high=(inst_g_n - 1), dtype="int")
34
35             while s == f:
36

```

```

37      f = np.random.randint(1, high=(inst_g_n - 1), dtype
38      ="int")
39      s = np.random.randint(1, high=(inst_g_n - 1), dtype
40      ="int")
41      Grafo = generadores_grafos[generador_grafo](inst_g_n,
42      0.25, seed=None)
43      aristas = Grafo.number_of_edges()
44      p_n_distribuidos = np.random.normal(15, 0.2, aristas)
45      incremento = 0
46
47      for (u, v) in Grafo.edges():
48          Grafo.edges[u, v]["capacity"] = p_n_distribuidos[
49          incremento]
50          incremento += 1
51          pesos = []
52          for (u, v) in Grafo.edges():
53              t = Grafo.edges[u, v]["capacity"]
54              pesos.append(t)
55          plt.figure(figsize=(8, 6))
56          n = plt.hist(pesos, bins=70, color="#932525", alpha=1,
57          edgecolor = 'black', linewidth=1)
58          plt.xlabel('Pesos de los Arcos')
59          plt.ylabel('Frecuencia de ocurrencia')
60
61          plt.savefig("histografo.png")
62          plt.savefig("histografo.eps")
63          for instancia_grafo in range(1, 6):
64              for algoritmo in algoritmos_flujo:
65                  d= Algoritmo_FM(algoritmo, Grafo, f, s,
66                  control_iteraciones)
67                  control_iteraciones=d
68
69          ds = pd.DataFrame(t_csv)
70          ds.to_csv(archivo_CSV, encoding="utf-8", index=None)

```

Generar_datos.py

4. Resultados del Análisis de los datos

Con los datos recopilado de las ejecuciones de los tres algoritmos de flujo máximo con los ciento veinte grafos y sus cinco combinaciones de fuentes y sumidero, se realizó una serie de diagramas y pruebas estadísticas. A continuación, se muestra un el fragmento de código donde se lleva a cabo lo antes mencionado.

```

1 df = pd.read_csv("Datost4.csv", index_col=None, usecols=[1,2,3,4,9],
2                 dtype={'generador_grafo': 'category',
3
3                 'algoritmo_fm': 'category', 'vertices': 'category', 'densidad':np.float64, 'mediana': np.float64} )
4 modelo = ols('mediana_log ~ generador_grafo+algoritmo_fm+vertices+
5                 densidad+generador_grafo*algoritmo_fm+algoritmo_fm*vertices+
6                 vertices*densidad+generador_grafo*vertices+generador_grafo*
7                 densidad+algoritmo_fm*densidad', data=df). fit()

```

```

4 print(modelo.summary())
5 modelo_csv = open("Anova.Mult.csv", 'w')
6 aov_table = sm.stats.anova_lm(modelo, typ=2)
7 df1=pd.DataFrame(aov_table)
8 df1.to_csv("modelo.csv")
9 for column in range(0, df["densidad"].count()):
10     pass
11     if df.iat[column, 3] >=0.2061 and df.iat[column, 3] <
12         0.20854:
13         df.iat[column, 3] = 1
14     elif df.iat[column, 3] >=0.20854 and df.iat[column, 3] <
15         0.21098:
16         df.iat[column, 3] = 2
17     else:
18         df.iat[column, 3] = 3
19 print(df["densidad"])
20 df['densidad'].replace({1:"baja", 2: 'media', 3: 'alta'}, inplace=
21 True)
22 print(df["densidad"])
23 logX = np.log1p(df['mediana'])
24 df = df.assign(mediana_log=logX.values)
25 df.drop(['mediana'], axis= 1, inplace= True)
26 factores=["vertices","generador_grafo","densidad","algoritmo_fm"]
27 plt.figure(figsize=(8, 6))
28 for i in factores:
29     print(rp.summary_cont(df['mediana_log'].groupby(df[i])))
30
31 anova = pg.anova(dv='mediana_log', between=i, data=df,
32 detailed=True, )
33 pg.export_table(anova, ("ANOVAs"+i+".csv"))
34
35 ax=sns.boxplot(x=df["mediana_log"], y=df[i], data=df, palette="cubehelix")
36
37 plt.savefig("boxplot_" + i + ".eps", bbox_inches='tight')
38 tukey = pairwise_tukeyhsd(endog = df["mediana_log"], groups= df[i], alpha=0.05)
39
40 tukey.plot_simultaneous(xlabel='Tiempo', ylabel=i)
41 plt.vlines(x=49.57,ymin=-0.5,ymax=4.5, color="red")
42 plt.savefig("simultaneous_tukey" + i + ".eps", bbox_inches='tight')
43
44 print(tukey.summary())
45 t_csv = open("Tukey"+i+".csv", 'w')
46 with t_csv:
47     writer = csv.writer(t_csv)
48     writer.writerows(tukey.summary())
49     plt.show()

```

Analisis_datos.py

4.1. Análisis de varianza (ANOVA)

El análisis de varianza (ANOVA) es la técnica central en el análisis de datos experimentales. La idea general de esta técnica es separar la variación total en las partes con las que contribuye cada fuente de variación en el experimento. En el caso de los diseños completamente al azar se separan la variabilidad debida a los tratamientos y la debida al error. Cuando la primera predomina sobre la segunda, es cuando se concluye que las medias son diferentes. Cuando los tratamientos no dominan contribuyen igual o menos que el error, por lo que se concluye que las medias son iguales [2].

Para analizar si los diferentes factores (algoritmo generador de grafo, algoritmo de flujo máximo, numero de vértices y densidad del grafo) influían en la variable dependiente *tiempo de ejecución* se realizó un ANOVA de un factor para cada caso.

4.1.1. Influencia del algoritmo generador de grafos en el tiempo de ejecución.

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 2: ANOVA, relación del algoritmo generador con el *tiempo de ejecución*

Factor	SS	DF	MS	F	p-unc	np2
generador_grafo	0.280	2	0.140	0.354	0.702	0
Within	712.085	1797	0.396	X	X	X

En el cuadro 2 se muestra que no existen diferencias entre las medianas de los grupos de factores ya que el *p - unc* es mayor que 0,05 por lo que se acepta la hipótesis de que el tipo de generador no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 3 de la página 9.

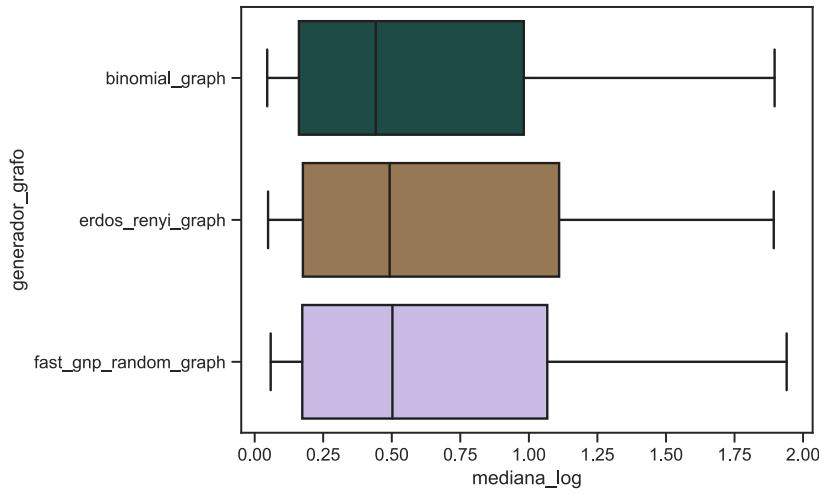


Figura 3: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con los algoritmos generadores de grafos.

4.1.2. Influencia del algoritmo de flujo máximo en el tiempo de ejecución.

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 3: ANOVA, relación del algoritmo de flujo máximo con el *tiempo de ejecución*

Factor	SS	DF	MS	F	p-unc	np2
algoritmo_fm	1.499	2	0.749	1.895	0.151	0.002
Within	710.867	1797	0.396	-	-	-

En el cuadro 3 se muestra que no existen diferencia entre las medianas de los grupos de factores ya que el *p - unc* es mayor que 0,05 por lo que se acepta la hipótesis de que el tipo de algoritmo no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 4 de la página 10.

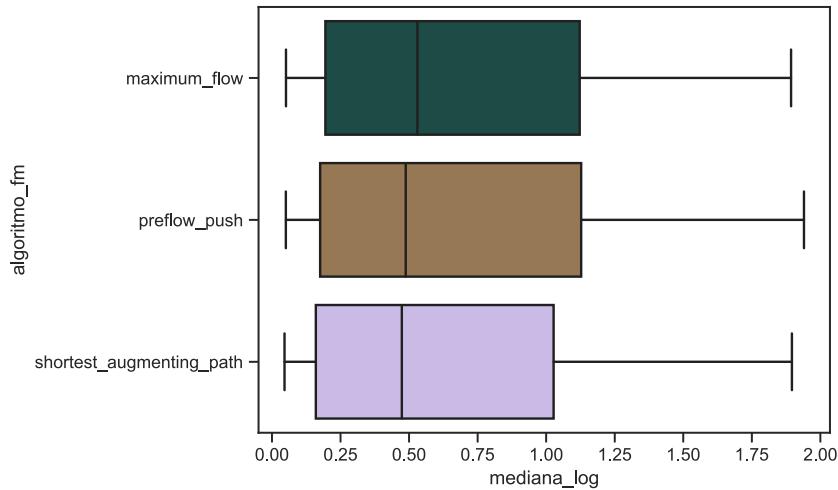


Figura 4: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con los algoritmos de flujo máximo.

4.1.3. Influencia del número de vértices en el tiempo de ejecución.

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 4: ANOVA, relación del número de vértices con el *tiempo de ejecución*

Factor	SS	DF	MS	F	p-unc	np2
vértices	706.976	3	235.659	78536.187	0	0.992
Within	5.389	1796	0.003	-	-	-

En el cuadro 4 se muestra que existen grandes diferencias entre las medianas de los grupos de factores ya que el *p-unc* es menor que 0,05 por lo que se rechaza la hipótesis de que la cantidad de vértices no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 5 de la página 11. Por tal motivo se realiza la prueba de Tukey mostrada en la figura 6 de la página 12 que evidencia las diferencias entre las medianas de los factores.

En el cuadro 5 se puede observar que en todos los casos se rechaza la hipótesis. En la figura 6 de la página 12 muestra claramente este hecho.

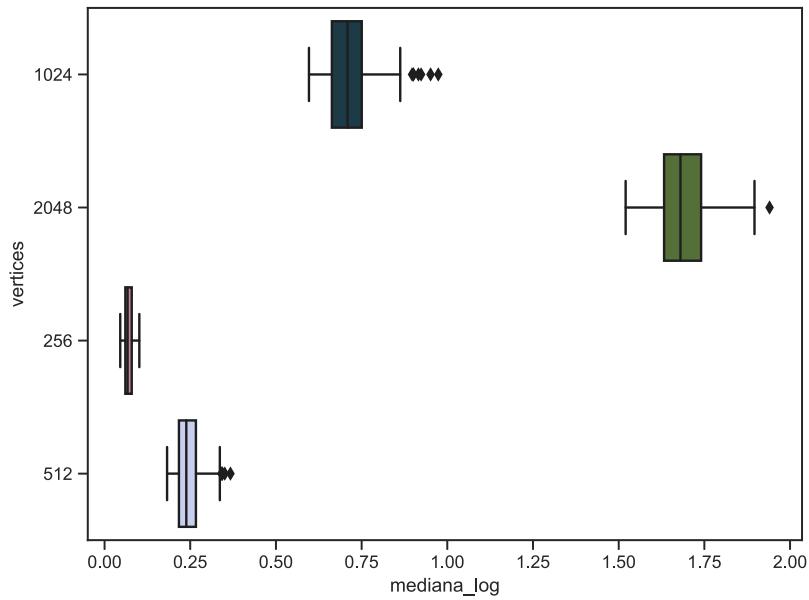


Figura 5: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con el número de vértices.

Cuadro 5: Tukey, influencia del número de vértices en el tiempo de ejecución.

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i> ↙
1024	2048	0.97	0.9606	0.9794	<i>True</i>
1024	256	-0.6445	-0.6539	-0.6352	<i>True</i>
1024	512	-0.4689	-0.4782	-0.4595	<i>True</i>
2048	256	-1.6146	-1.624	-1.6052	<i>True</i>
2048	512	-1.4389	-1.4483	-1.4295	<i>True</i>
256	512	0.1757	0.1663	0.1851	<i>True</i>

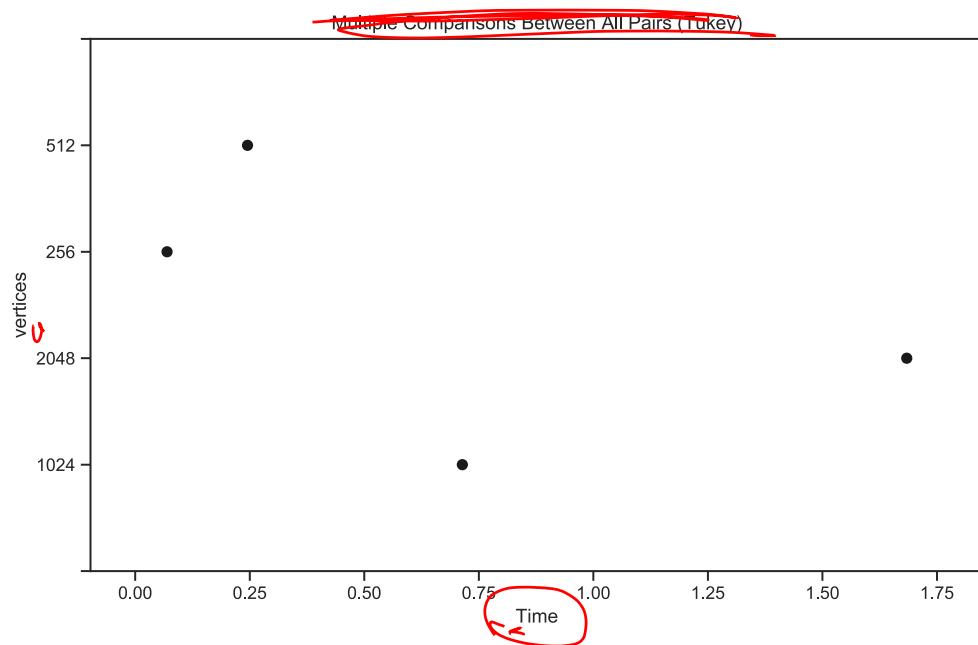


Figura 6: Diagrama simultaneo que relaciona los tiempos de ejecución con los grupos del factor número de vértices.

4.1.4. Influencia de la densidad de los grafos en el tiempo de ejecución.

En el caso del factor densidad se hizo una categorización por rangos para hacer cómoda la visualización de su relación con la variable dependiente estos rangos se obtuvieron a través de los contenedores que devolvió el histograma que se muestra en la figura 7 de la página 13.

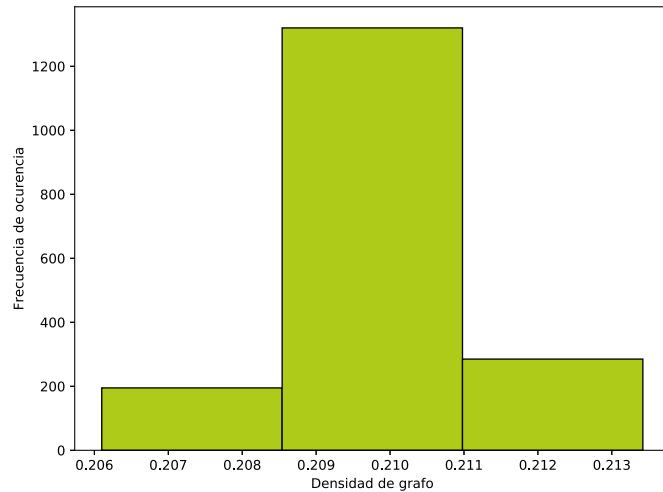


Figura 7: Histograma de densidad de los grafos.

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 6: ANOVA, influencia de la densidad de los grafos en el tiempo de ejecución.

Factor	SS	DF	MS	F	p-unc	np2
densidad	170.777	2	85.388	283.320	0.000	0.24
Within	541.589	1797	0.301	-	-	-

En el cuadro 7 se muestra que como en el cuadro 4 existen grandes diferencias entre las medianas de los grupos de factores ya que el *p-unc* es menor que 0,05 por lo que se rechaza la hipótesis de que la densidad de los grafos no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 8 de la página 14.

Por tal motivo se realiza la prueba de Tukey mostrara las diferencias entre las medianas de los factores, lo que se evidencia en el cuadro 7 de la página 14.

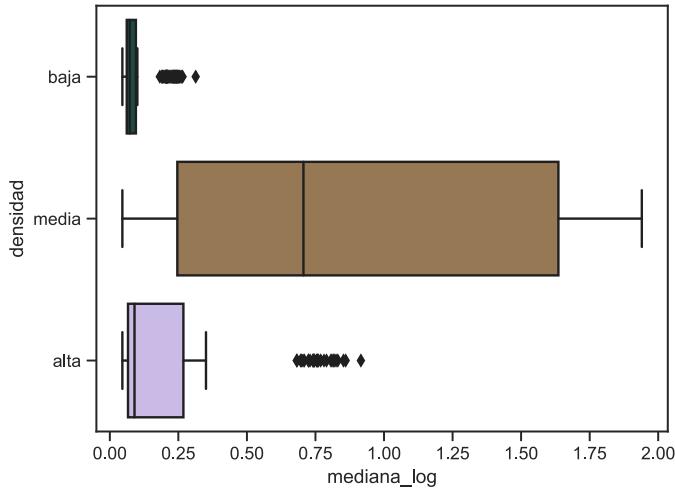


Figura 8: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con la densidad de los grafos.

Cuadro 7: Tukey,influencia de la densidad de los grafos en el tiempo de ejecución.

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	-0.1085	-0.2281	0.0112	<i>False</i>
alta	media	0.6497	0.5656	0.7338	<i>True</i>
baja	media	0.7582	0.6594	0.8569	<i>True</i>

En el cuadro 7 se puede observar que en dos de los tres casos se rechaza la hipótesis y en el pareo de densidad alta y baja no existen diferencias estadísticas en cuanto a la mediana. En la figura 9 de la página 15 muestra claramente este hecho.

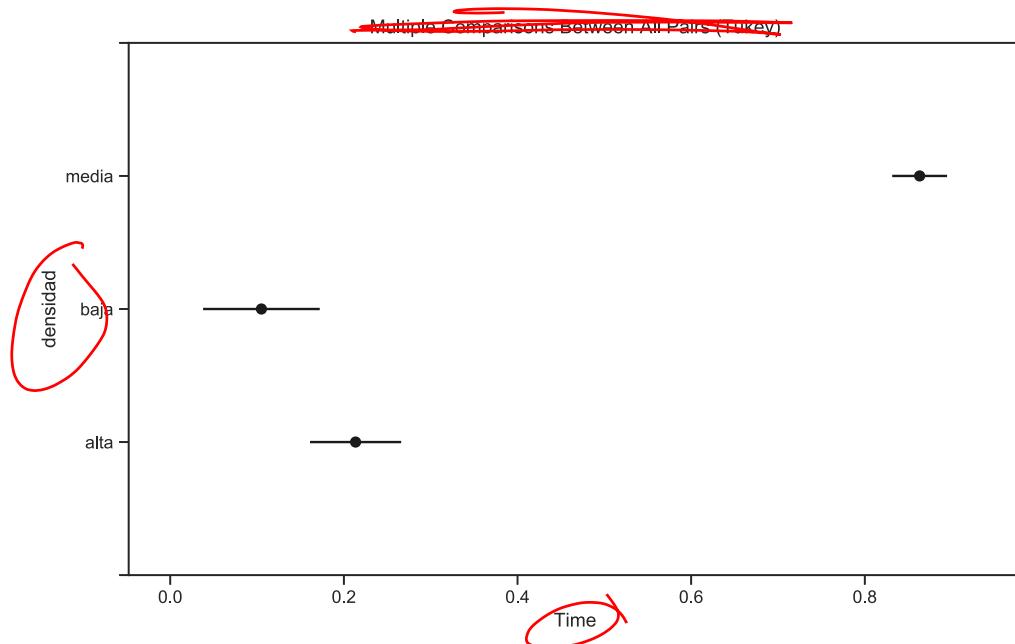


Figura 9: Diagrama simultaneo que relaciona los tiempos de ejecución con los grupos del factor densidad de los grafos.

4.1.5. Influencia de los cuatro factores (algoritmo generador de grafo, algoritmo de flujo máximo, numero de vértices y densidad del grafo) en el tiempo de ejecución ~~X~~

Para analizar este caso se realizó ANOVA multifactorial dando como resultado el cuadro 8 de la página 16.

Cuadro 8: ANOVA multifactor, influencia de los cuatro factores en el tiempo de ejecución.

	sum_sq	df	F	PR(>F)
generador_grafo	0.0291	2	33.0499	0.0000
algoritmo_fm	-0.0001	2	-0.1439	1.0000
vertices	46.4767	3	35163.6974	0.0000
densidad	0.0000	2	0.0000	0.9999
generador_grafo:algoritmo_fm	0.0008	4	0.4714	0.7567
algoritmo_fm:vertices	0.0001	6	0.0248	0.8743
vertices:densidad	0.0001	6	0.0425	0.9584
generador_grafo:vertices	0.0651	6	24.6313	0.0000
generador_grafo:densidad	0.0000	4	0.0181	0.8930
algoritmo_fm:densidad	0.0001	4	0.0425	0.9584
Residual	0.7798	1770		

En el cuadro 8 se puede observar que los factores que más influyen en el tiempo de ejecución son el número de vértices y la densidad de los grafos, que además existe una relación entre el número de nodos y la densidad de los grafos.

Referencias

- [1] Bartłomiej Bosek, Dariusz Leniowski, Piotr Sankowski, and Anna Zych-Pawlewicz. Shortest augmenting paths for online matchings on trees. *Theory of Computing Systems*, 62(2):337–348, Feb 2018.
- [2] Humberto Gutiérrez and Román de la Vara. *Análisis y diseño de experimentos*. The McGraw-Hill Companies, Inc., segunda edición edition, 2008. 60–74.
- [3] Desarrolladores  NetworkX. https://networkx.github.io/documentation/stable/reference/generated/networkx.generators.random_graphs.erdos_renyi_graph.html#networkx.generators.random_graphs.erdos_renyi_graph. Accessed: 01-04-2019.
- [4] Desarrolladores  NetworkX. https://networkx.github.io/documentation/stable/reference/generated/networkx.generators.random_graphs.fast_gnp_random_graph.html#networkx.generators.random_graphs.fast_gnp_random_graph. Accessed: 01-04-2019.
- [5] Desarrolladores  NetworkX. https://networkx.github.io/documentation/stable/reference/generated/networkx.generators.random_graphs.binomial_graph.html#networkx.generators.random_graphs.binomial_graph. Accessed: 01-04-2019.
- [6] Desarrolladores  NetworkX. https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.flow.maximum_flow.html#networkx.algorithms.flow.maximum_flow. Accessed: 01-04-2019.
- [7] Desarrolladores  NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.coloring.greedy_color.html. Accessed: 18-03-2019.
- [8] Sadegh Nobari, Xuesong Lu, Panagiotis Karras, and Stéphane Bressan. Fast random graph generation. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 331–342, New York, NY, USA, 2011. ACM.

Tarea 4

5271

26 de mayo de 2019

1. Algoritmos generadores de grafos

Gracias a la versatilidad de los grafos como modelo de representación de datos, los procesos aleatorios de generación de grafos también son relevantes en aplicaciones que van desde la física, biología a la sociología [8].

Para realizar la tarea se seleccionaron tres algoritmos generadores de grafos de la biblioteca de *networkx*, con el objetivo de crear los grafos con pesos con distribución normal (como se muestra en la figura 1 de la página 2) a los que se le aplicarán los algoritmos de flujo máximo para realizar los experimentos requeridos. Los tres algoritmos escogidos son de generación aleatorias.

Los algoritmos escogidos fueron los siguientes:

- *Erdős–Rényi graph*, en el modelo $G(n, p)$, el grafo se construye conectando los n vértices al azar. Cada arista se incluye en el grafo con probabilidad p independiente de cualquier otro borde [2].
- *Fast gnp random graph*, este algoritmo recibe como parámetros n números de vértices y probabilidad p de ocurrencia de aristas, el mismo devuelve un grafo aleatorio [3].
- *Binomial graph*, este algoritmo recibe como parámetros n números de vértices y probabilidad p de ocurrencia de aristas, el mismo devuelve un grafo aleatorio, para grafos dispersos (para valores pequeños de p), *Fast gnp random graph* es un algoritmo más rápido [4].

En la figura 2 de la 3 se muestra ejemplos de grafos generados por los algoritmos antes mencionados.

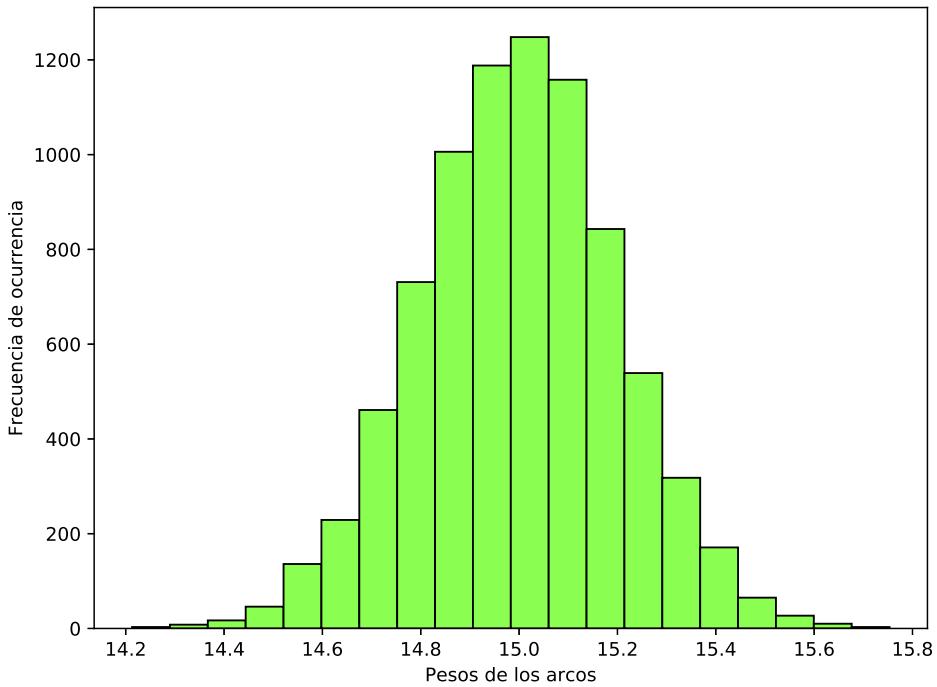


Figura 1: Histograma de distribución de los pesos.

2. Algoritmos de flujo máximo.

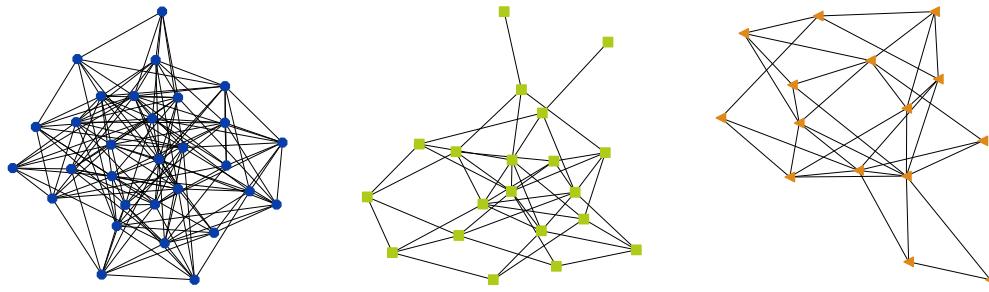
En los problemas de flujo en redes, las aristas representan vías por las que puede circular elementos: datos, agua, corriente eléctrica, entre otras. Los pesos de las aristas representan la capacidad máxima de una vía: velocidad de una conexión, volumen máximo de agua, voltaje de una línea eléctrica, entre otras; aunque es posible que la cantidad real de flujo sea menor.

El problema del flujo máximo consiste en lo siguiente: dado un grafo con pesos, $G = (V, A, W)$, que representa las capacidades máximas de los canales, un vértice fuente f y otro sumidero s en V , encontrar la cantidad máxima de flujo que puede circular desde f hasta s .

En la biblioteca de *networkx* encontramos varios algoritmos con los que podemos atacar los problemas de flujo máximo, para la realización de esta tarea se escogerán tres algoritmos de dicha biblioteca.

Los algoritmos escogidos fueron los siguientes:

- *Shortest augmenting path*, es uno de los enfoques más clásicos para la máxima coincidencia y los problemas de flujo máximo. Sorprendentemente, aunque esta idea es una de las técnicas más básicas, está lejos de ser completamente entendida. Es más fácil hablar de ello introduciendo el problema de emparejamiento bipartito en línea [1]. Este algoritmo encuentra el flujo máximo de un solo producto utilizando la ruta de aumento más corto y devuelve la red residual resultante después de calcular el flujo máximo.



(a) *Binomial graph*, con 30 vértices (b) *Erdős-Rényi graph*, con 20 (c) *Fast $G_{n,p}$ random graph*, con 15 vértices

Figura 2: Ejemplo de grafos generados con los algoritmos seleccionados.

- *Maximum flow*, encuentra la ruta por la cual pasa la máxima cantidad de flujo, recibe como parámetros un grafo G , una fuente f , un sumidero s y además una capacidad que de no tenerla, se considera que el borde tiene una capacidad infinita. Se puede aplicar en grafos tanto dirigidos como no dirigidos. [5].
- *Preflow push*, encuentra un flujo máximo de un solo producto utilizando el algoritmo de empuje previo al flujo de la etiqueta más alta. Esta función devuelve la red residual resultante después de calcular el flujo máximo. Este algoritmo tiene un tiempo de ejecución de $O(n^2\sqrt{m})$ para n vértices y m aristas [7].

3. Generación de datos.

Con el objetivo de realizar las mediciones de los tiempos de ejecución de los algoritmos de flujo máximo seleccionados se desarrolló el siguiente código.

En primer lugar, se crea una función (`Algoritmo_FM()`) que recibe como parámetros el algoritmo de flujo máximo (`algoritmoF`) que se va a utilizar, el grafo al que se le aplicara el algoritmo (`Graf`), la fuente (`fuente`) y sumidero (`sumidero`). Al llamar esta función se genera con cada ejecución los datos que son guardados en un *data frame* del cual se muestra un fragmento en el cuadro 1 de la página 4.

```

1 def Algoritmo_FM(algoritmoF ,Graf, funte , sumidero ,):
2     tiempos_ejecucion = []
3     for medicion in range(1, mediciones + 1):
4         t_inicio = dt.datetime.now()
5         obj = algoritmos_flujo[algoritmoF](Graf, funte , sumidero , capacity="capacity")
6         t_fin = dt.datetime.now()
7
8         tiempo_consumido_segundos = (t_fin - t_inicio).total_seconds()
9         tiempos_ejecucion.append(tiempo_consumido_segundos)
10    media = stats.mean(tiempos_ejecucion)
11
12    t_csv["grafo"].append("vertices" + str(inst_g_n) + "aristas" + str(aristas))
13    t_csv["generador_grafo"].append(generador_grafo)
14    t_csv["vertices"].append(inst_g_n)
15    t_csv["densidad"].append(nx.density(Grafo))
16    t_csv["aristas"].append(aristas)
17    t_csv["f"].append(f)

```

```

18     t_csv[ "s" ].append( s )
19     t_csv[ "algoritmo_fm" ].append( algoritmo )
20     t_csv[ "media" ].append( round( media , 5 ) )
21     t_csv[ "mediana" ].append( round( stats . median( tiempos_ejecucion ) , 5 ) )
22     t_csv[ "varianza" ].append( round( stats . pvariance( tiempos_ejecucion , mu=media ) , 5 ) )
23     t_csv[ "desv" ].append( round( stats . pstdev( tiempos_ejecucion , mu=media ) , 5 ) )

```

Generar_datos.py

Cuadro 1: Fragmento de *data frame* generado.

grafo	generador	algoritmo_fm	vertices	densidad	aristas	f	s	mediana	varianza	desv
v256a8140	fast_gnp	shortest_a_path	256	0.24939	8140	33	101	0.06247	0.00004	0.00618
v256a8141	fast_gnp	edmonds_karp	256	0.24939	8140	33	101	0.04692	0.00001	0.00318
v256a8142	fast_gnp	preflow_push	256	0.24939	8140	33	101	0.08225	0.00025	0.01596
v256a8143	fast_gnp	shortest_a_path	256	0.24939	8140	33	101	0.06251	0.00004	0.00638
v256a8144	fast_gnp	edmonds_karp	256	0.24939	8140	33	101	0.05291	0.00007	0.00834
v256a8145	fast_gnp	preflow_push	256	0.24939	8140	33	101	0.08253	0.00017	0.01296
v256a8146	fast_gnp	shortest_a_path	256	0.24939	8140	33	101	0.06903	0.00020	0.01398
v256a8147	fast_gnp	edmonds_karp	256	0.24939	8140	33	101	0.05296	0.00003	0.00566

En este otro fragmento del código es donde se generan los grafos y se llama la función (`Algoritmo_FM()`).

```

1 numero_intancias = 10
2 mediciones = 5
3 archivo_CSV = "Datost4.csv"
4 control_iteraciones = 0
5
6 generadores_grafos = {
7     "fast_gnp_random_graph": nx.fast_gnp_random_graph ,
8     "binomial_graph": nx.binomial_graph ,
9     "erdos_renyi_graph": nx.erdos_renyi_graph
10 }
11
12 algoritmos_flujo = {
13     "shortest_augmenting_path": shortest_augmenting_path ,
14     "maximum_flow": maximum_flow ,
15     "preflow_push": preflow_push
16 }
17
18 t_csv = {
19     "grafo": [] , "generador_grafo": [] ,
20     "algoritmo_fm": [] , "vertices": [] ,
21     "densidad": [] , "aristas": [] ,
22     "f": [] , "s": [] , "media": [] ,
23     "mediana": [] , "varianza": [] ,
24     "desv": []
25 }
26 for generador_grafo in generadores_grafos:
27     for inst_g_n in [round(pow(2, value + 1))
28                      for value in
29                      range(7, 11 )]:
30
31
32     for grafo in range(1, numero_intancias + 1):
33         f = np.random.randint(1, high=(inst_g_n - 1), dtype="int")
34         s = np.random.randint(1, high=(inst_g_n - 1), dtype="int")

```

```

36
37     while s == f:
38         f = np.random.randint(1, high=(inst_g_n - 1), dtype="int")
39         s = np.random.randint(1, high=(inst_g_n - 1), dtype="int")
40         Grafo = generadores_grafos[generador_grafo](inst_g_n, 0.25, seed=None)
41         aristas = Grafo.number_of_edges()
42         p_n_distribuidos = np.random.normal(15, 0.2, aristas)
43         incremento = 0
44
45         for (u, v) in Grafo.edges():
46             Grafo.edges[u, v][“capacity”] = p_n_distribuidos[incremento]
47             incremento += 1
48             pesos = []
49             for (u, v) in Grafo.edges():
50                 t = Grafo.edges[u, v][“capacity”]
51                 pesos.append(t)
52             plt.figure(figsize=(8, 6))
53             n = plt.hist(pesos, bins=70, color="#932525", alpha=1, edgecolor = ‘black’,
54             linewidth=1)
55             plt.xlabel(‘Pesos de los Arcos’)
56             plt.ylabel(‘Frecuencia de ocurrencia’)
57
58             plt.savefig(“histografi.png”)
59             plt.savefig(“histografi.eps”)
60             for instancia_grafo in range(1, 6):
61                 for algoritmo in algoritmos_flujo:
62                     d= Algoritmo_FM(algoritmo, Grafo, f, s, control_iteraciones)
63                     control_iteraciones=d
64
65 ds = pd.DataFrame(t_csv)

```

Generar_datos.py

4. Resultados del análisis de los datos

Con los datos recopilado de las ejecuciones de los tres algoritmos de flujo máximo con los ciento veinte grafos y sus cinco combinaciones de fuentes y sumidero, se realizó una serie de diagramas y pruebas estadísticas. A continuación, se muestra un el fragmento de código donde se lleva a cabo lo antes mencionado.

```

1 df = pd.read_csv(“Datost4.csv”, index_col=None, usecols=[1,2,3,4,9], dtype={‘
2   generador_grafo’: ‘category’,
3   : ‘category’, ‘vertices’: ‘category’, ‘densidad’:np.float64, ‘mediana’: np.float64} )
4 modelo = ols(‘mediana_log ~ generador_grafo+algoritmo_fm+vertices+densidad+
5   generador_grafo*algoritmo_fm+algoritmo_fm*vertices+vertices*densidad+
6   generador_grafo*vertices+generador_grafo*densidad+algoritmo_fm*densidad’ ,data=df).
7   fit()
8 print(modelo.summary())
9 modelo_csv = open(“Anova_Mult.csv”, ‘w’)
10 aov_table = sm.stats.anova_lm(modelo, typ=2)
11 df1=pd.DataFrame(aov_table)
12 df1.to_csv(“modelo.csv”)
13 for column in range(0, df[“densidad”].count()):
14     pass
15     if df.iat[column, 3] >=0.2061 and df.iat[column, 3] < 0.20854:
16         df.iat[column, 3] = 1
17     elif df.iat[column, 3] >=0.20854 and df.iat[column, 3] < 0.21098:
18         df.iat[column, 3] = 2
19     else:

```

```

16     df.iat [column , 3] = 3
17 print (df[ "densidad" ])
18 df[ 'densidad' ]. replace ({1:"baja" , 2: 'media' , 3:'alta' } , inplace= True)
19 print (df[ "densidad" ])
20 logX = np.log1p(df[ 'mediana' ])
21 df = df.assign (mediana_log=logX . values)
22 df.drop ([ 'mediana' ] , axis= 1, inplace= True)
23
24 factores=[ "vertices" , "generador_grafo" , "densidad" , "algoritmo_fm" ]
25 plt.figure (figsize=(8, 6))
26 for i in factores :
27     print (rp.summary_cont (df[ 'mediana_log' ]. groupby (df[ i ])))
28
29 anova = pg.anova (dv='mediana_log' , between=i , data=df, detailed=True , )
30 pg . export_table (anova , ("ANOVA" +i+ ".csv"))
31
32 ax=sns.boxplot (x=df[ "mediana_log" ] , y=df[ i ] , data=df, palette="cubehelix")
33
34 plt.savefig ("boxplot_" + i + ".eps" , bbox_inches='tight')
35 tukey = pairwise_tukeyhsd (endog = df[ "mediana_log" ] , groups= df[ i ] , alpha=0.05)
36
37 tukey . plot_simultaneous ( xlabel='Tiempo' , ylabel=i)
38 plt.vlines (x=49.57 , ymin=-0.5 , ymax=4.5 , color="red")
39 plt.savefig ("simultaneous_tukey" + i + ".eps" , bbox_inches='tight')
40
41 print (tukey . summary ())
42 t_csv = open ("Tukey" +i+ ".csv" , 'w')
43 with t_csv :
44     writer = csv.writer (t_csv)
45     writer . writerows (tukey . summary ())
46     plt.show()

```

Analisis_datos.py

4.1. Análisis de varianza (ANOVA)

El análisis de varianza (ANOVA) es la técnica central en el análisis de datos experimentales. La idea general de esta técnica es separar la variación total en las partes con las que contribuye cada fuente de variación en el experimento. En el caso de los diseños completamente al azar se separan la variabilidad debida a los tratamientos y la debida al error. Cuando la primera predomina sobre la segunda, es cuando se concluye que las medias son diferentes. Cuando los tratamientos no dominan contribuyen igual o menos que el error, por lo que se concluye que las medias son iguales [6].

Para analizar si los diferentes factores (algoritmo generador de grafo, algoritmo de flujo máximo, numero de vértices y densidad del grafo) influían en la variable dependiente *tiempo de ejecución* se realizó un ANOVA de un factor para cada caso.

4.1.1. Influencia del algoritmo generador de grafos en el tiempo de ejecución.

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

En el cuadro 2 se muestra que no existen diferencia entre las medianas de los grupos de factores ya que el **p-unc** es mayor que 0,05 por lo que se acepta la hipótesis de que el tipo de generador no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 3 de la página 7.

Cuadro 2: ANOVA, relación del algoritmo generador con el *tiempo de ejecución*

Factor	SS	DF	MS	F	p-unc	np2
generador_grafo	0.280	2	0.140	0.354	0.702	0
Within	712.085	1797	0.396			

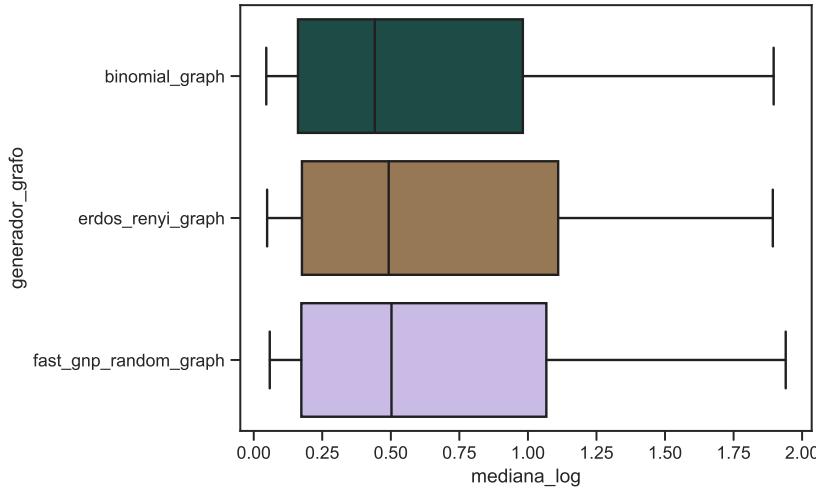


Figura 3: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con los algoritmos generadores de grafos.

4.1.2. Influencia del algoritmo de flujo máximo en el tiempo de ejecución.

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 3: ANOVA, relación del algoritmo de flujo máximo con el *tiempo de ejecución*

Factor	SS	DF	MS	F	p-unc	np2
algoritmo.fm	1.499	2	0.749	1.895	0.151	0.002
Within	710.867	1797	0.396			

En el cuadro 3 se muestra que no existen diferencias entre las medianas de los grupos de factores ya que el **p-unc** es mayor que 0,05 por lo que se acepta la hipótesis de que el tipo de algoritmo no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 4 de la página 8.

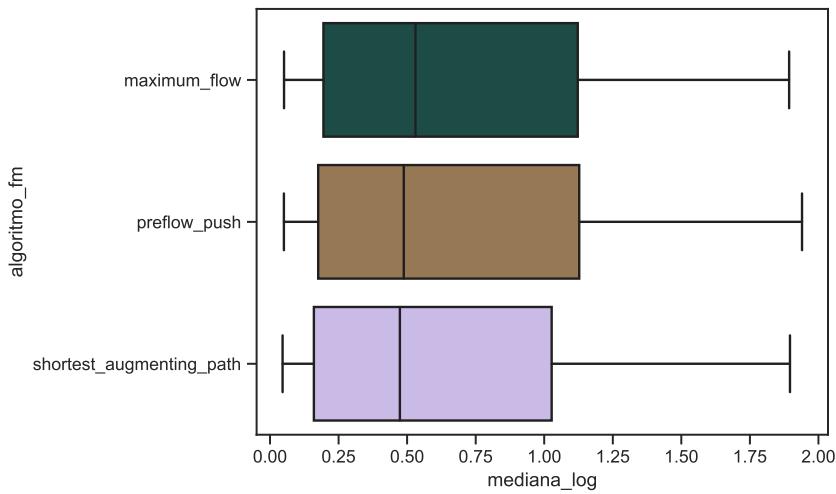


Figura 4: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con los algoritmos de flujo máximo.

4.1.3. Influencia del número de vértices en el tiempo de ejecución.

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 4: ANOVA, relación del número de vértices con el *tiempo de ejecución*

Factor	SS	DF	MS	F	p-unc	np2
vértices	706.976	3	235.659	78536.187	0	0.992
Within	5.389	1796	0.003			

En el cuadro 4 se muestra que existen grandes diferencia entre las medianas de los grupos de factores ya que el **p-unc** es menor que 0,05 por lo que se rechaza la hipótesis de que la cantidad de vértices no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 5 de la página 9. Por tal motivo se realiza la prueba de Tukey mostrara las diferencias entre las medianas de los factores, lo que se evidencia en el cuadro 5 de la página 9.

En el cuadro 5 se puede observar que en todos los casos se rechaza la hipótesis. En la figura 6 de la página 10 muestra claramente este hecho.

4.1.4. Influencia de la densidad de los grafos en el tiempo de ejecución.

En el caso del factor densidad se hizo una categorización por rangos para hacer cómoda la visualización de su relación con la variable dependiente estos rangos se obtuvieron a través de los contenedores que devolvió el histograma que se muestra en la figura 7 de la página 11.

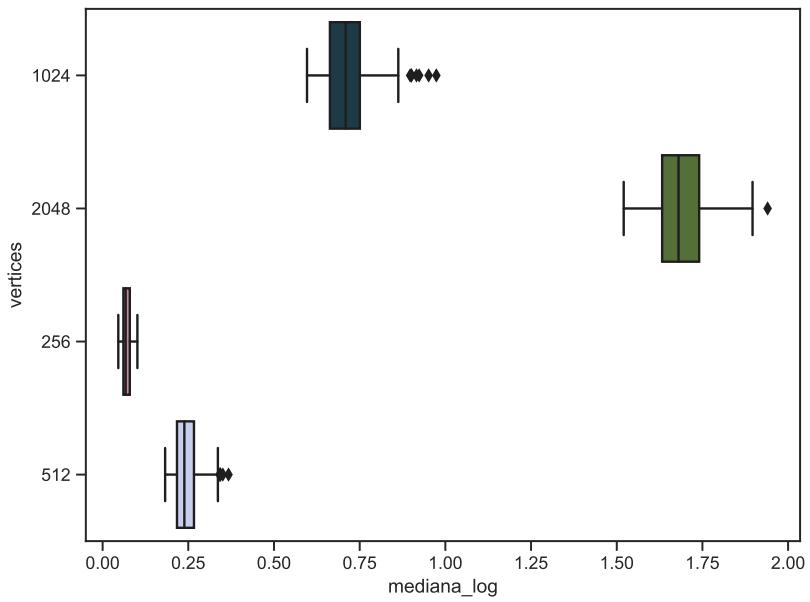


Figura 5: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con el número de vértices.

Cuadro 5: Tukey, influencia del número de vértices en el tiempo de ejecución.

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>Rechazar</i>
1024	2048	0.97	0.9606	0.9794	<i>Se rechaza</i>
1024	256	-0.6445	-0.6539	-0.6352	<i>Se rechaza</i>
1024	512	-0.4689	-0.4782	-0.4595	<i>Se rechaza</i>
2048	256	-1.6146	-1.624	-1.6052	<i>Se rechaza</i>
2048	512	-1.4389	-1.4483	-1.4295	<i>Se rechaza</i>
256	512	0.1757	0.1663	0.1851	<i>Se rechaza</i>

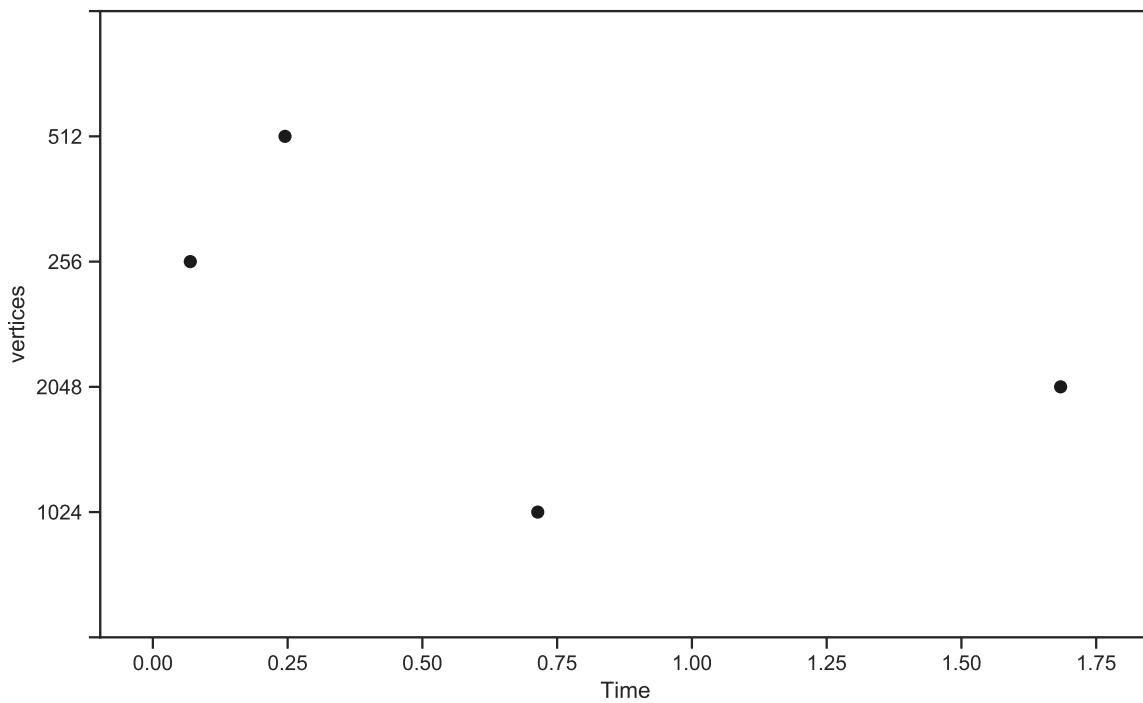


Figura 6: Diagrama simultaneo que relaciona los tiempos de ejecución con los grupos del factor número de vértices.

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 6: ANOVA, influencia de la densidad de los grafos en el tiempo de ejecución.

Factor	SS	DF	MS	F	p-unc	np2
densidad	170.777	2	85.388	283.320	0.000	0.24
Within	541.589	1797	0.301			

En el cuadro 7 se muestra que como en el cuadro 4 existen grandes diferencia entre las medianas de los grupos de factores ya que el **p-unc** es menor que 0,05 por lo que se rechaza la hipótesis de que la densidad de los grafos no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 8 de la página 12. Por tal motivo se realiza la prueba de Tukey mostrara las diferencias entre las medianas de los factores, lo que se evidencia en el cuadro 7 de la página 11.

En el cuadro 7 se puede observar que en dos de los tres casos se rechaza la hipótesis y en el pareo de densidad alta y baja no existen diferencias estadísticas en cuanto a la mediana. En la figura 9 de la página 12 muestra claramente este hecho.

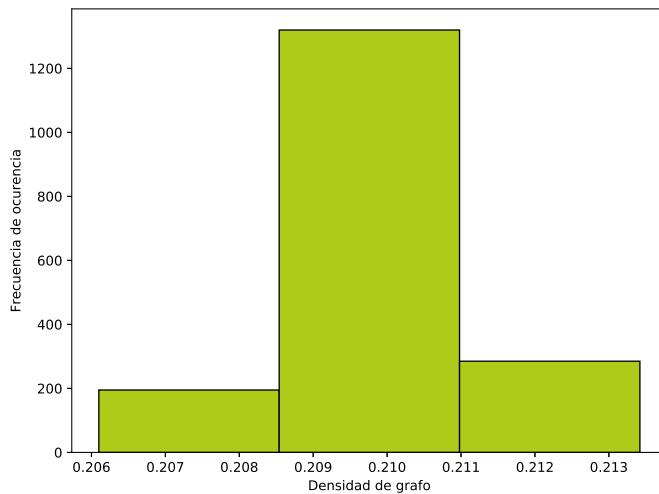


Figura 7: Histograma de densidad de los grafos.

Cuadro 7: Tukey,influencia de la densidad de los grafos en el tiempo de ejecución.

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>Rechazar</i>
alta	baja	-0.1085	-0.2281	0.0112	<i>No se rechaza</i>
alta	media	0.6497	0.5656	0.7338	<i>Se rechaza</i>
baja	media	0.7582	0.6594	0.8569	<i>Se rechaza</i>

4.1.5. Influencia de los cuatro factores (algoritmo generador de grafo, algoritmo de flujo máximo, número de vértices y densidad del grafo) en el tiempo de ejecución

Para analizar este caso se realizó ANOVA multifactorial dando como resultado el cuadro 8 de la página 13.

En el cuadro 8 se puede observar que los factores que influyen en el tiempo de ejecución son el número de vértices y la densidad de los grafos, que además existe una relación entre el número de nodos y la densidad de los grafos.

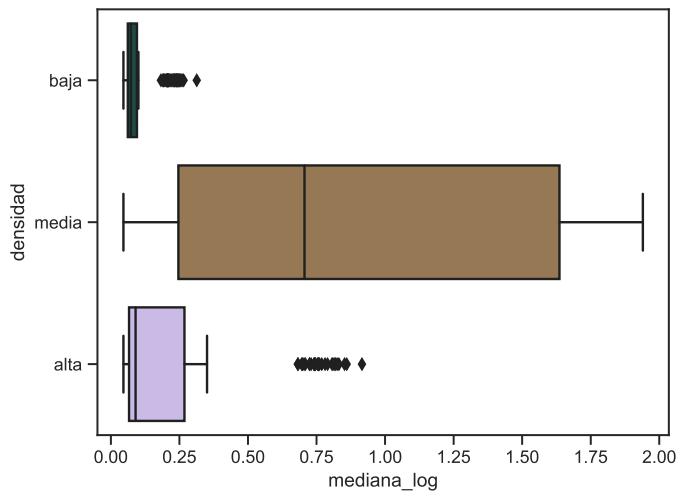


Figura 8: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con la densidad de los grafos.

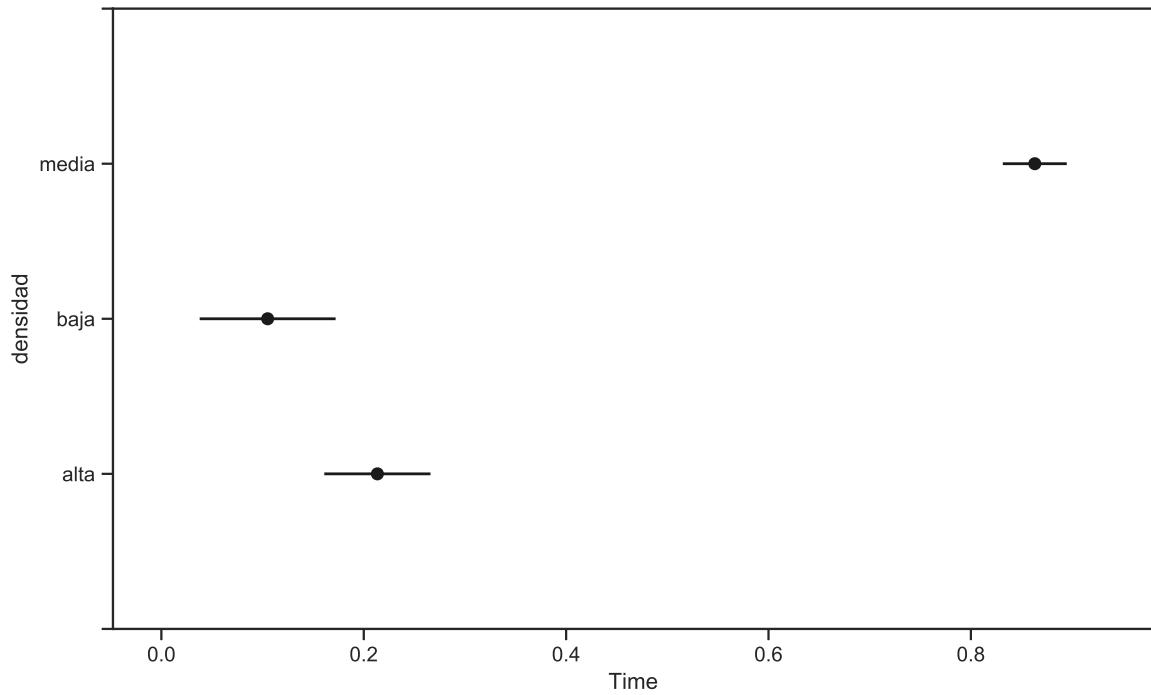


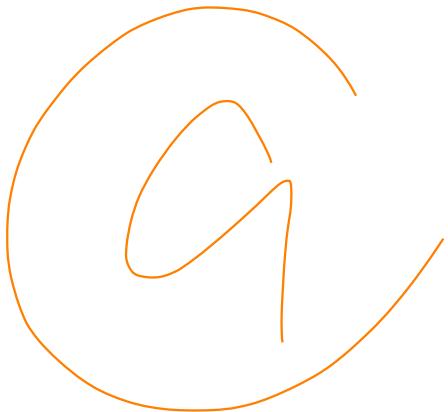
Figura 9: Diagrama simultaneo que relaciona los tiempos de ejecución con los grupos del factor densidad de los grafos.

Cuadro 8: ANOVA multifactor, influencia de los cuatro factores en el tiempo de ejecución.

	sum_sq	df	F	PR(>F)
generador_grafo	0.029	2	33.049	0.000
algoritmo_fm	-0.000	2	-0.143	1.000
vertices	46.476	3	35163.697	0.000
densidad	0.000	2	0.000	0.999
generador_grafo:algoritmo_fm	0.000	4	0.471	0.756
algoritmo_fm:vertices	0.000	6	0.024	0.874
vertices:densidad	0.000	6	0.042	0.958
generador_grafo:vertices	0.065	6	24.631	0.000
generador_grafo:densidad	0.000	4	0.018	0.893
algoritmo_fm:densidad	0.000	4	0.042	0.958
Residual	0.779	177		

Referencias

- [1] Bartłomiej Bosek, Dariusz Leniowski, Piotr Sankowski, and Anna Zych-Pawlewicz. Shortest augmenting paths for online matchings on trees. *Theory of Computing Systems*, 62(2):337–348, Feb 2018.
- [2] Desarrolladores de NetworkX. https://networkx.github.io/documentation/stable/reference/generated/networkx.generators.random_graphs.erdos_renyi_graph.html#networkx.generators.random_graphs.erdos_renyi_graph. Accessed: 01-04-2019.
- [3] Desarrolladores de NetworkX. https://networkx.github.io/documentation/stable/reference/generated/networkx.generators.random_graphs.fast_gnp_random_graph.html#networkx.generators.random_graphs.fast_gnp_random_graph. Accessed: 01-04-2019.
- [4] Desarrolladores de NetworkX. https://networkx.github.io/documentation/stable/reference/generated/networkx.generators.random_graphs.binomial_graph.html#networkx.generators.random_graphs.binomial_graph. Accessed: 01-04-2019.
- [5] Desarrolladores de NetworkX. https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.flow.maximum_flow.html#networkx.algorithms.flow.maximum_flow. Accessed: 01-04-2019.
- [6] Humberto Gutiérrez and Román de la Vara. *Análisis y diseño de experimentos*. The McGraw-Hill Companies, Inc., segunda edición edition, 2008. 60–74.
- [7] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.coloring.greedy_color.html. Accessed: 18-03-2019.
- [8] Sadegh Nobari, Xuesong Lu, Panagiotis Karras, and Stéphane Bressan. Fast random graph generation. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT ’11, pages 331–342, New York, NY, USA, 2011. ACM.



Tarea 5

5271

30 de abril de 2019

1. Algoritmo generador de grafos

Gracias a la versatilidad de los grafos como modelo de representación de datos, los procesos aleatorios de generación de grafos también son relevantes en aplicaciones que van desde la física, biología a la sociología [8].

Otra de las aplicaciones de los grafos de generación aleatoria es en la representación de una red telefónica donde los vértices son centrales telefónicas, las aristas son los enlaces troncales los cuales presentan cierta capacidad de llamadas. De esta red se desea conocer el flujo máximo entre una central telefónica fuente y una central telefónica sumidero.

En realización de la tarea anterior se seleccionaron tres algoritmos generadores de grafos de la biblioteca de *networkx*, con el objetivo de crear los grafos con pesos con distribución normal a los que se le aplicaron los tres algoritmos de flujo máximo para realizar los experimentos. Los tres algoritmos generadores antes mencionados son los siguientes.

Los algoritmos son los siguientes:

- *Erdős Rényi graph* (en el modelo $G(n, p)$, el grafo se construye conectando los n vértices al azar. Cada arista se incluye en el grafo con probabilidad p independiente de cualquier otro borde) [3].
- *Fast gnp random graph* (este algoritmo recibe como parámetros n números de vértices y probabilidad p de ocurrencia de aristas, el mismo devuelve un grafo aleatorio) [4].
- *Binomial graph* (este algoritmo recibe como parámetros n números de vértices y probabilidad p de ocurrencia de aristas, el mismo devuelve un grafo aleatorio, para grafos dispersos (para valores pequeños de p), *Fast gnp random graph* es un algoritmo más rápido) [5].

De los algoritmos anteriores el seleccionado para la realización de esta tarea es el *Erdős Rényi graph* dado que la tarea anterior arrojo que ninguno de los tres influían en el tiempo de ejecución de los algoritmo de flujo máximo que se le aplicaron a los grafos generados. como se muestra en el cuadro 1, es decir que los tres generan grafos con características similares.

Cuadro 1: ANOVA, relación del algoritmo generador con el *tiempo de ejecución*

Factor	SS	DF	MS	F	p-unc	np2
generador_grafo	0.280	2	0.140	0.354	0.702	0
Within	712.085	1797	0.396	-	-	-

En el cuadro 1 se muestra que no existen diferencia entre las medianas de los grupos de factores ya que el *p - unc* es mayor que 0,05 por lo que se acepta la hipótesis de que el tipo de generador no influye en el *tiempo de ejecución*.

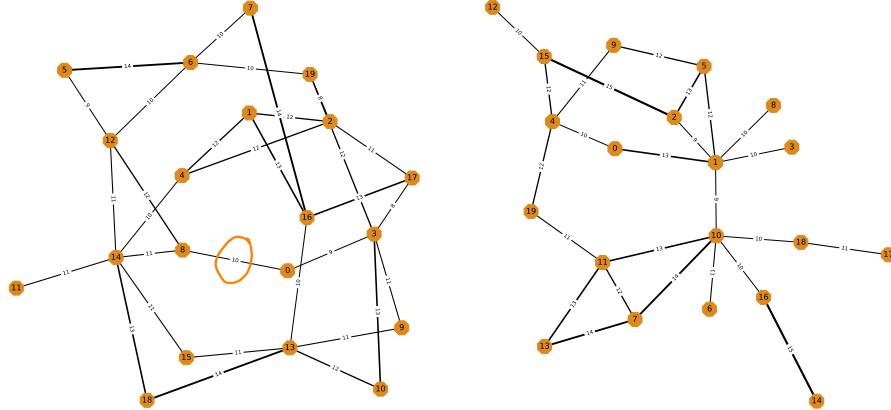
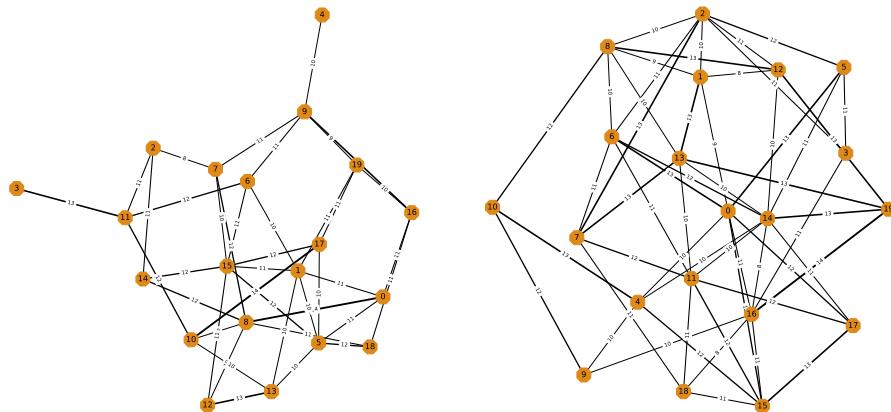
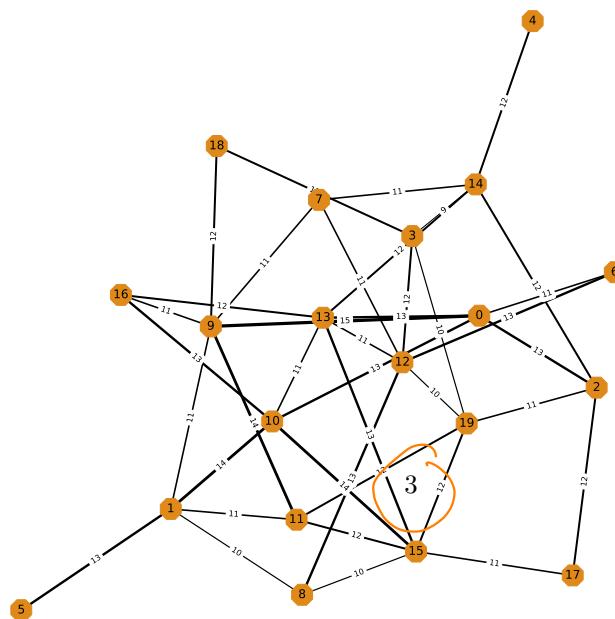
Para la generación de los grafos con el algoritmo *Erdős Rényi graph* hizo el siguiente código de Python donde se generan cinco grafos con la misma cantidad de vértices pero con diferentes números de aristas y capacidades de las mismas como se muestra en la figura 1 de la página 3.

```

1 def leer_capacity(G):
2     capacity = []
3     for (u, v) in G.edges():
4         capacity.append(G.edges[u, v][“capacity”])
5     return capacity
6
7 Grafo = nx.erdos_renyi_graph(20, 0.19, seed=None)
8 aristas = Grafo.number_of_edges()
9 pesos_normalmente_distribuidos = np.random.normal(12, 1.5, aristas)
10 loop = 0
11 for (u, v) in Grafo.edges():
12     Grafo.edges[u, v][“capacity”] = pesos_normalmente_distribuidos[
13         loop]
14     loop += 1
15 df = pd.DataFrame()
16 df = nx.to_pandas_adjacency(Grafo, dtype=int, weight=‘capacity’)
17 df.to_csv(“Grafo5” + “.csv”, index=None, header=None)

```

Generar_g.py

(a) *Grafo1*, con 20 vértices(b) *Grafo2*, con 20 vértices(c) *Grafo3*, con 20 vértices(d) *Grafo4*, con 20 vértices(e) *Grafo5*, con 20 vértices

2. Algoritmo de flujo máximo

En los problemas de flujo en redes, las aristas representan vías por las que puede circular elementos: datos, agua, corriente eléctrica, llamadas telefónicas entre otras. Los pesos de las aristas representan la capacidad máxima de una vía: velocidad de una conexión, volumen máximo de agua, voltaje de una línea eléctrica, cantidad máxima llamadas entre otras; aunque es posible que la cantidad real de flujo sea menor.

El problema del flujo máximo consiste en lo siguiente: dado un grafo con pesos, $G = (V, A, W)$, que representa las capacidades máximas de los canales, un vértice fuente f y otro sumidero s en V , encontrar la cantidad máxima de flujo que puede circular desde f hasta s .

En la biblioteca de *networkx* encontramos varios algoritmos con los que podemos atacar los problemas de flujo máximo, para la realización de esta tarea se escogerá uno de los tres algoritmos utilizados en la tarea anterior pertenecientes a dicha librería. Basándonos en los resultados obtenidos en las pruebas realizadas en la tarea anterior.

Los algoritmos escogidos en la tarea anterior son los siguientes:

- *Shortest augmenting path* es uno de los enfoques más clásicos para la máxima coincidencia y los problemas de flujo máximo. Sorprendentemente, aunque esta idea es una de las técnicas más básicas, está lejos de ser completamente entendida. Es más fácil hablar de ello introduciendo el problema de emparejamiento bipartito en línea [1]. Este algoritmo encuentra el flujo máximo de un solo producto utilizando la ruta de aumento más corto y devuelve la red residual resultante después de calcular el flujo máximo.
- *Maximum flow* encuentra la ruta por la cual pasa la máxima cantidad de flujo, recibe como parámetros un grafo G , una fuente f , un sumidero s y además una capacidad que de no tenerla, se considera que el borde tiene una capacidad infinita. Se puede aplicar en grafos tanto dirigidos como no dirigidos  [6].
- *Preflow push* encuentra un flujo máximo de un solo producto utilizando el algoritmo de empuje previo al flujo de la etiqueta más alta. Esta función devuelve la red residual resultante después de calcular el flujo máximo. Este algoritmo tiene un tiempo de ejecución de $O(n^2\sqrt{m})$ para n vértices y m aristas  [7].

De los algoritmos anteriores el que se utiliza en esta tarea es el *Shortest augmenting path* por ser el algoritmo que mejores tiempos registró en la tarea anterior.

3. Algoritmo de acomodo

El algoritmo Kamada kawai layout posiciona los nodos utilizando la función de costo de longitud de camino *Kamada-Kawai*, dando la mejor visualización de los grafos generados.

4. Generación de datos

Con el objetivo de realizar las mediciones de los tiempos de ejecución de los algoritmo de flujo máximo seleccionado así como el calculo de las seis propiedades de los nodos (distribución de grado, coeficiente de agrupamiento, centralidad de cercanía, centralidad de carga, excentricidad, pagerank) se desarrolló el siguiente código.

En primer lugar, se crea una función (`Maxflow()`) que recibe como parámetros el grafo al que se le aplicara el algoritmo (`G`), la fuente (`a`) y sumidero (`b`). Al llamar esta función se genera con cada ejecución los datos que son guardados en un *data frame* del cual se muestra un fragmento en el cuadro 2 de la página 5.

Cuadro 2: Fragmento del *data frame* que contiene los datos recopilados.

Grafo	Fuente	Sumidero	Media	Mediana	FlujoMax	Grado	CoefAg	CentCer	CentCag	Excent	PageRag
Grafo1	0	1	0.026	0.020	19	2	0.000	0.388	0.048	4	0.035
Grafo1	15	7	0.022	0.020	22	2	0.000	0.413	0.035	4	0.034
Grafo2	0	9	0.029	0.020	23	2	0.000	0.396	0.044	4	0.039
Grafo2	2	4	0.028	0.024	34	3	0.333	0.404	0.123	4	0.056
Grafo3	1	2	0.044	0.052	30	6	0.267	0.543	0.084	3	0.068
Grafo3	1	15	0.042	0.024	63	6	0.267	0.543	0.084	3	0.068
Grafo4	0	13	0.079	0.084	69	7	0.143	0.613	0.096	2	0.064
Grafo4	3	10	0.055	0.032	33	3	0.333	0.475	0.016	3	0.032
Grafo5	3	15	0.020	0.020	43	4	0.167	0.463	0.059	4	0.052
Grafo5	8	2	0.076	0.074	33	3	0.000	0.452	0.034	4	0.039

En este fragmento del código se realiza toda la recopilación de la información necesaria para el análisis de los factores que influyen en el tiempo de ejecución y el flujo máximo, también dibuja los grafos a los que se les aplica el algoritmo.

```
1 def Leer_grafos (nombr):
2     dr = pd.read_csv((nombr+".csv"), header=None)
3     A = nx.from_pandas_adjacency(dr)
4     return A
5
6 def leer_capacity (G):
7     capacity = []
8     for (u, v) in G.edges():
9         capacity.append(G.edges[u, v][ "weight"])
10    return capacity
11
```

```

12
13 def Maxflow(G, a, b):
14     tiempo_inicial = dt.datetime.now()
15     # d =shortest_augmenting_path(G, a, b, capacity="weight")
16     for i in range(40):
17         d =shortest_augmenting_path(G, a, b, capacity="weight")
18         print( dt.datetime.now())
19     tiempo_final = (dt.datetime.now() - tiempo_inicial).
20     total_seconds()
21     return tiempo_final
22
23 def listnodos(d,f,s):
24     for nodes in d.nodes():
25         if d.nodes[nodes]!=f and d.nodes[nodes]!=s :
26             nodos.append(nodes)
27     return nodos
28 def Tiempos(G,nombre):
29     df = {"Grafo": [], "Fuente": [], "Sumidero": [], "Media": [], "Mediana": [],
30           "DesvStd": [], "flujoMax": [],
31           "Grado": [], "CoefAg": [], "CentCer": [], "CentCag": [],
32           "Excentricidad": [], "PageRag": []}
33     Nodes = G.nodes
34     print(Nodes)
35     for i in Nodes:
36         for j in Nodes:
37             if i != j:
38                 t = []
39                 for k in range(10):
40                     print(t, end="\n\n")
41                     time = Maxflow(G, i, j)
42                     t.append(time)
43                 PageRag = nx.pagerank(G, weight="capacity")
44                 df["Grafo"].append(nombre)
45                 df["Grado"].append(G.degree(i))
46                 df["CoefAg"].append(round(nx.clustering(G, i), 4))
47                 df["CentCer"].append(round(nx.closeness_centrality
48                               (G, i), 4))
49                 df["CentCag"].append(round(nx.load_centrality
50                               (G, i), 4))
51                 df["Excentricidad"].append(nx.eccentricity(G, i))
52                 df["PageRag"].append(round(PageRag[i], 4))
53                 df["Fuente"].append(i)
54                 df["Sumidero"].append(j)
55                 df["Media"].append(round(np.mean(t), 4))
56                 df["Mediana"].append(round(np.median(t), 4))
57                 df["DesvStd"].append(round(np.std(t), 4))
58                 df["flujoMax"].append(nx.maximum_flow_value
59                               (G, i, j, capacity="weight")))
60
61     dd = pd.DataFrame(df)
62     dd.to_csv(nombre+"D.csv", index=None)
63 G = Leer_grafos("Grafo1")
64 widths = []
65 widths = leer_capacity(G)
66 print(widths)
67 widths [:] = [(x - 7)/2 for x in widths]

```

```

66 plt.figure(figsize=(15, 15))
67 labels = {}
68 for u, v, data in G.edges(data=True):
69     labels[(u, v)] = data['weight']
70 position = nx.kamada_kawai_layout(G, dist=None, pos=None, scale
71 =0.5, center=None, dim=2)
72 nx.draw_networkx_nodes(G, position, node_size=800, node_color="#
73 de8919", node_shape="8")
74 nx.draw_networkx_edges(G, position, width=widths, edge_color='black
75 ')
76 nx.draw_networkx_edge_labels(G, position, edge_labels=labels,
77 font_size=10, label_pos=.5)
78 nx.draw_networkx_labels(G, position, font_size=15)
79
80 df = pd.DataFrame(position)
81 df.to_csv("position1" + ".csv", index=None, header=None)
82 plt.axis("off")
83 plt.savefig("Grafola" + ".png", bbox_inches='tight')
84 plt.savefig("Grafola" + ".eps", bbox_inches='tight')
85 plt.show(G)
86
87 d = shortest_augmenting_path(G, 4, 8, capacity="weight")
88 flowma = []
89 widths1 = []
90 widths0 = []
91 widths2 = []
92 widths3 = []
93 maxi = 0
94 nodos = []
95 nodosF = [4]
96 nodosS = [8]
97 nodos = listnodos(d, 4, 8)
98 for edges in d.edges():
99     widths.append(d.edges[edges]["flow"])
100    if d.edges[edges]["flow"] > 0:
101        widths1.append(d.edges[edges]["capacity"])
102        widths0.append(edges)
103        flowma.append(d.edges[edges]['flow'])
104    elif d.edges[edges]["flow"] == 0:
105        widths2.append(edges)
106        widths3.append(d.edges[edges]["capacity"])
107 print(widths1)
108 print(widths3)
109 flowma[:] = [x + 20 for x in flowma]
110 for i in flowma:
111    if i > maxi:
112        maxi = i
113 widths[:] = [x/10*x/6 for x in widths]
114 widths1[:] = [(x - 7)/2 for x in widths1]
115 widths3[:] = [(x - 7)/2 for x in widths3]
116 plt.figure(figsize=(15, 15))
117 position = pd.read_csv('position1.csv', header=None)
118 for u, v, data in d.edges(data=True):
119    if data['flow'] >= 0:
120        labels[(u, v)] = data['flow']
121 nx.draw_networkx_nodes(d, position, nodelist=nodos, node_size=800,

```

```

119     node_color="#de8919", cnode_shape="8")
120 nx.draw_networkx_nodes(d, position, nodelist=nodosF, node_size=800,
121     node_color="red", cnode_shape="8")
122 nx.draw_networkx_nodes(d, position, nodelist=nodosS, node_size=800,
123     node_color="green", cnode_shape="8")
124
125 nx.draw_networkx_edges(d, position, edgelist=widths2, edge_color='black',
126     width=widths3, arrows=False)
127 nx.draw_networkx_edges(d, position, edgelist=widths0, edge_cmap=plt.
128     cm.Purples, width=widths1, edge_color=flowma,
129     edge_vmin=0, edge_vmax=maxi)
130
131 nx.draw_networkx_edge_labels(d, position, edge_labels=labels,
132     font_size=10, label_pos=.5)
133 nx.draw_networkx_labels(d, position, font_size=15)
134
135 plt.axis("off")
136
137 plt.savefig("Grafo1cf" + ".png", bbox_inches='tight')
138 plt.savefig("Grafo1cf" + ".eps", bbox_inches='tight')
139
140 plt.show(G)
141 list=["Grafo1","Grafo2","Grafo3","Grafo4","Grafo5"]
142
143 for k in list:
144     print(k)
145     l = Leer_grafos(k)
146     Tiempos(l,k)

```

Leer_grafo.py

La aplicación del algoritmo de flujo máximo *Shortest augmenting path* a todas las posibles combinaciones de fuentes y sumidero de cada uno de los cinco grafos nos dio como resultado los valores de flujo máximo para cada una de estas combinaciones y nos muestra cómo va variando el óptimo desde la peor combinación fuente sumidero a la mejor.

Para el Grafo1 esto se muestra en la figura 2 de la página 9.

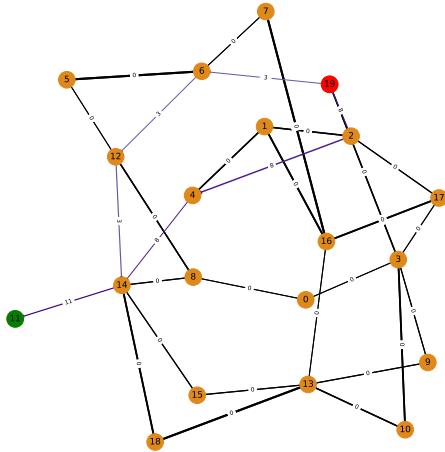
Para el Grafo2 esto se muestra en la figura 3 de la página 10.

Para el Grafo3 esto se muestra en la figura 4 de la página 11.

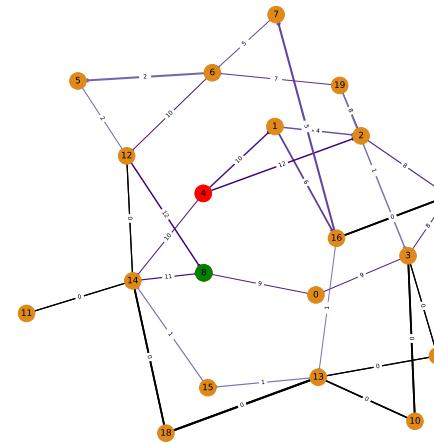
Para el Grafo4 esto se muestra en la figura 5 de la página 12.

Para el Grafo5 esto se muestra en la figura 6 de la página 13.

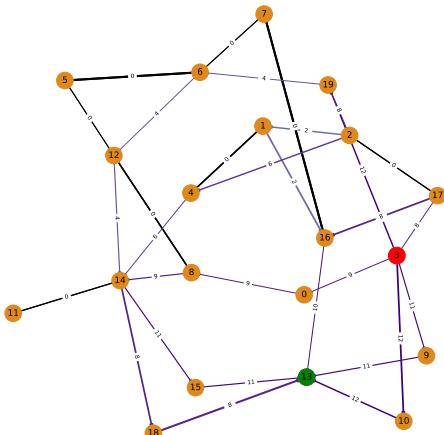




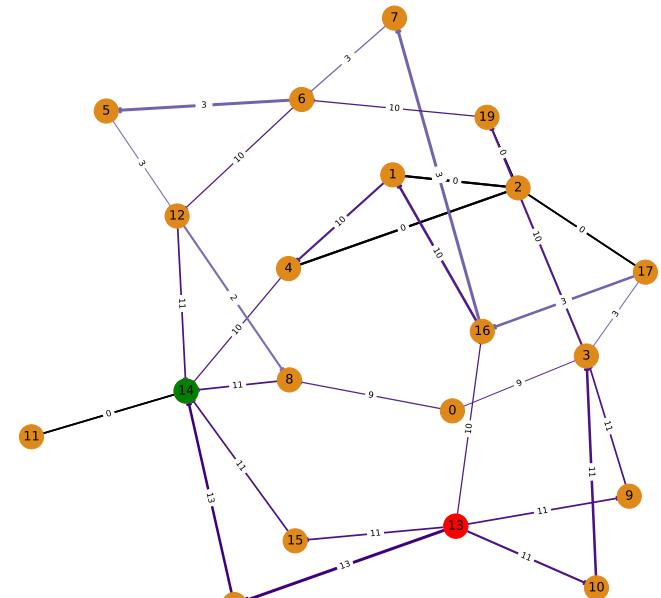
(a) Grafo 1, flujo máximo 11



(b) Grafo 1, flujo máximo 32

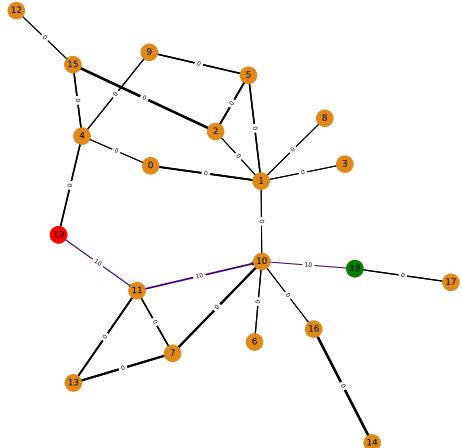


(c) Grafo 1, flujo máximo 52

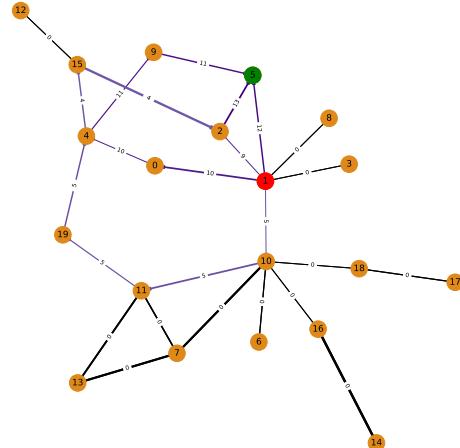


(d) *Grafo1*, flujo máximo 56

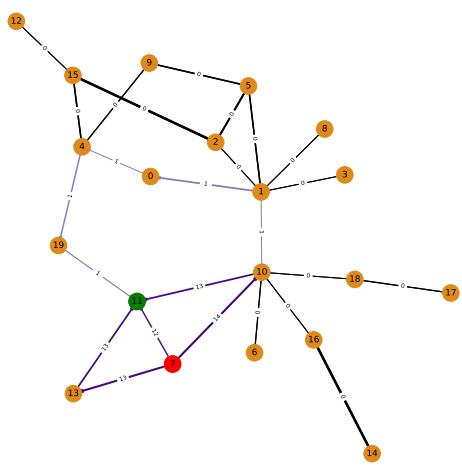
Figura 2: Grafos resultantes de la aplicación del algoritmo de flujo máximo, donde el grosor de las aristas representa la capacidad de las mismas, el color negro la ausencia de flujo, el color violeta la presencia de flujo y la intensidad del color violeta la cantidad de flujo que pasa por la arista. El color rojo del vértice representa la fuente y el vértice verde el sumidero.



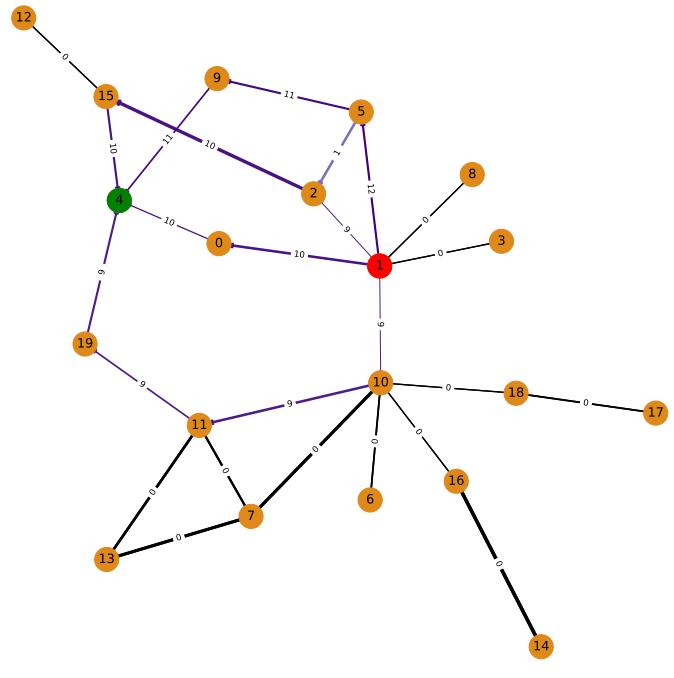
(a) *Grafo2*, flujo máximo 10



(b) *Grafo2*, flujo máximo 36



(c) *Grafo2*, flujo máximo 39



(d) *Grafo2*, flujo máximo 40

Figura 3: Grafos resultantes de la aplicación del algoritmo de flujo máximo, donde el grosor de las aristas representa la capacidad de las mismas, el color negro la ausencia de flujo, el color violeta la presencia de flujo y la intensidad del color violeta la cantidad de flujo que pasa por la arista. El color rojo del vértice representa la fuente y el vértice verde el sumidero.

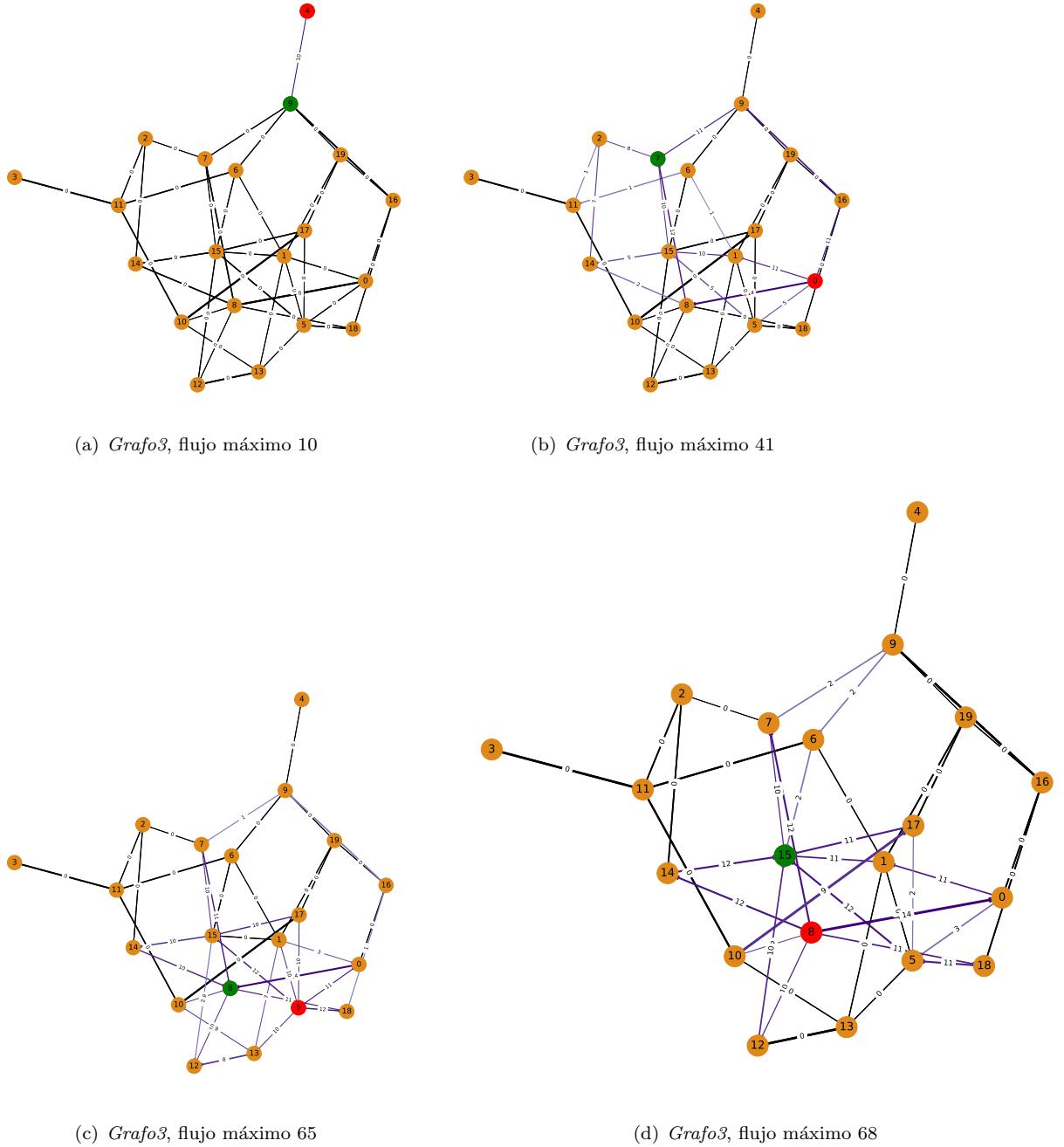
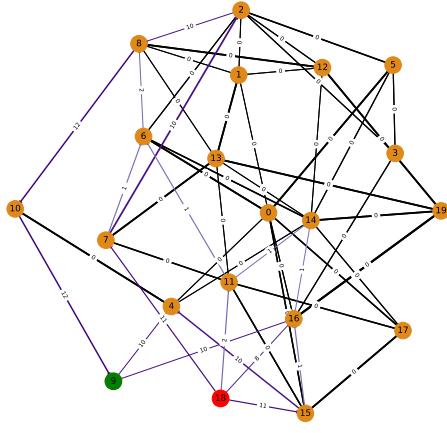
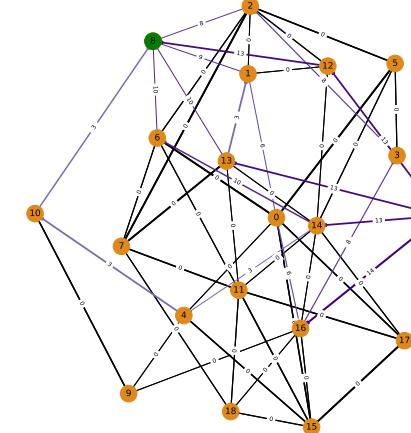


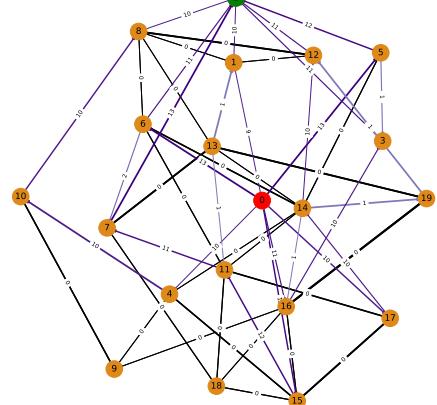
Figura 4: Grafos resultantes de la aplicación del algoritmo de flujo máximo, donde el grosor de las aristas representa la capacidad de las mismas, el color negro la ausencia de flujo, el color violeta la presencia de flujo y la intensidad del color violeta la cantidad de flujo que pasa por la arista. El color rojo del vértice representa la fuente y el vértice verde el sumidero.



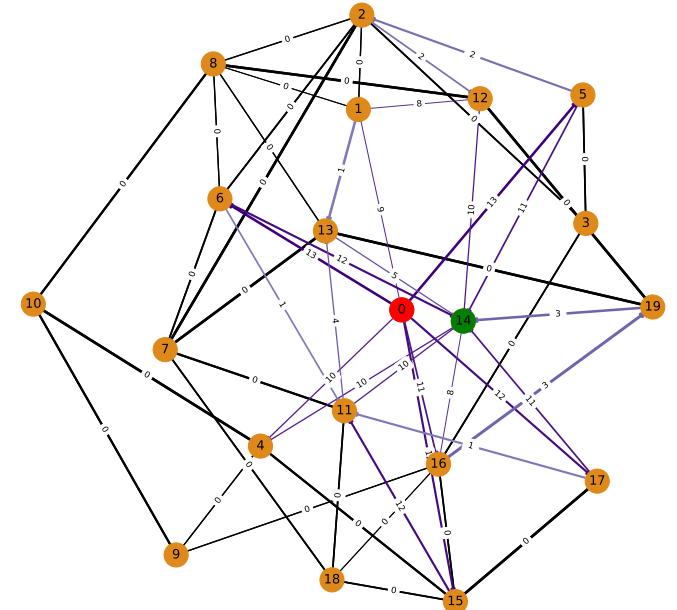
(a) *Grafo4*, flujo máximo 32



(b) *Grafo4*, flujo máximo 53



(c) *Grafo4*, flujo máximo 78



(d) *Grafo4*, flujo máximo 80

Figura 5: Grafos resultantes de la aplicación del algoritmo de flujo máximo, donde el grosor de las aristas representa la capacidad de las mismas, el color negro la ausencia de flujo, el color violeta la presencia de flujo y la intensidad del color violeta la cantidad de flujo que pasa por la arista. El color rojo del vértice representa la fuente y el vértice verde el sumidero.

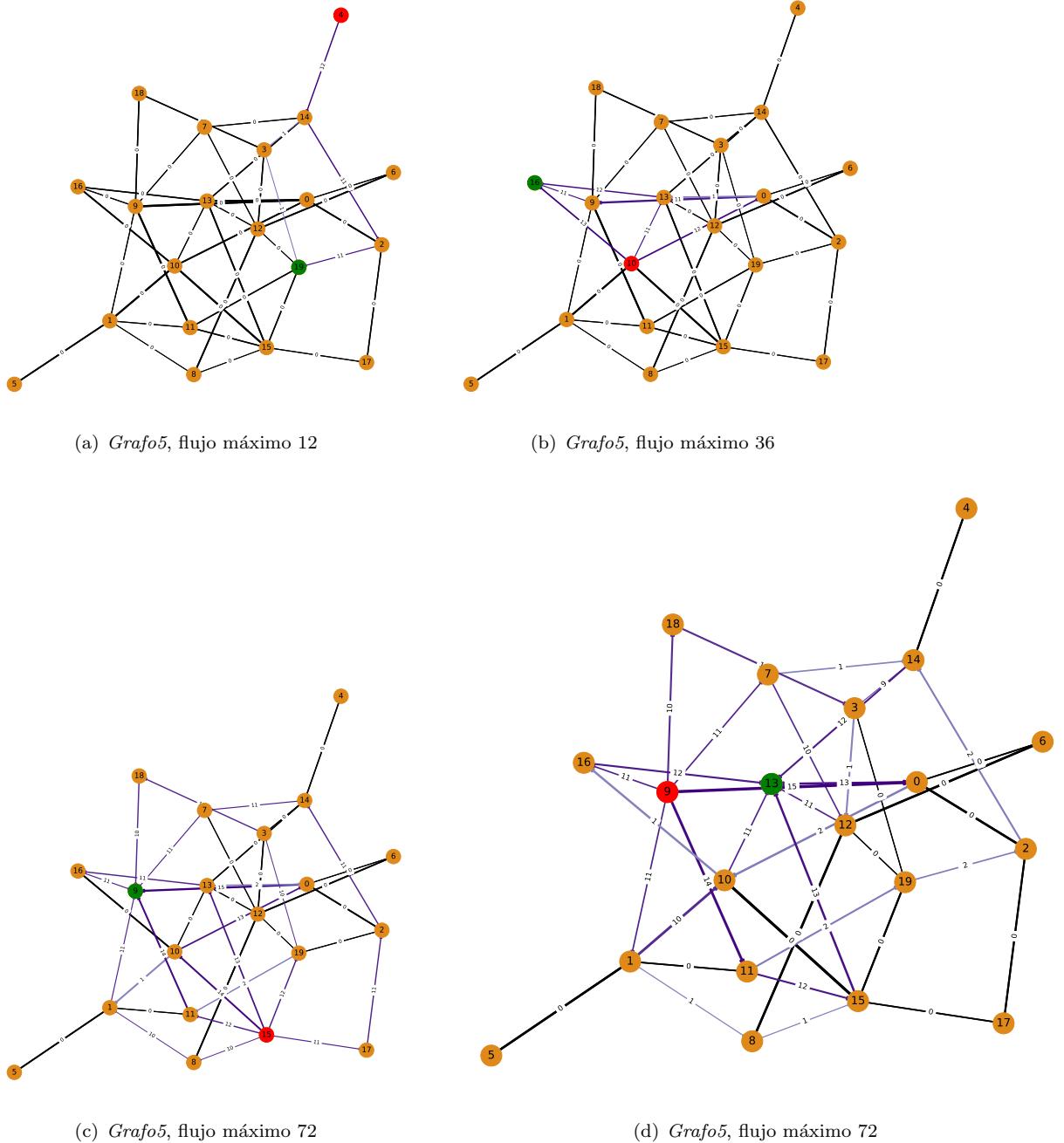


Figura 6: Grafos resultantes de la aplicación del algoritmo de flujo máximo, donde el grosor de las aristas representa la capacidad de las mismas, el color negro la ausencia de flujo, el color violeta la presencia de flujo y la intensidad del color violeta la cantidad de flujo que pasa por la arista. El color rojo del vértice representa la fuente y el vértice verde el sumidero.

5. Resultados del análisis de los datos

Con los datos recopilado de las ejecuciones del algoritmo de flujo máximo en los cinco grafos y todas las posibles combinaciones de fuentes y sumidero, se realizó histogramas para cada una de las seis características en los cinco grafos para determinar si los valores de la media del tiempo de ejecución tienen una distribución normal. Estos histogramas nos indican que la distribución no es normal como muestran la figuras 7, 8, 9, 10, 11, 12 de las páginas 15, 16, 17, 18, 19, 20 respectivamente.

Dado que la distribución de los valores de las características no es normal, se hizo una categorización por rangos para hacer cómoda la visualización de la relación con las variables dependientes, estos rangos se obtuvieron a través de los contenedores que devolvieron histogramas que se le aplicaron a cada propiedad.

Con esta categorización de las características se realizaron las pruebas estadísticas para analizar la influencia de las características estructurales de los grafos en el tiempo de ejecución del algoritmo de flujo máximo y el valor de flujo máximo.

El siguiente fragmento de código nos muestra la realización de las pruebas antes mencionadas.

```
1 logX = np.log1p(df[ 'Mediana' ])
2 df = df.assign(mediana_log=logX.values)
3 df.drop([ 'Mediana' ], axis= 1, inplace= True)
4
5 factores=[ "Grado" , "CoefAg" , "CentCer" , "CentCag" , "Excentricidad" , "
6 PageRag" ]
7 plt.figure(figsize=(8, 6))
8 for i in factores:
9     print(rp.summary_cont(df[ 'FlujoMax' ].groupby(df[ i ])))
10
11 anova = pg.anova (dv='FlujoMax' , between=i , data=df , detailed=
12 True , )
13 pg._export_table (anova,( "ANOVAFlujoMax"+i+".csv" ))
14
15 ax=sns.boxplot(x=df[ "FlujoMax" ] , y=df[ i ] , data=df , palette="cubehelix")
16
17 plt.savefig("boxplot_FlujoMax" + i + ".eps" , bbox_inches='tight')
18 tukey = pairwise_tukeyhsd(endog = df[ "FlujoMax" ] , groups= df[ i ] ,
19 alpha=0.05)
20
21 tukey.plot_simultaneous(xlabel='Flujo Maximo' , ylabel=i)
22
23 plt.savefig("simultaneous_tukey" + i + ".eps" , bbox_inches='tight')
24
25 print(tukey.summary())
26 t_csv = open("TukeyFlujoMax"+i+".csv" , 'w')
27 with t_csv:
```

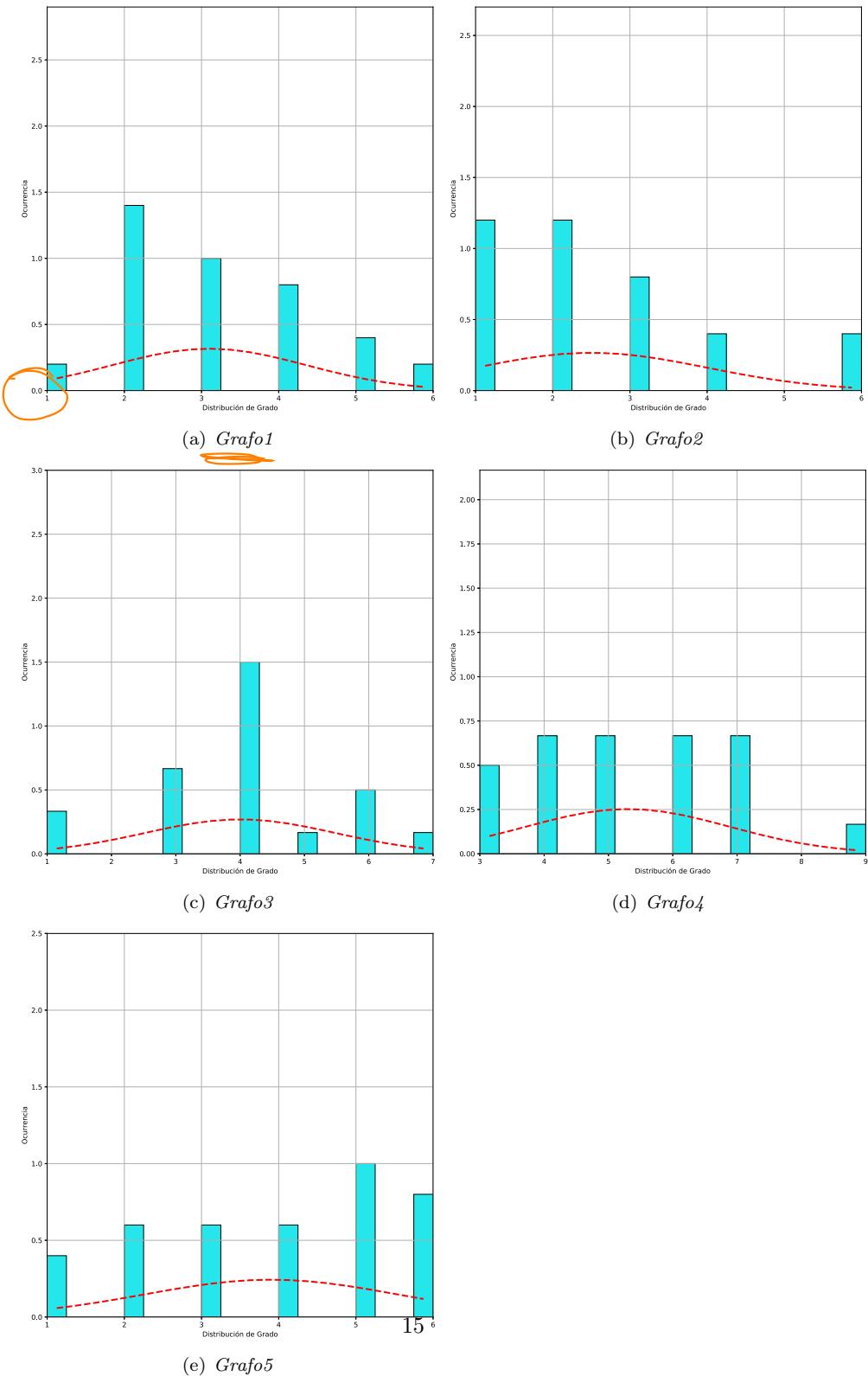


Figura 7: Histogramas que muestran la distribución de los valores de la característica distribución de grado.

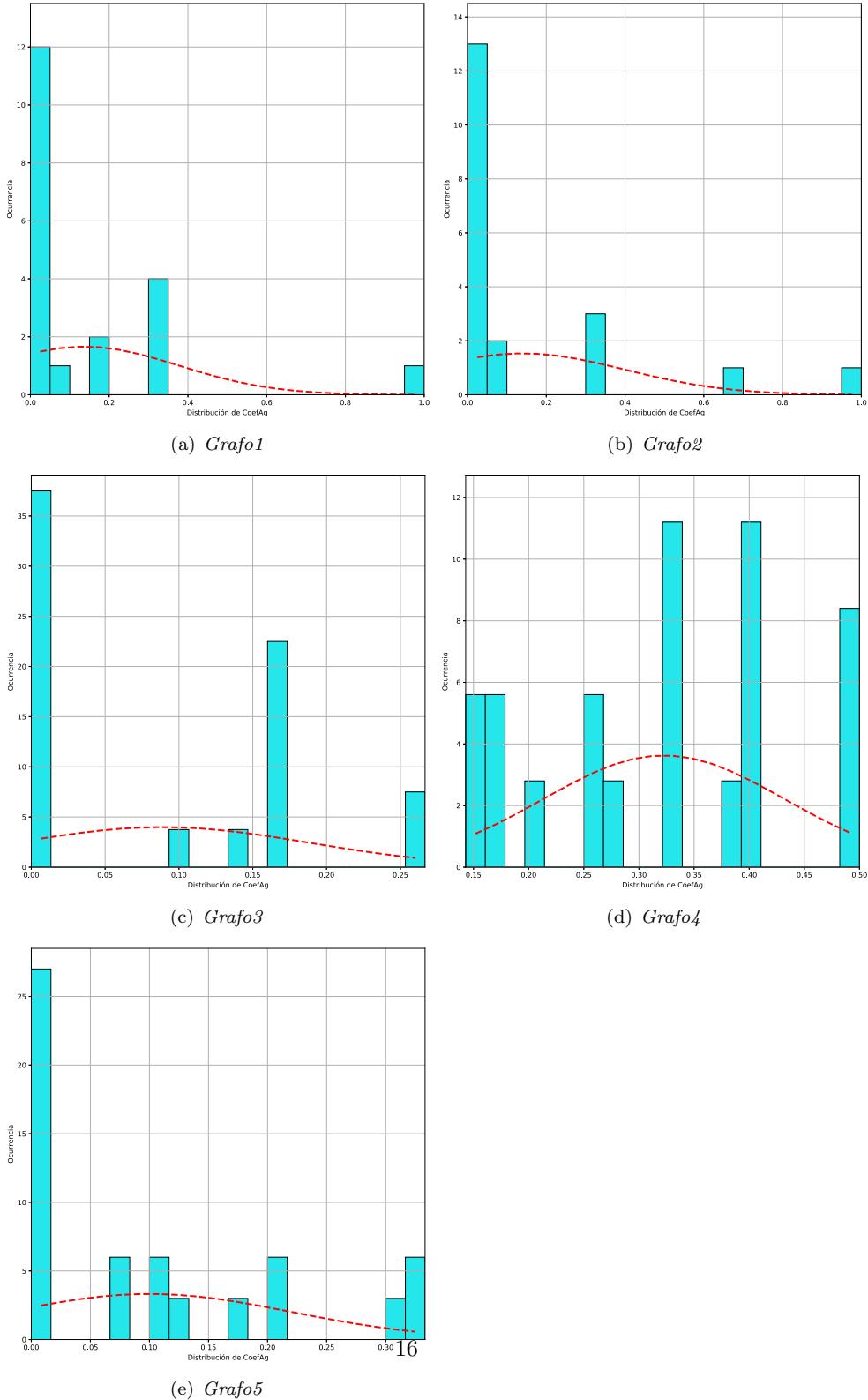


Figura 8: Histogramas que muestran la distribución de los valores de la característica coeficiente de agrupamiento.

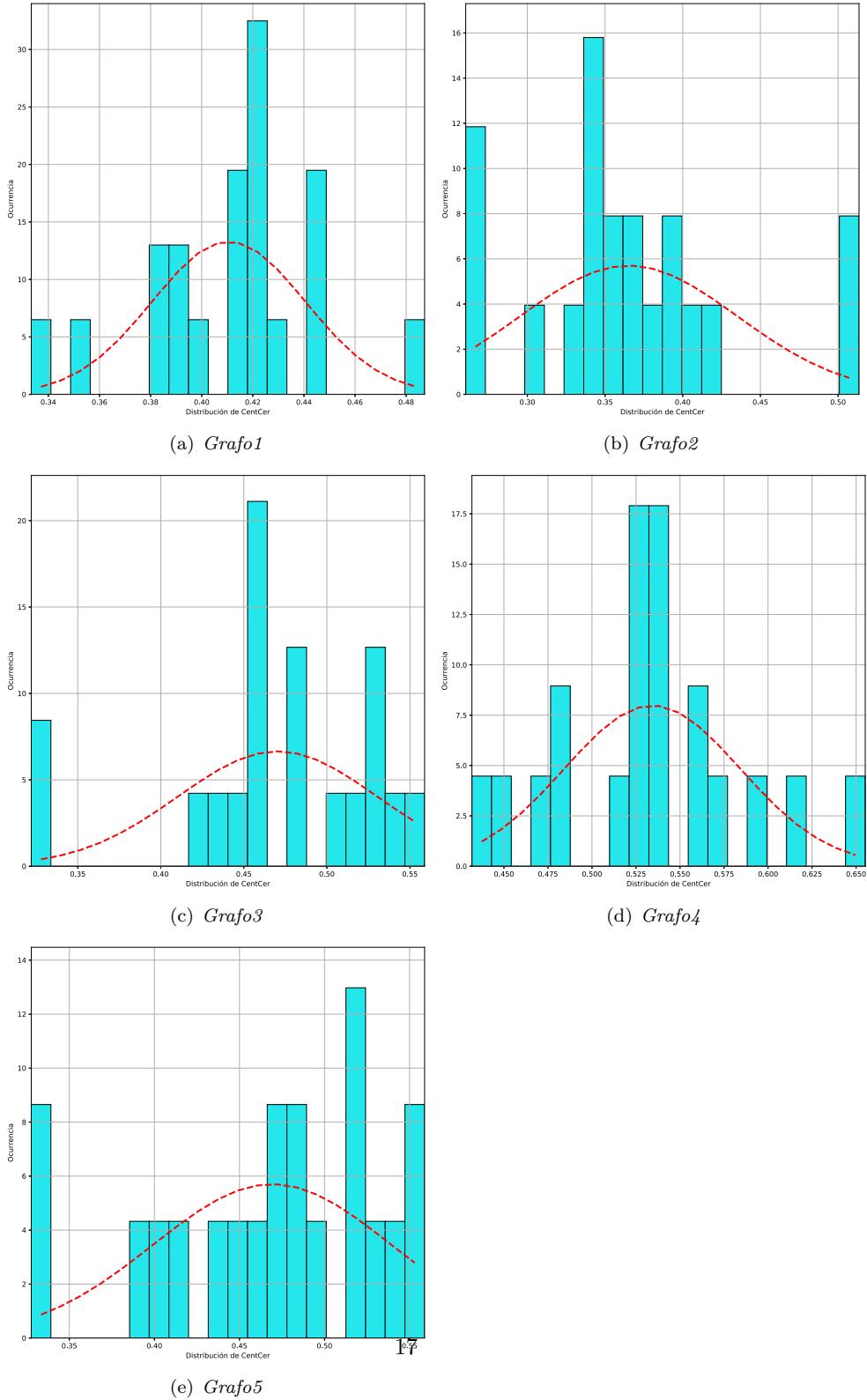


Figura 9: Histogramas que muestran la distribución de los valores de la característica centralidad de cercanía.

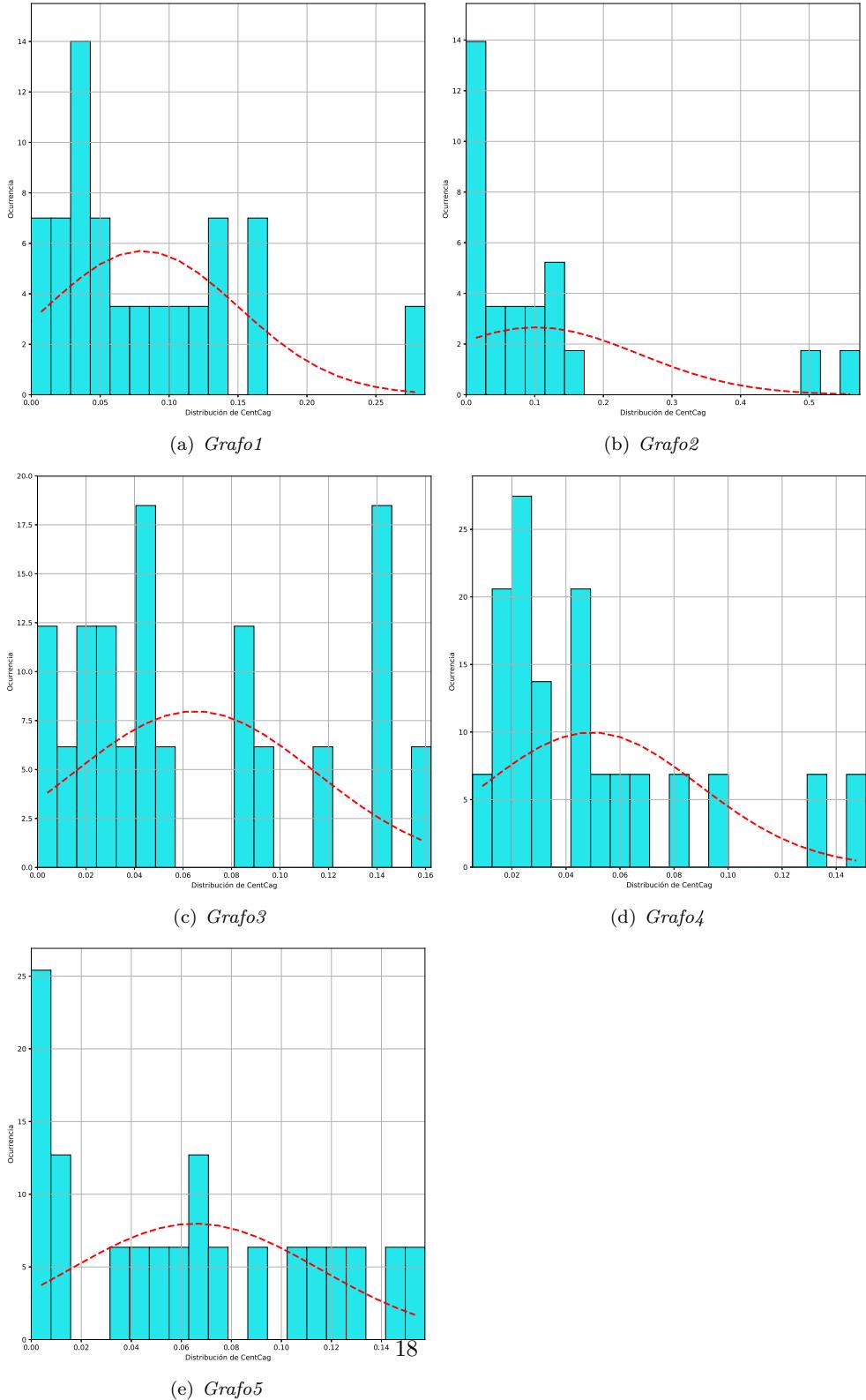


Figura 10: Histogramas que muestran la distribución de los valores de la característica centralidad de carga.

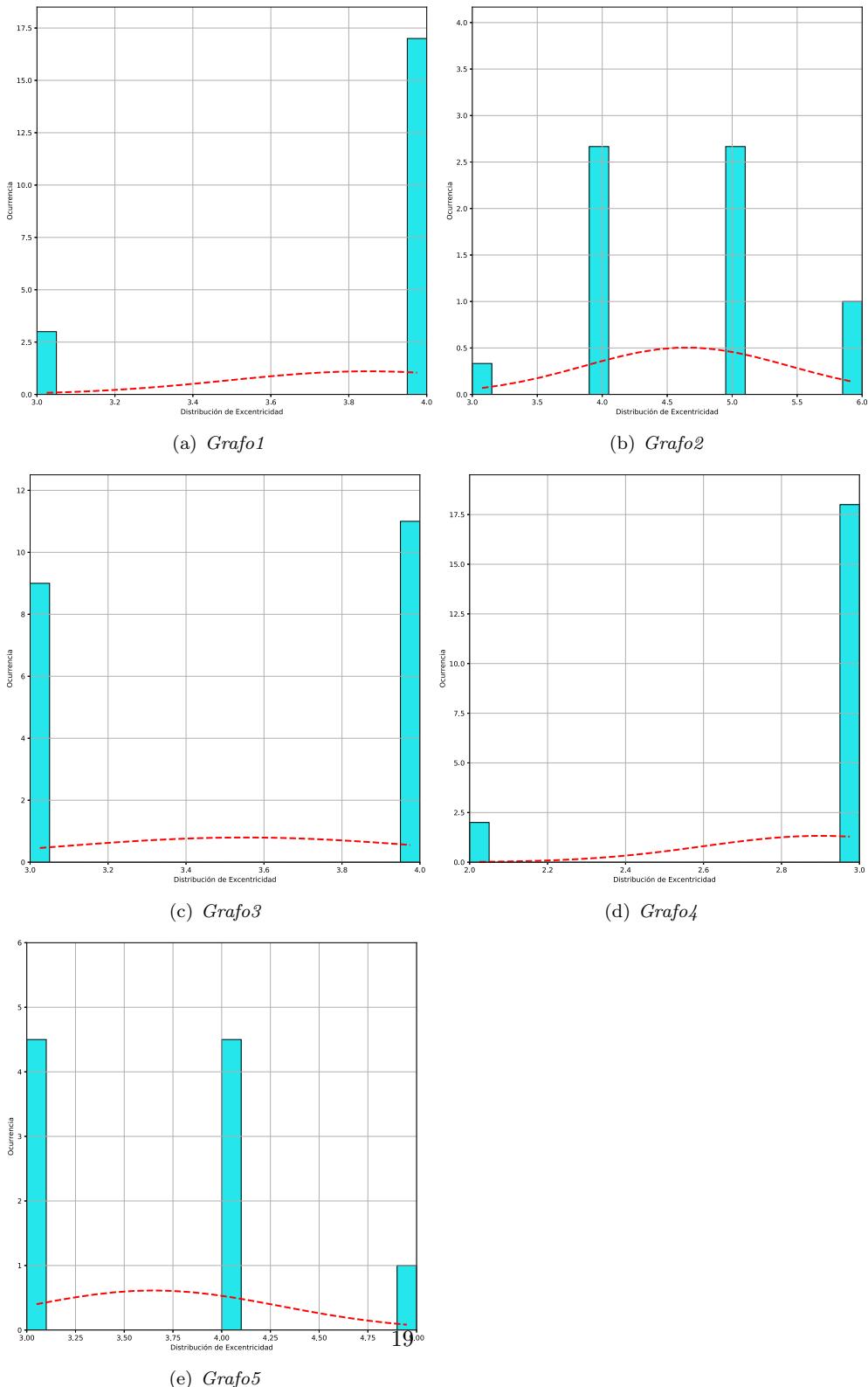


Figura 11: Histogramas que muestran la distribución de los valores de la característica excentricidad.

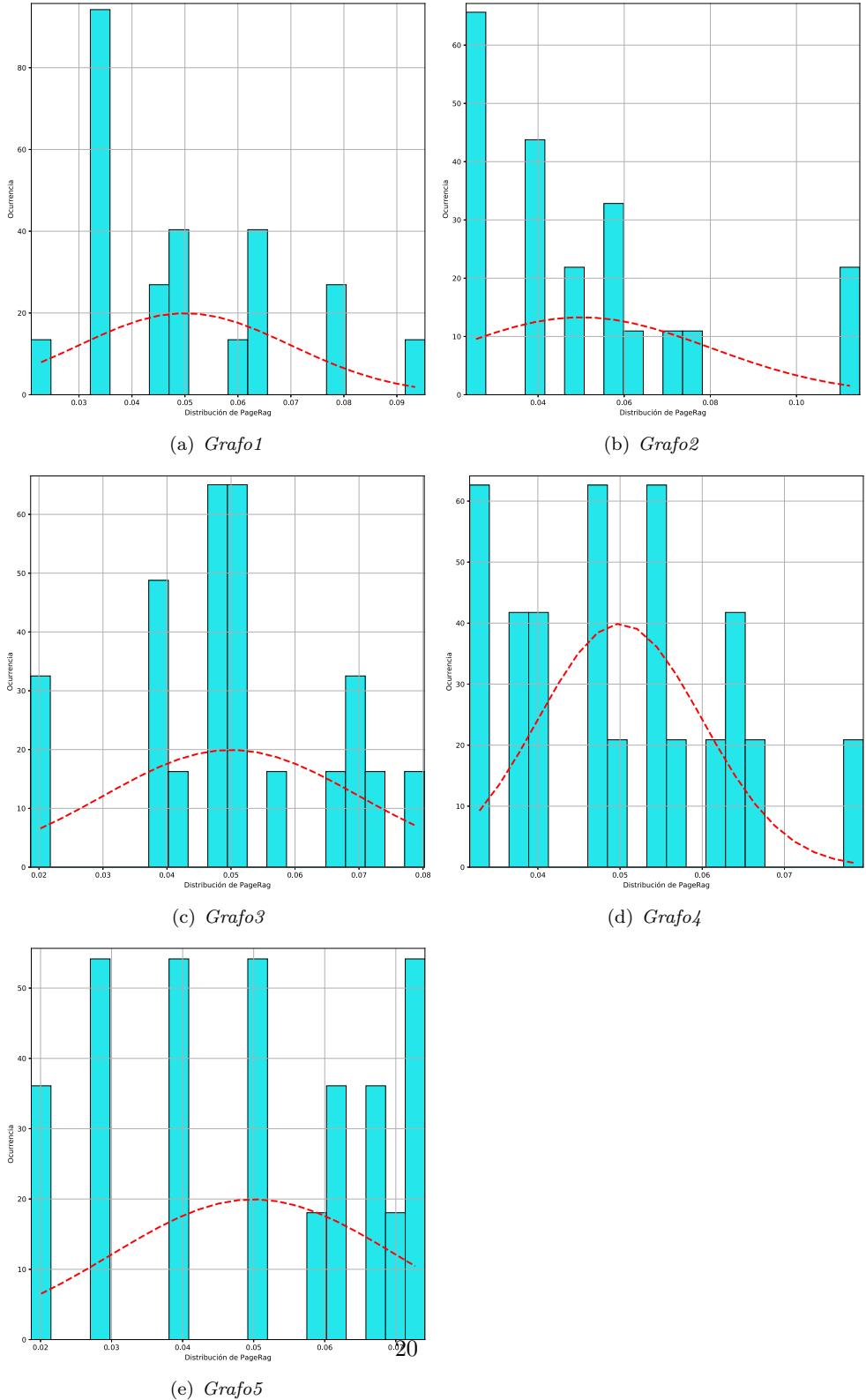


Figura 12: Histogramas que muestran la distribución de los valores de la característica *PageRank*.

```

25     writer = csv.writer(t_csv)
26     writer.writerow(tukey.summary())
27     plt.show()
28
29 modelo = ols('FlujoMax ~ Grado + CoefAg + CentCer + CentCag +
30   Excentricidad + PageRag +
31   ' + Grado*CoefAg + Grado*CentCer+ Grado*CentCag + Grado*
32   Excentricidad + Grado*PageRag +
33   '+CoefAg*CentCer + CoefAg*CentCag + CoefAg*
34   Excentricidad + CoefAg*PageRag +
35   ' + CentCer*CentCag + CentCer*Excentricidad + CentCer*
36   PageRag + CentCag*Excentricidad +
37   ' + CentCag*PageRag + Excentricidad*PageRag' ,data=df)
38 . fit()
39 print(modelo.summary())
40 modelo_csv = open("Anova_MultFlujoMax.csv", 'w')
41 aov_table = sm.stats.anova_lm(modelo, typ=2)
42 df1=pd.DataFrame(aov_table)
43 df1.to_csv("modeloFlujoMax.csv")

```

Analisis_datos1.py

5.1. Análisis de varianza (ANOVA)

El análisis de varianza (ANOVA) es la técnica central en el análisis de datos experimentales. La idea general de esta técnica es separar la variación total en las partes con las que contribuye cada fuente de variación en el experimento. En el caso de los diseños completamente al azar se separan la variabilidad debida a los tratamientos y la debida al error. Cuando la primera predomina sobre la segunda, es cuando se concluye que las medias son diferentes. Cuando los tratamientos no dominan contribuyen igual o menos que el error, por lo que se concluye que las medias son iguales [2]. Para analizar si los diferentes factores (distribución de grado, coeficiente de agrupamiento, centralidad de cercanía, centralidad de carga, excentricidad, *pagerank*) influyen en la variable dependiente *tiempo de ejecución* y valores de flujo máximo se realizó un ANOVA de un factor para cada caso.

5.1.1. Influencia de la distribución de grado en el tiempo de ejecución

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

En el cuadro 3 se muestra que existen diferencia entre las medianas de los grupos de factores ya que el $p - unc$ es menor que 0,05 por lo que se rechaza la hipótesis de que la distribución de grado no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 13 de la página 22.

Cuadro 3: Influencia de la distribución de grado en el tiempo de ejecución (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>Grado</i>	209597.452	2.000	104798.726	662.186	0.000	0.411
<i>Within</i>	300222.666	1897.000	158.262	-	-	-

{ Ubf lem p-unc }

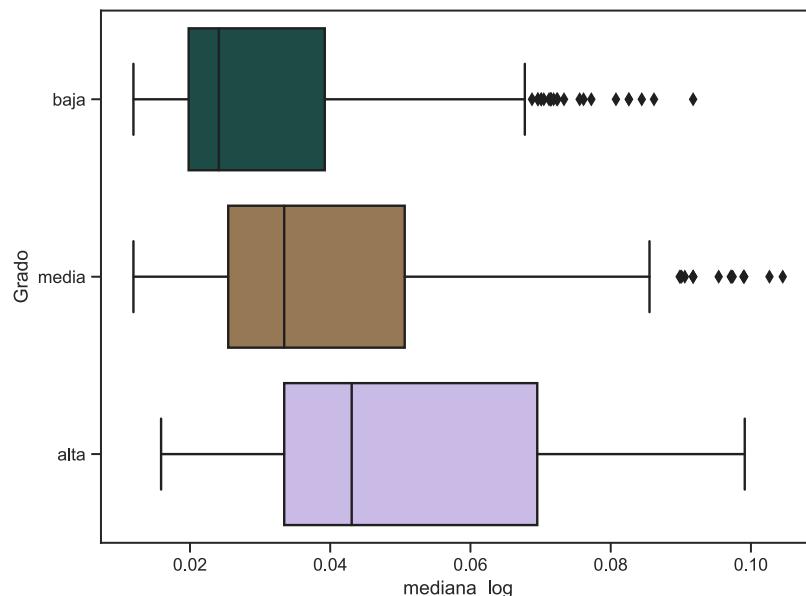


Figura 13: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con la distribución de grado.

En el cuadro 3 se muestra que existen grandes diferencia entre las medianas de los grupos de factores ya que el $p - unc$ es menor que 0,05 por lo que se se rechaza la hipótesis de que la distribución de grado no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 4 de la página 11. Por tal motivo se realiza la prueba de Tukey mostrara las diferencias entre las medianas de los factores, lo que se evidencia en el cuadro 4 de la página 23.

Cuadro 4: Influencia de la distribución de grado en el tiempo de ejecución (*Tukey*)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	-0.020	-0.024	-0.016	True
alta	media	-0.011	-0.015	-0.007	True
baja	media	0.009	0.007	0.011	True

5.1.2. Influencia del coeficiente de agrupamiento en el tiempo de ejecución

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 5: Influencia del coeficiente de agrupamiento en el tiempo de ejecución (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>CoefAg</i>	0.013	2.000	0.006	19.962	0.000	0.021
<i>Within</i>	0.610	1897.000	0.000			

En el cuadro 5 se muestra que existen diferencia entre las medianas de los grupos de factores ya que el $p - unc$ es menor que 0,05 por lo que se se rechaza la hipótesis de que el coeficiente de agrupamiento no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 14 de la página 24.

En el cuadro 5 se muestra que existen marcadas diferencia entre las medianas de dos de los grupos de factores $p - unc$ es menor que 0,05 por lo que se se rechaza la hipótesis de que el coeficiente de agrupamiento no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 14 de la página 24. Por tal motivo se realiza la prueba de Tukey mostrara las diferencias entre las medianas de los factores, lo que se evidencia en el cuadro 6 de la página 24.

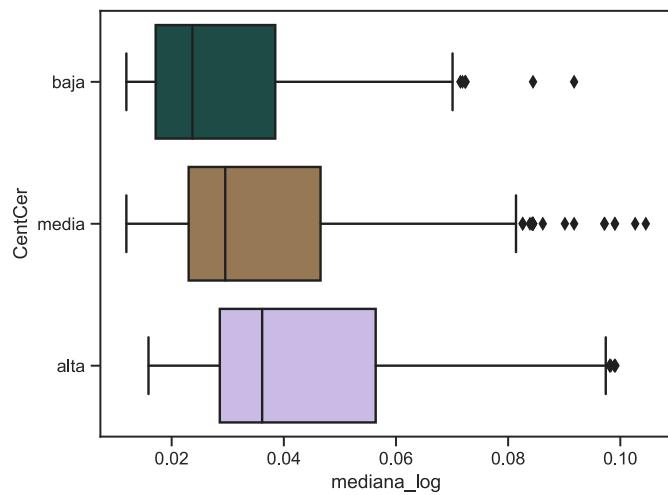


Figura 14: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con el coeficiente de agrupamiento.

Cuadro 6: Influencia del coeficiente de agrupamiento en el tiempo de ejecución (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	0.005	-0.001	0.010	False
alta	media	0.014	0.007	0.020	True
baja	media	0.009	0.005	0.013	True

5.1.3. Influencia de la centralidad de cercanía en el tiempo de ejecución

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 7: Influencia de la centralidad de cercanía en el tiempo de ejecución

Source	SS	DF	MS	F	p-unc	np2
CentCer	0.054	2.000	0.027	89.249	0.000	0.086
Within	0.570	1897.000	0.000			

En el cuadro 7 se muestra que existen diferencia entre las medianas de los grupos de factores ya que el $p - unc$ es menor que 0,05 por lo que se rechaza la hipótesis de que la centralidad de cercanía no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 15 de la página 25.

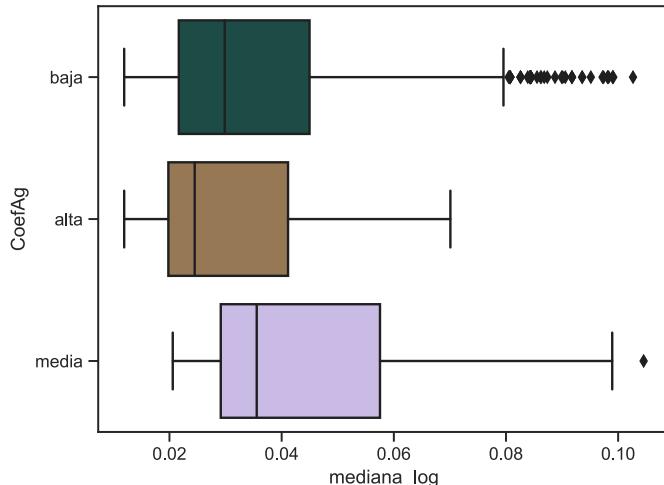


Figura 15: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con la centralidad de cercanía.

En el cuadro 7 se muestra que existen marcadas diferencia entre las medianas de dos de los grupos de factores $p - unc$ es menor que 0,05 por lo que se rechaza la hipótesis de que la centralidad de cercanía no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 15 de la página 25. Por tal motivo se realiza la prueba de Tukey mostrara las diferencias entre las medianas de los factores, lo que se evidencia en el cuadro 8 de la página 26.

Cuadro 8: Influencia de la centralidad de cercanía en el tiempo de ejecución (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	-0.016	-0.018	-0.013	<i>True</i>
alta	media	-0.008	-0.011	-0.006	<i>True</i>
baja	media	0.007	0.005	0.010	<i>True</i>

5.1.4. Influencia de la centralidad de carga en el tiempo de ejecución

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 9: Influencia de la centralidad de carga en el tiempo de ejecución (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>CentCag</i>	0.000	2.000	0.000	0.476	0.621	0.001
<i>Within</i>	0.623	1897.000	0.000			

En el cuadro 9 se muestra que no existen diferencia entre las medianas de los grupos de factores ya que el *p-unc* es mejor que 0,05 por lo que se acepta la hipótesis de que la centralidad de carga no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 16 de la página 27.

5.1.5. Influencia de la excentricidad en el tiempo de ejecución

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 10: Influencia de la excentricidad en el tiempo de ejecución (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>Excentricidad</i>	0.043	2.000	0.021	70.117	0.000	0.069
<i>Within</i>	0.580	1897.000	0.000			

En el cuadro 10 se muestra que existen diferencia entre las medianas de los grupos de factores ya que el *p-unc* es menor que 0,05 por lo que se rechaza la hipótesis de que la excentricidad no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 17 de la página 27.

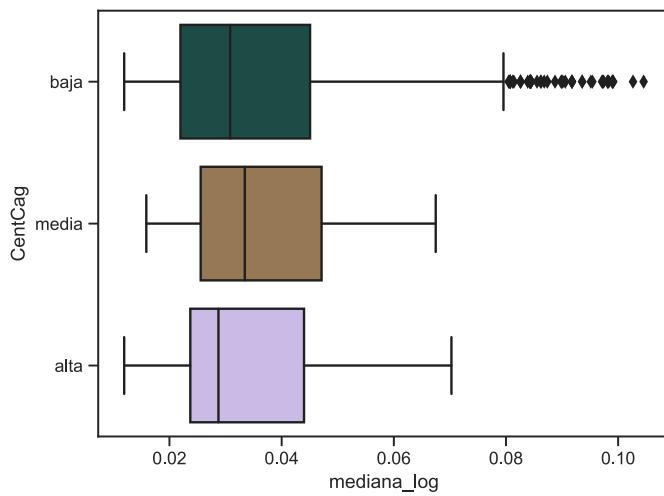


Figura 16: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con la centralidad de carga.

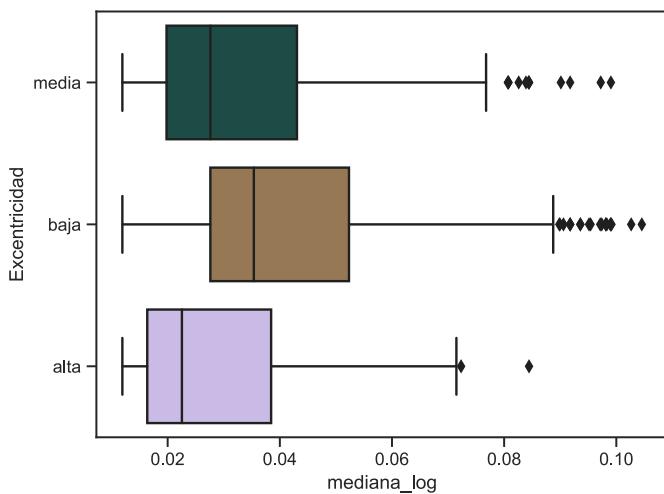


Figura 17: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con la excentricidad.

En el cuadro 10 se muestra que existen diferencia entre las medianas de dos de los grupos de factores $p - unc$ es menor que 0,05 por lo que se rechaza la hipótesis de que la excentricidad no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 17 de la página 27. Por tal motivo se realiza la prueba de Tukey mostrara las diferencias entre las medianas de los factores, lo que se evidencia en el cuadro 11 de la página 28.

Cuadro 11: Influencia de la excentricidad en el tiempo de ejecución (Tukey)

group1	group2	meandiff	lower	upper	reject
alta	baja	0.013	0.010	0.016	True
alta	media	0.004	0.001	0.007	True
baja	media	-0.008	-0.010	-0.006	True

5.1.6. Influencia del *PageRank* en el tiempo de ejecución

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 12: Influencia del *PageRank* en el tiempo de ejecución (ANOVA)

Source	SS	DF	MS	F	p-unc	np2
PageRag	0.016	2.000	0.008	25.358	0.000	0.026
Within	0.607	1897.000	0.000			

En el cuadro 12 se muestra que existen diferencia entre las medianas de los grupos de factores ya que el $p - unc$ es menor que 0,05 por lo que se rechaza la hipótesis de que el *PageRank* no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 18 de la página 29

En el cuadro 12 se muestra que existen diferencia entre las medianas de dos de los grupos de factores $p - unc$ es menor que 0,05 por lo que se rechaza la hipótesis de que el *PageRank* no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 17 de la página 27. Por tal motivo se realiza la prueba de Tukey mostrara las diferencias entre las medianas de los factores, lo que se evidencia en el cuadro 13 de la página 29.

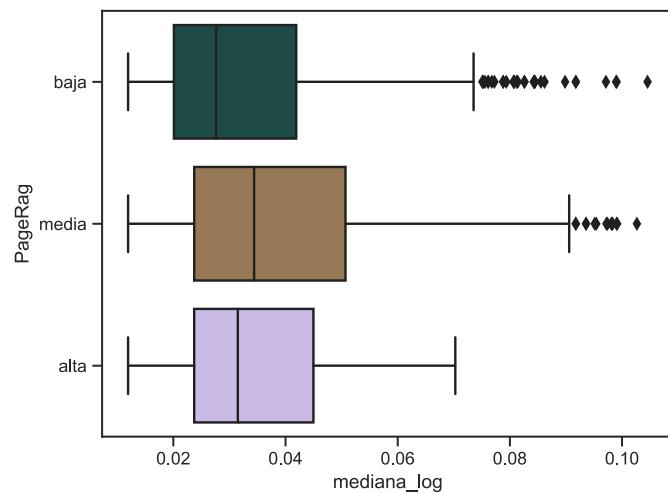


Figura 18: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con el *PageRank*.

Cuadro 13: Influencia del *PageRank* en el tiempo de ejecuciónAdd (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	-0.001	-0.006	0.005	<i>False</i>
alta	media	0.005	-0.001	0.011	<i>False</i>
baja	media	0.006	0.004	0.008	<i>True</i>

5.1.7. Influencia de los seis factores (distribución de grado, coeficiente de agrupamiento, centralidad de cercanía, centralidad de carga, excentricidad, *pagerank*) en el tiempo de ejecución

Para analizar este caso se realizó ANOVA multifactorial dando como resultado el cuadro 14 de la página 30.

Cuadro 14: Influencia de los seis factores (distribución de grado, coeficiente de agrupamiento, centralidad de cercanía, centralidad de carga, excentricidad, *pagerank*) en el tiempo de ejecución (MANOVA)

	<i>sum_sq</i>	<i>df</i>	<i>F</i>	<i>PR(>F)</i>
Grado	0.000	2.000	0.000	1.000
CoefAg	0.000	2.000	0.000	1.000
CentCer	0.000	2.000	-0.409	1.000
CentCag	0.000	2.000	0.000	1.000
Excentricidad	0.000	2.000	0.000	1.000
PageRag	0.000	2.000	0.000	1.000
Grado:CoefAg	0.001	4.000	0.594	0.619
Grado:CentCer	0.001	4.000	1.048	0.381
Grado:CentCag	0.001	4.000	0.854	0.491
Grado:Excentricidad	0.001	4.000	1.064	0.373
Grado:PageRag	0.001	4.000	0.858	0.489
CoefAg:CentCer	0.001	4.000	0.560	0.571
CoefAg:CentCag	0.001	4.000	0.525	0.665
CoefAg:Excentricidad	0.000	4.000	0.203	0.816
CoefAg:PageRag	0.001	4.000	0.430	0.731
CentCer:CentCag	0.001	4.000	0.758	0.517
CentCer:Excentricidad	0.001	4.000	0.543	0.653
CentCer:PageRag	0.001	4.000	0.755	0.519
CentCag:Excentricidad	0.001	4.000	0.584	0.626
CentCag:PageRag	0.001	4.000	0.640	0.527
Excentricidad:PageRag	0.001	4.000	0.677	0.608
Residual	0.549	1876.000		

En el cuadro 7 se puede observar que los factores que más influyen en el tiempo de ejecución son el número de vértices y la densidad de los grafos, que además existe una relación entre el número de nodos y la densidad de los grafos.

5.1.8. Influencia de las seis propiedades en el flujo máximo

Los cuadro 15, 16, 17, 18, 19, 19 de las páginas 31, 31, 31, 31, 31, 31 respectivamente, muestran los resultados de la aplicación del ANOVA en las seis propiedades con respecto los valores de flujo máximo.

Cuadro 15: Influencia de la distribución de grado en el flujo máximo (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>Grado</i>	209597.452	2.000	104798.726	662.186	0.000	0.411
<i>Within</i>	300222.666	1897.000	158.262			

Cuadro 16: Influencia de la distribución de coeficiente de agrupamiento en el flujo máximo (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>CoefAg</i>	59963.814	2.000	29981.907	126.431	0.000	0.118
<i>Within</i>	449856.304	1897.000	237.141			

Cuadro 17: Influencia de la distribución de centralidad de cercanía en el flujo máximo (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>CentCer</i>	256496.045	2.000	128248.023	960.377	0.000	0.503
<i>Within</i>	253324.073	1897.000	133.539			

Cuadro 18: Influencia de la centralidad de carga en el flujo máximo (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>CentCag</i>	6469.019	2.000	3234.510	12.190	0.000	0.013
<i>Within</i>	503351.098	1897.000	265.341			

Cuadro 19: Influencia de la excentricidad en el flujo máximo (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
Excentricidad	200853.698	2.000	100426.849	616.603	0.000	0.394
<i>Within</i>	308966.420	1897.000	162.871	-	-	-

Cuadro 20: Influencia de el *pagerank* en el flujo máximo (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>PageRag</i>	67070.251	2.000	33535.125	143.684	0.000	0.132
<i>Within</i>	442749.867	1897.000	233.395			

En los cuadro anteriores se muestra que existen diferencia entre las medianas de los grupos de factores ya que el $p - unc$ es menor que 0,05 por lo que se rechaza la hipótesis de que las propiedades no influye en el *flujo máximo*. Esto se puede observar claramente en las figuras 19, 20, 21, 22, 23, 23 de las páginas 32, 33, 33, 34, 35 respectivamente.

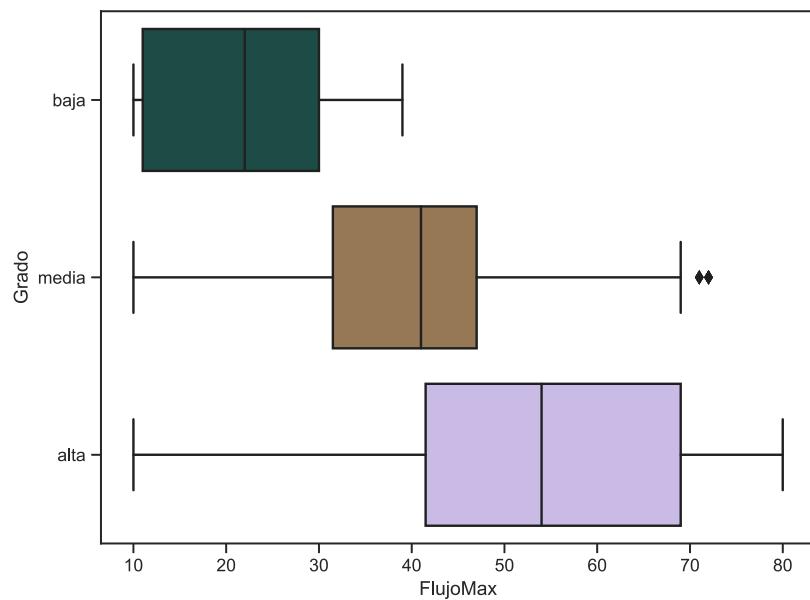


Figura 19: Diagrama de caja y bigotes que relaciona los flujos máximos con la distribución de grado.

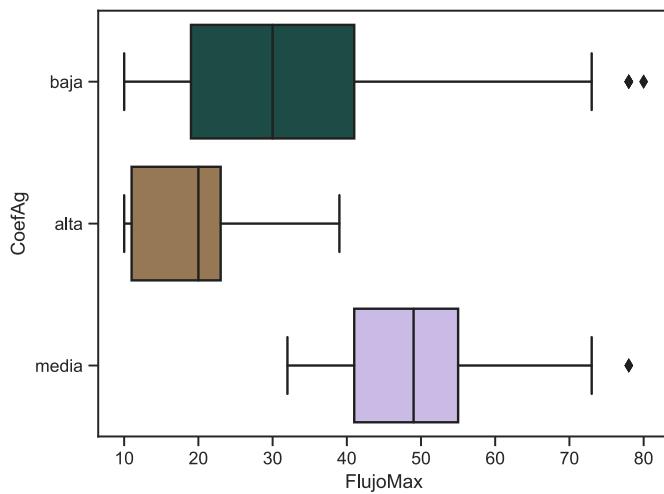


Figura 20: Diagrama de caja y bigotes que relaciona los flujos máximos con el coeficiente de agrupamiento.

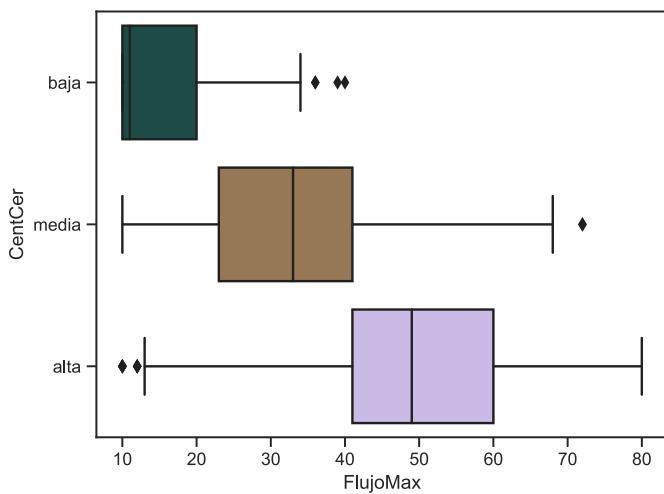


Figura 21: Diagrama de caja y bigotes que relaciona los flujos máximos con la centralidad de cercanía.

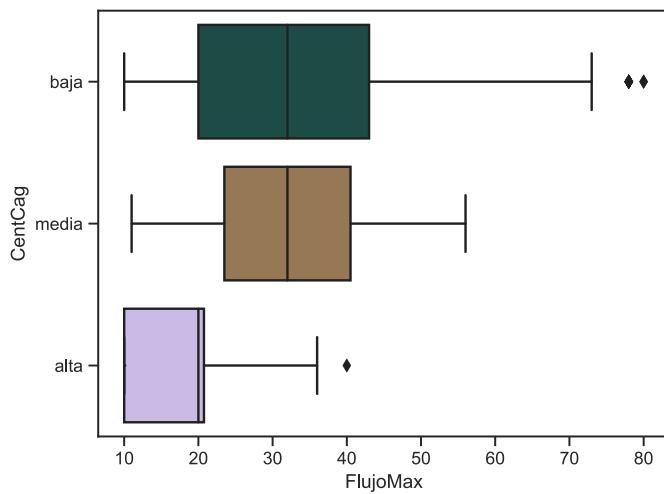


Figura 22: Diagrama de caja y bigotes que relaciona los flujos máximos con la centralidad de carga.

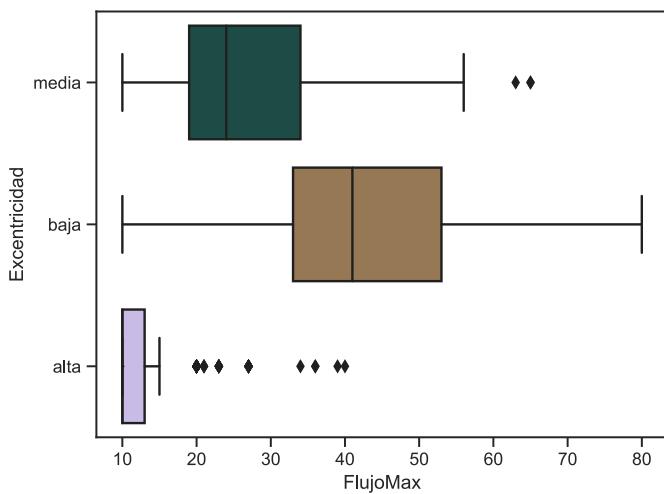


Figura 23: Diagrama de caja y bigotes que relaciona los flujos máximos con la excentricidad.

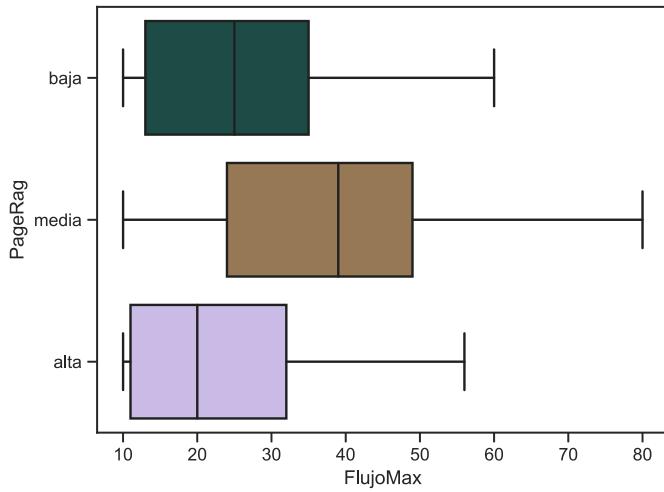


Figura 24: Diagrama de caja y bigotes que relaciona los flujos máximos con el *pagerank*.

Cuadro 21: Influencia de la distribución de grado en el flujo máximo (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	-33.721	-36.659	-30.782	<i>True</i>
alta	media	-15.682	-18.613	-12.751	<i>True</i>
baja	media	18.039	16.642	19.436	<i>True</i>

Cuadro 22: Influencia de el coeficiente de agrupamiento en el flujo máximo (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	11.253	6.389	16.118	<i>True</i>
alta	media	30.351	24.741	35.961	<i>True</i>
baja	media	19.098	16.039	22.156	<i>True</i>

Cuadro 23: Influencia de la centralidad de cercanía en el flujo máximo (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	-33.901	-35.715	-32.086	<i>True</i>
alta	media	-17.016	-18.568	-15.463	<i>True</i>
baja	media	16.885	15.355	18.415	<i>True</i>

Cuadro 24: Influencia de la centralidad de carga en el flujo máximo (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	13.165	6.904	19.427	<i>True</i>
alta	media	13.816	3.080	24.551	<i>True</i>
baja	media	0.651	-8.160	9.461	<i>False</i>

Cuadro 25: Influencia de la excentricidad en el flujo máximo (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	29.093	26.913	31.272	<i>True</i>
alta	media	12.855	10.692	15.017	<i>True</i>
baja	media	-16.238	-17.711	-14.764	<i>True</i>

Debido a las diferencias obtenidas en todas las ANOVAs realizadas, procedemos a realizar la prueba de Tukey cuyos resultados se muestran en los cuadros

En los cuadros 21, 22, 23, 24, 25, 26 de las páginas 35, 35, 35, 36, 36, 36 se pueden observar claramente que las seis características influye en el valor máximo de flujo.

5.1.9. Influencia de los seis factores (~~distribución de grado, coeficiente de agrupamiento, centralidad de cercanía, centralidad de carga, excentricidad, pagerank~~) en el flujo máximo

Para analizar este caso se realizó ANOVA multifactorial dando como resultado el cuadro 27 de la página 37.

En el cuadro 27 de la página 37 se observa claramente que todos las características influye en los valores del flujo máximo corroborando lo arrojado en las pruebas anteriores.

Cuadro 26: Influencia del *pagerank* en el flujo máximo (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	3.867	-0.998	8.733	<i>False</i>
alta	media	15.846	10.916	20.776	<i>True</i>
baja	media	11.979	10.269	13.689	<i>True</i>

Cuadro 27: Influencia de los seis factores (distribución de grado, coeficiente de agrupamiento, centralidad de cercanía, centralidad de carga, excentricidad, *pagerank*) en el flujo máximo (MANOVA)

	<i>sum_sq</i>	<i>df</i>	<i>F</i>	<i>PR(>F)</i>	\leq
Grado	0.001	2.000	0.000	1.000	
CoefAg	-45.503	2.000	-0.210	1.000	
CentCer	2523.680	2.000	11.667	0.000	
CentCag	0.000	2.000	0.000	1.000	
Excentricidad	0.000	2.000	0.000	1.000	
PageRag	0.000	2.000	0.000	1.000	
Grado:CoefAg	34.118	4.000	0.079	0.971	
Grado:CentCer	37.572	4.000	0.087	0.987	
Grado:CentCag	54.632	4.000	0.126	0.973	
Grado:Excentricidad	35.792	4.000	0.083	0.988	
Grado:PageRag	59.651	4.000	0.138	0.968	
CoefAg:CentCer	55.193	4.000	0.128	0.880	
CoefAg:CentCag	38.260	4.000	0.088	0.966	
CoefAg:Excentricidad	34.182	4.000	0.079	0.924	
CoefAg:PageRag	45.273	4.000	0.105	0.957	
CentCer:CentCag	46.187	4.000	0.107	0.956	
CentCer:Excentricidad	41.053	4.000	0.095	0.963	
CentCer:PageRag	74.952	4.000	0.173	0.915	
CentCag:Excentricidad	90.135	4.000	0.208	0.891	
CentCag:PageRag	79.931	4.000	0.185	0.831	
Excentricidad:PageRag	51.771	4.000	0.120	0.976	
Residual	202895.688	1876.000			

6. Conclusiones

Después del análisis realizado se concluyó que las características estructurales sí influye tanto en el tiempo de ejecución como en el valor máximo de flujo , en los grafos generados para esta tarea la características que influyen en mayor medida es la distribución de grado y *pagerank*.

Referencias

- [1] Bartłomiej Bosek, Dariusz Leniowski, Piotr Sankowski, and Anna Zych-Pawlewicz. Shortest augmenting paths for online matchings on trees. *Theory of Computing Systems*, 62(2):337–348, Feb 2018.
- [2] Humberto Gutiérrez and Román de la Vara. *Análisis y diseño de experimentos*. The McGraw-Hill Companies, Inc., segunda edición edition, 2008. 60–74.
- [3] Desarrolladores NetworkX. https://networkx.github.io/documentation/stable/reference/generated/networkx.generators.random_graphs.erdos_renyi_graph.html#networkx.generators.random_graphs.erdos_renyi_graph. Accessed: 01-04-2019.
- [4] Desarrolladores NetworkX. https://networkx.github.io/documentation/stable/reference/generated/networkx.generators.random_graphs.fast_gnp_random_graph.html#networkx.generators.random_graphs.fast_gnp_random_graph. Accessed: 01-04-2019.
- [5] Desarrolladores NetworkX. https://networkx.github.io/documentation/stable/reference/generated/networkx.generators.random_graphs.binomial_graph.html#networkx.generators.random_graphs.binomial_graph. Accessed: 01-04-2019.
- [6] Desarrolladores NetworkX. https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.flow.maximum_flow.html#networkx.algorithms.flow.maximum_flow. Accessed: 01-04-2019.
- [7] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.coloring.greedy_color.html. Accessed: 18-03-2019.
- [8] Sadegh Nobari, Xuesong Lu, Panagiotis Karras, and Stéphane Bressan. Fast random graph generation. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT ’11, pages 331–342, New York, NY, USA, 2011. ACM.

Tarea 5

5271

26 de mayo de 2019

1. Algoritmo generador de grafos

Gracias a la versatilidad de los grafos como modelo de representación de datos, los procesos aleatorios de generación de grafos también son relevantes en aplicaciones que van desde la física, biología a la sociología [8].

Otra de las aplicaciones de los grafos de generación aleatoria es en la representación de una red telefónica donde los vértices son centrales telefónicas, las aristas son los enlaces troncales los cuales presentan cierta capacidad de llamadas. De esta red se desea conocer el flujo máximo entre una central telefónica fuente y una central telefónica sumidero.

En realización de la tarea anterior se seleccionaron tres algoritmos generadores de grafos de la biblioteca de *networkx*, con el objetivo de crear los grafos con pesos con distribución normal a los que se le aplicaron los tres algoritmos de flujo máximo para realizar los experimentos. Los tres algoritmos generadores antes mencionados son los siguientes.

Los algoritmos son los siguientes:

- *Erdős Rényi graph* (en el modelo $G(n, p)$, el grafo se construye conectando los n vértices al azar. Cada arista se incluye en el grafo con probabilidad p independiente de cualquier otro borde) [3].
- *Fast gnp random graph* (este algoritmo recibe como parámetros n números de vértices y probabilidad p de ocurrencia de aristas, el mismo devuelve un grafo aleatorio) [4].
- *Binomial graph* (este algoritmo recibe como parámetros n números de vértices y probabilidad p de ocurrencia de aristas, el mismo devuelve un grafo aleatorio, para grafos dispersos (para valores pequeños de p), *Fast gnp random graph* es un algoritmo más rápido) [5].

De los algoritmos anteriores el seleccionado para la realización de esta tarea es el *Erdős Rényi graph* dado que la tarea anterior arroja que ninguno de los tres influían en el tiempo de ejecución de los algoritmo de flujo máximo que se le aplicaron a los grafos generados. como se muestra en el cuadro 1, es decir que los tres generan grafos con características similares.

Cuadro 1: ANOVA, relación del algoritmo generador con el *tiempo de ejecución*

Factor	SS	DF	MS	F	p-unc	np2
generador-grafo	0.280	2	0.140	0.354	0.702	0
Within	712.085	1797	0.396	-	-	-

En el cuadro 1 se muestra que no existen diferencia entre las medianas de los grupos de factores ya que el **p-unc** es mayor que 0,05 por lo que se acepta la hipótesis de que el tipo de generador no influye en el *tiempo de ejecución*.

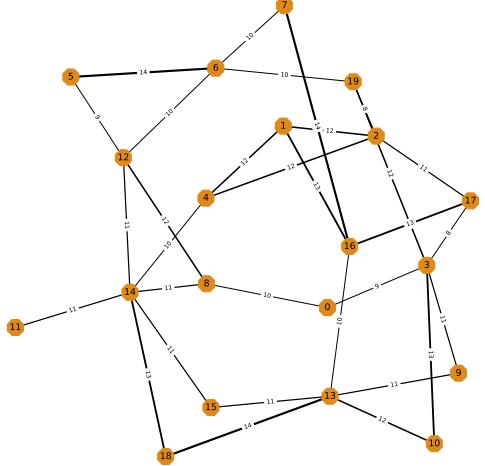
Para la generación de los grafos con el algoritmo *Erdős Rényi graph* hizo el siguiente código de Python donde se generan cinco grafos con la misma cantidad de vértices pero con diferentes números de aristas y capacidades de las mismas como se muestra en la figura 1 de la página 3.

```

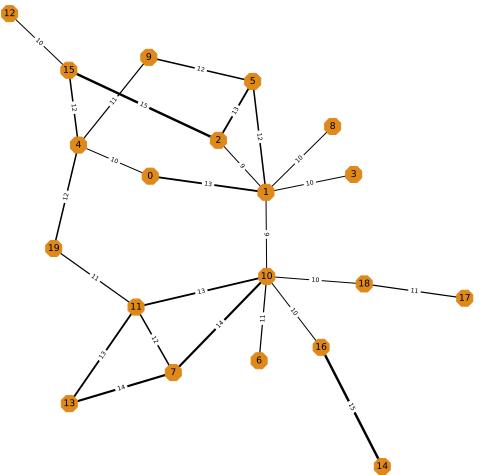
1 def leer_capacity(G):
2     capacity=[]
3     for (u, v) in G.edges():
4         capacity.append(G.edges[u, v]["capacity"])
5     return capacity
6
7 Grafo = nx.erdos_renyi_graph(20, 0.19, seed=None)
8 aristas = Grafo.number_of_edges()
9 pesos_normalmente_distribuidos = np.random.normal(12, 1.5, aristas)
10 loop = 0
11 for (u, v) in Grafo.edges():
12     Grafo.edges[u, v]["capacity"] = pesos_normalmente_distribuidos[loop]
13     loop += 1
14 df = pd.DataFrame()
15 df = nx.to_pandas_adjacency(Grafo, dtype=int, weight='capacity')
16 df.to_csv("Grafo5" + ".csv", index=None, header=None)

```

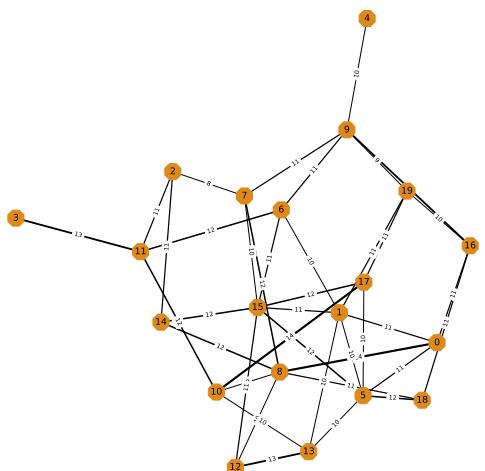
Generar-g.py



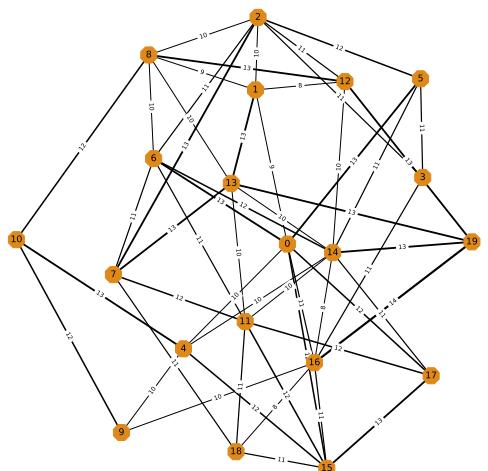
(a) *Grafo1*, con 20 vértices



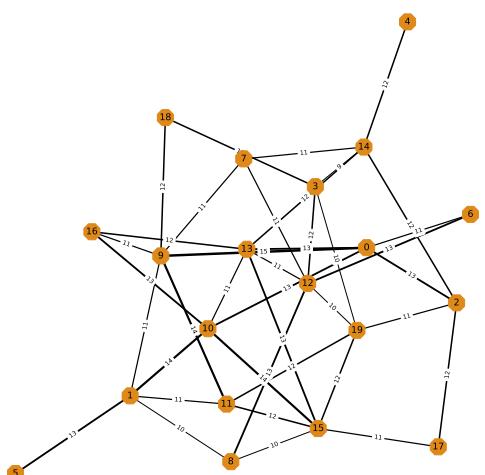
(b) *Grafo2*, con 20 vértices



(c) *Grafo3*, con 20 vértices



(d) *Grafo4*, con 20 vértices



(e) *Grafo5*, con 20 vértices

Figura 1: Grafos generados con el algoritmo seleccionado, donde el grosor de las aristas representan la capacidad de las mismas.

2. Algoritmo de flujo máximo

En los problemas de flujo en redes, las aristas representan vías por las que puede circular elementos: datos, agua, corriente eléctrica, llamadas telefónicas entre otras. Los pesos de las aristas representan la capacidad máxima de una vía: velocidad de una conexión, volumen máximo de agua, voltaje de una línea eléctrica, cantidad máxima llamadas entre otras; aunque es posible que la cantidad real de flujo sea menor.

El problema del flujo máximo consiste en lo siguiente: dado un grafo con pesos, $G = (V, A, W)$, que representa las capacidades máximas de los canales, un vértice fuente f y otro sumidero s en V , encontrar la cantidad máxima de flujo que puede circular desde f hasta s .

En la biblioteca de *networkx* encontramos varios algoritmos con los que podemos atacar los problemas de flujo máximo, para la realización de esta tarea se escogerá uno de los tres algoritmos utilizados en la tarea anterior pertenecientes a dicha librería. Basándonos en los resultados obtenidos en las pruebas realizadas en la tarea anterior.

Los algoritmos escogidos en la tarea anterior son los siguientes:

- *Shortest augmenting path* es uno de los enfoques más clásicos para la máxima coincidencia y los problemas de flujo máximo. Sorprendentemente, aunque esta idea es una de las técnicas más básicas, está lejos de ser completamente entendida. Es más fácil hablar de ello introduciendo el problema de emparejamiento bipartito en línea [1]. Este algoritmo encuentra el flujo máximo de un solo producto utilizando la ruta de aumento más corto y devuelve la red residual resultante después de calcular el flujo máximo.
- *Maximum flow* encuentra la ruta por la cual pasa la máxima cantidad de flujo, recibe como parámetros un grafo G , una fuente f , un sumidero s y además una capacidad que de no tenerla, se considera que el borde tiene una capacidad infinita. Se puede aplicar en grafos tanto dirigidos como no dirigidos [6].
- *Preflow push* encuentra un flujo máximo de un solo producto utilizando el algoritmo de empuje previo al flujo de la etiqueta más alta. Esta función devuelve la red residual resultante después de calcular el flujo máximo. Este algoritmo tiene un tiempo de ejecución de $O(n^2\sqrt{m})$ para n vértices y m aristas [7].

De los algoritmos anteriores el que se utiliza en esta tarea es el *Shortest augmenting path* por ser el algoritmo que mejores tiempos registró en la tarea anterior.

3. Algoritmo de acomodo

El algoritmo *Kamada kawai layout* posiciona los nodos utilizando la función de costo de longitud de camino *Kamada-Kawai*, dando la mejor visualización de los grafos generados.

4. Generación de datos

Con el objetivo de realizar las mediciones de los tiempos de ejecución de los algoritmo de flujo máximo seleccionado así como el calculo de las seis propiedades de los nodos (distribución de grado, coeficiente de agrupamiento, centralidad de cercanía, centralidad de carga, excentricidad, *pagerank*) se desarrolló el siguiente código.

En primer lugar, se crea una función (*Maxflow()*) que recibe como parámetros el grafo al que se le aplicara el algoritmo (G), la fuente (**a**) y sumidero (**b**). Al llamar esta función se genera con cada ejecución los datos que son guardados en un *data frame* del cual se muestra un fragmento en el cuadro 2 de la página 5.

Cuadro 2: Fragmento del *data frame* que contiene los datos recopilados.

Grafo	Fuente	Sumidero	Media	Mediana	FlujoMax	Grado	CoefAg	CentCer	CentCag	Excent	PageRag
1	0	1	0.026	0.020	19	2	0.000	0.388	0.048	4	0.035
1	15	7	0.022	0.020	22	2	0.000	0.413	0.035	4	0.034
2	0	9	0.029	0.020	23	2	0.000	0.396	0.044	4	0.039
2	2	4	0.028	0.024	34	3	0.333	0.404	0.123	4	0.056
3	1	2	0.044	0.052	30	6	0.267	0.543	0.084	3	0.068
3	1	15	0.042	0.024	63	6	0.267	0.543	0.084	3	0.068
4	0	13	0.079	0.084	69	7	0.143	0.613	0.096	2	0.064
4	3	10	0.055	0.032	33	3	0.333	0.475	0.016	3	0.032
5	3	15	0.020	0.020	43	4	0.167	0.463	0.059	4	0.052
5	8	2	0.076	0.074	33	3	0.000	0.452	0.034	4	0.039

En este fragmento del código se realiza toda la recopilación de la información necesaria para el análisis de los factores que in fluyen en el tiempo de ejecución y el flujo máximo, también dibuja los grafos a los que se les aplicó el algoritmo.

```

1 def Leer_grafos(nomb):
2     dr = pd.read_csv((nomb+".csv"), header=None)
3     A = nx.from_pandas_adjacency(dr)
4     return A
5
6 def leer_capacity(G):
7     capacity=[]
8     for (u, v) in G.edges():
9         capacity.append(G.edges[u, v]["weight"])
10    return capacity
11
12
13 def Maxflow(G, a, b):
14     tiempo_inicial = dt.datetime.now()
15     # d =shortest_augmenting_path(G, a, b, capacity="weight")
16     for i in range(40):
17         d =shortest_augmenting_path(G, a, b, capacity="weight")
18         print(dt.datetime.now())
19     tiempo_final = (dt.datetime.now() - tiempo_inicial).total_seconds()
20     return tiempo_final
21
22
23 def listnodos(d,f,s):
24     for nodes in d.nodes():
25         if d.nodes[nodes]!=f and d.nodes[nodes]!=s :
26             nodos.append(nodes)
27     return nodos
28 def Tiempos(G,nombre):
29     df = {"Grafo": [], "Fuente": [], "Sumidero": [], "Media": [], "Mediana": [], "DesvStd": [], "flujoMax": [],
30           [], "Grado": [], "CoefAg": [], "CentCer": [], "CentCag": [], "Excentricidad": [], "PageRag": []}
31     Nodes = G.nodes
32     print(Nodes)
33     for i in Nodes:
34         for j in Nodes:
35             if i != j:
36                 t = []
37                 for k in range(10):
38                     print(t, end="\n\n")
39                     time = Maxflow(G, i, j)
40                     t.append(time)
41                 PageRag = nx.pagerank(G, weight="capacity")
42                 df["Grafo"].append(nombre)
43                 df["Grado"].append(G.degree(i))
44                 df["CoefAg"].append(round(nx.clustering(G, i), 4))
45                 df["CentCer"].append(round(nx.closeness_centrality
46                                         (G, i), 4))
47                 df["CentCag"].append(round(nx.load_centrality
48                                         (G, i), 4))
49                 df["Excentricidad"].append(nx.eccentricity(G, i))
50                 df["PageRag"].append(round(PageRag[i], 4))
51                 df["Fuente"].append(i)
52                 df["Sumidero"].append(j)
```

```

53 df[ "Media" ].append( round( np.mean(t), 4) )
54 df[ "Mediana" ].append( round( np.median(t), 4) )
55 df[ "DesvStd" ].append( round( np.std(t), 4) )
56 df[ "flujoMax" ].append( nx.maximum_flow_value
57 (G, i, j, capacity="weight"))
58
59 dd = pd.DataFrame(df)
60 dd.to_csv(nombre+"D.csv", index=None)
61 G = Leer_grafos("Grafo1")
62 widths = []
63 widths = leer_capacity(G)
64 print(widths)
65 widths[:] = [(x - 7)/2 for x in widths]
66 plt.figure(figsize=(15, 15))
67 labels = {}
68 for u, v, data in G.edges(data=True):
69     labels[(u, v)] = data[ 'weight' ]
70 position = nx.kamada_kawai_layout(G, dist=None, pos=None, scale=0.5, center=None, dim=2)
71 nx.draw_networkx_nodes(G, position, node_size=800, node_color="#de8919", node_shape="8")
72 nx.draw_networkx_edges(G, position, width=widths, edge_color='black')
73 nx.draw_networkx_edge_labels(G, position, edge_labels=labels, font_size=10, label_pos=.5)
74 nx.draw_networkx_labels(G, position, font_size=15)
75
76 df = pd.DataFrame(position)
77 df.to_csv("position1" + ".csv", index=None, header=None)
78 plt.axis("off")
79 plt.savefig("Grafola" + ".png", bbox_inches='tight')
80 plt.savefig("Grafola" + ".eps", bbox_inches='tight')
81 plt.show(G)
82
83 d = shortest_augmenting_path(G, 4, 8, capacity="weight")
84 flowma = []
85 widths1 = []
86 widths0 = []
87 widths2 = []
88 widths3 = []
89 maxi = 0
90 nodos = []
91 nodosF = [4]
92 nodosS = [8]
93 nodos = listnodos(d, 4, 8)
94 for edges in d.edges():
95     widths.append(d.edges[edges][ "flow" ])
96     if d.edges[edges][ "flow" ] > 0:
97         widths1.append(d.edges[edges][ "capacity" ])
98         widths0.append(edges)
99         flowma.append(d.edges[edges][ 'flow' ])
100
101 elif d.edges[edges][ "flow" ] == 0:
102     widths2.append(edges)
103     widths3.append(d.edges[edges][ "capacity" ])
104 print(widths1)
105 print(widths3)
106 flowma[:] = [x + 20 for x in flowma]
107 for i in flowma:
108     if i > maxi:
109         maxi = i
110 widths[:] = [x/10*x/6 for x in widths]
111 widths1[:] = [(x - 7)/2 for x in widths1]
112 widths3[:] = [(x - 7)/2 for x in widths3]
113 plt.figure(figsize=(15, 15))
114 position = pd.read_csv('position1.csv', header=None)
115 for u, v, data in d.edges(data=True):
116     if data[ 'flow' ] >= 0:
117         labels[(u, v)] = data[ 'flow' ]
118 nx.draw_networkx_nodes(d, position, nodelist=nodos, node_size=800, node_color="#de8919", cnode_shape="8")
119 nx.draw_networkx_nodes(d, position, nodelist=nodosF, node_size=800, node_color="red", cnode_shape="8")
120 nx.draw_networkx_nodes(d, position, nodelist=nodosS, node_size=800, node_color="green", cnode_shape="8")
121
122 nx.draw_networkx_edges(d, position, edgelist=widths2, edge_color='black', width=widths3, arrows=False)
123 nx.draw_networkx_edges(d, position, edgelist=widths0, edge_cmap=plt.cm.Purples, width=widths1,
124 edge_color=flowma,
```

```

124         edge_vmin=0, edge_vmax=maxi)
125
126 nx.draw_networkx_edge_labels(d, position , edge_labels=labels , font_size=10, label_pos=.5)
127 nx.draw_networkx_labels(d, position , font_size=15)
128 plt.axis("off")
129
130
131 plt.savefig("Grafo1cf" + ".png", bbox_inches='tight')
132 plt.savefig("Grafo1cf" + ".eps", bbox_inches='tight')
133
134 plt.show(G)
135 list=[ "Grafo1" , "Grafo2" , "Grafo3" , "Grafo4" , "Grafo5" ]
136
137 for k in list:
138     print(k)
139     l = Leer_grafos(k)
140     Tiempos(l ,k)
141

```

Leer_grafo.py

La aplicación del algoritmo de flujo máximo *Shortest augmenting path* a todas las posibles combinaciones de fuentes y sumidero de cada uno de los cinco grafos nos dio como resultado los valores de flujo máximo para cada una de estas combinaciones y nos muestra cómo va variando el óptimo desde la peor combinación fuente sumidero a la mejor.

Para el Grafo1 esto se muestra en la figura 2 de la página 8.

Para el Grafo2 esto se muestra en la figura 3 de la página 9.

Para el Grafo3 esto se muestra en la figura 4 de la página 10.

Para el Grafo4 esto se muestra en la figura 5 de la página 11.

Para el Grafo5 esto se muestra en la figura 6 de la página 12.

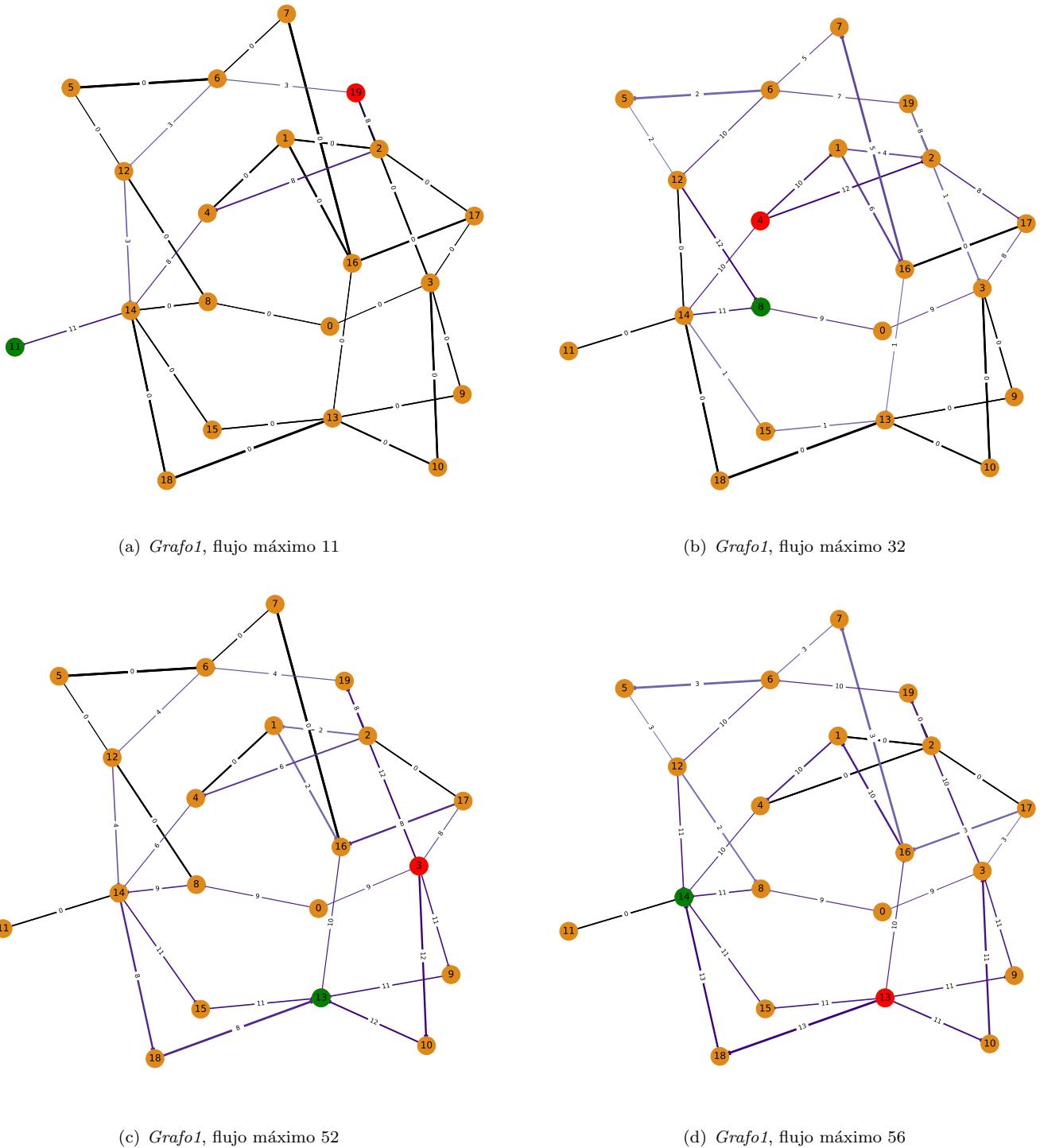
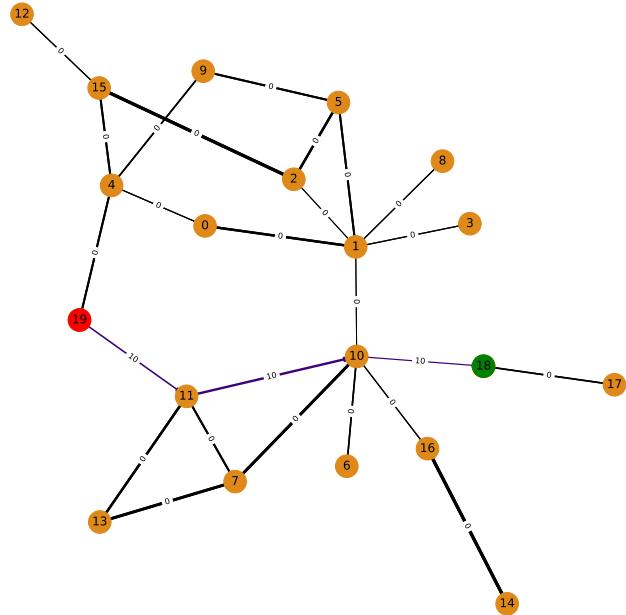
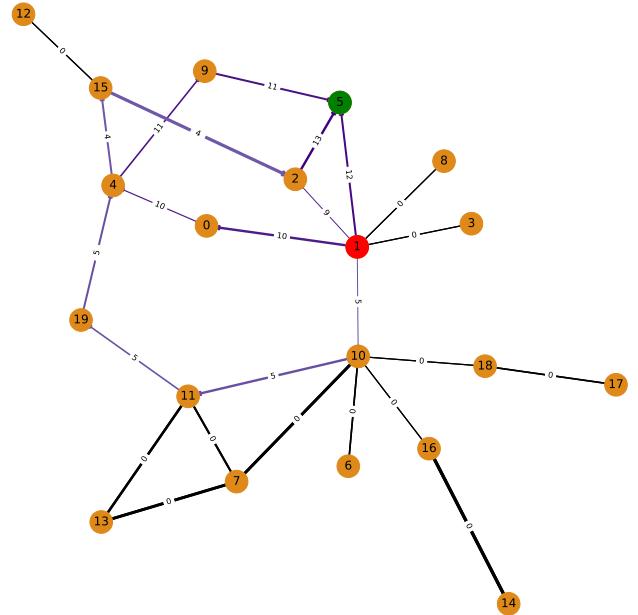


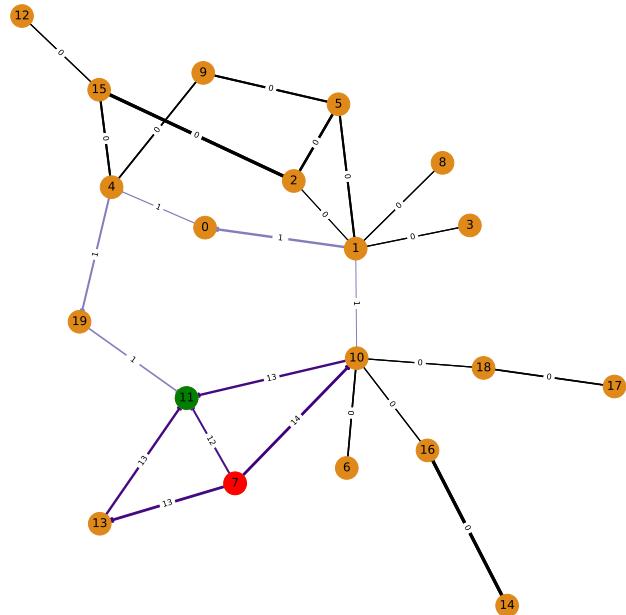
Figura 2: Grafos resultantes de la aplicación del algoritmo de flujo máximo, donde el grosor de las aristas representa la capacidad de las mismas, el color negro la ausencia de flujo, el color violeta la presencia de flujo y la intensidad del color violeta la cantidad de flujo que pasa por la arista. El color rojo del vértice representa la fuente y el vértice verde el sumidero.



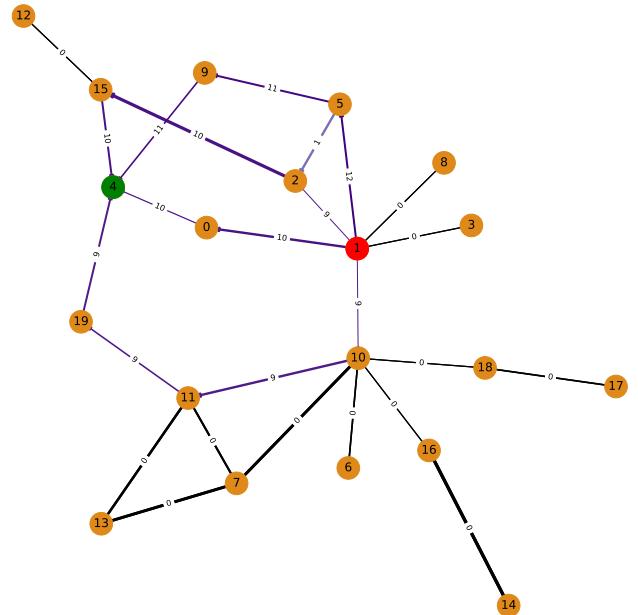
(a) *Grafo2*, flujo máximo 10



(b) *Grafo2*, flujo máximo 36



(c) *Grafo2*, flujo máximo 39



(d) *Grafo2*, flujo máximo 40

Figura 3: Grafos resultantes de la aplicación del algoritmo de flujo máximo, donde el grosor de las aristas representa la capacidad de las mismas, el color negro la ausencia de flujo, el color violeta la presencia de flujo y la intensidad del color violeta la cantidad de flujo que pasa por la arista. El color rojo del vértice representa la fuente y el vértice verde el sumidero.

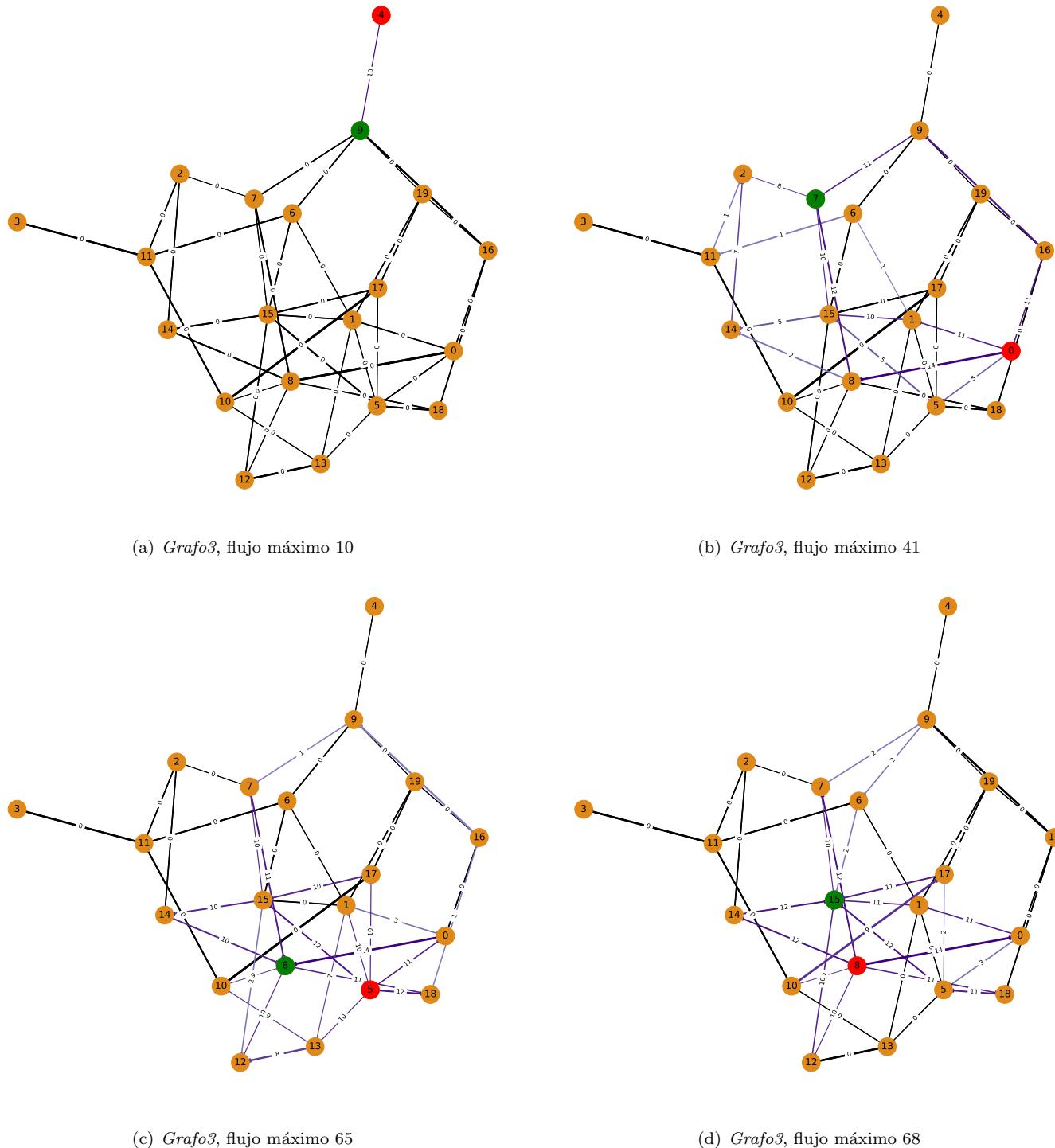


Figura 4: Grafos resultantes de la aplicación del algoritmo de flujo máximo, donde el grosor de las aristas representa la capacidad de las mismas, el color negro la ausencia de flujo, el color violeta la presencia de flujo y la intensidad del color violeta la cantidad de flujo que pasa por la arista. El color rojo del vértice representa la fuente y el vértice verde el sumidero.

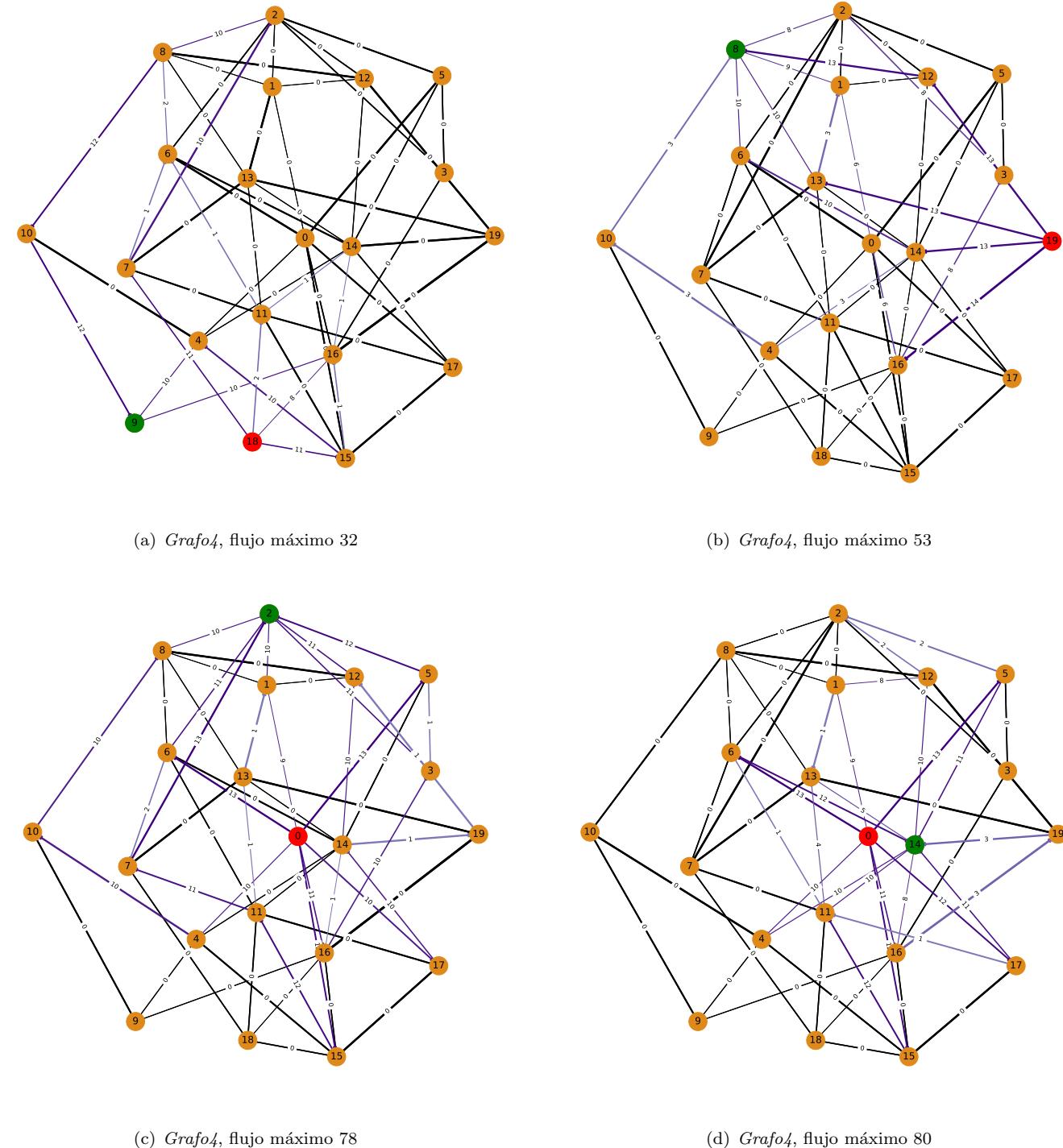


Figura 5: Grafos resultantes de la aplicación del algoritmo de flujo máximo, donde el grosor de las aristas representa la capacidad de las mismas, el color negro la ausencia de flujo, el color violeta la presencia de flujo y la intensidad del color violeta la cantidad de flujo que pasa por la arista. El color rojo del vértice representa la fuente y el vértice verde el sumidero.

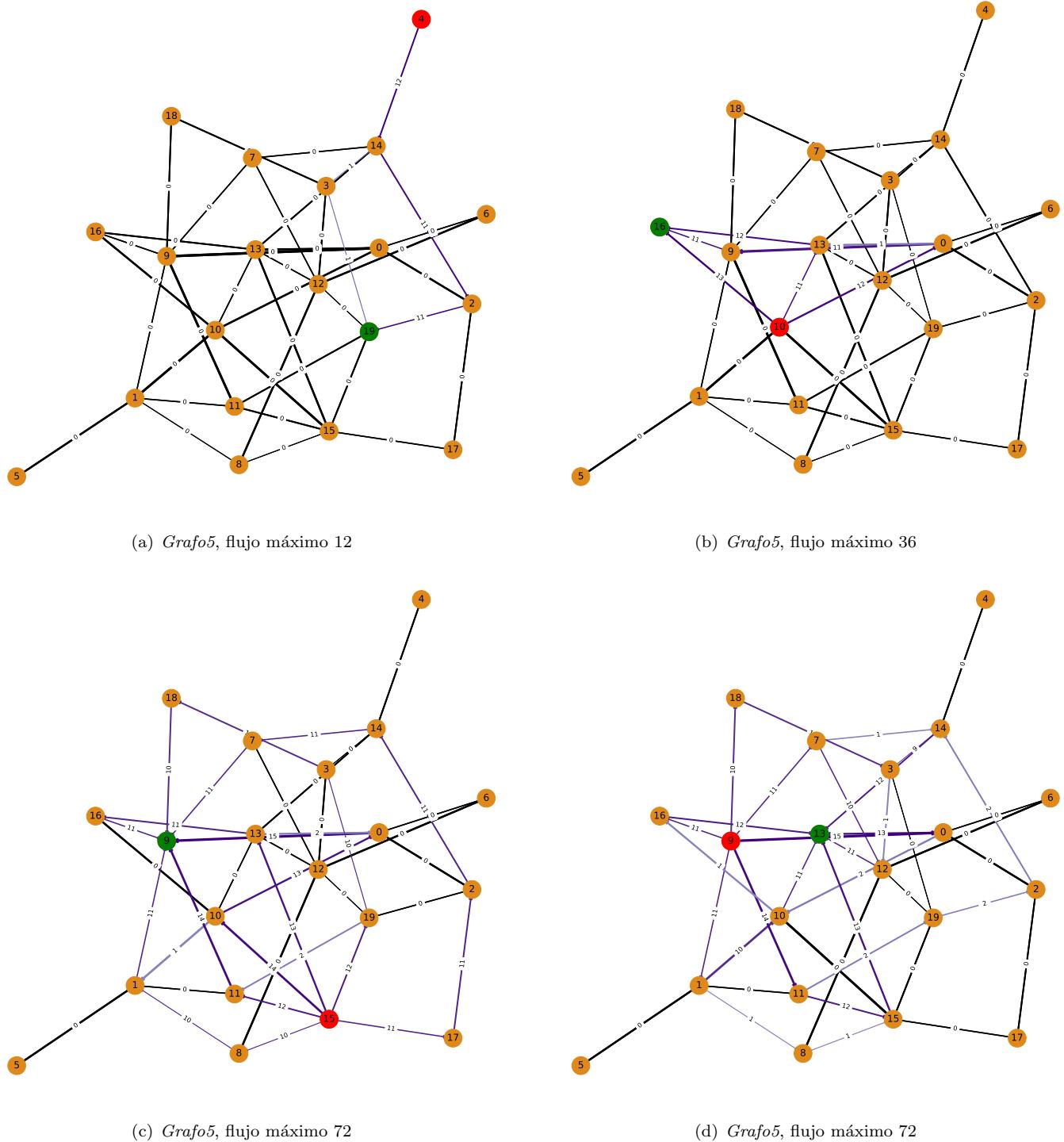


Figura 6: Grafos resultantes de la aplicación del algoritmo de flujo máximo, donde el grosor de las aristas representa la capacidad de las mismas, el color negro la ausencia de flujo, el color violeta la presencia de flujo y la intensidad del color violeta la cantidad de flujo que pasa por la arista. El color rojo del vértice representa la fuente y el vértice verde el sumidero.

5. Resultados del análisis de los datos

Con los datos recopilado de las ejecuciones del algoritmo de flujo máximo en los cinco grafos y todas las posibles combinaciones de fuentes y sumidero, se realizó histogramas para cada una de las seis características en los cinco grafos para determinar si los valores de la media del tiempo de ejecución tienen una distribución normal. Estos histogramas nos indican que la distribución no es normal como muestran la figuras 7, 8, 9, 10, 11, 12 de las páginas 14, 15, 16, 17, 18, 19 respectivamente.

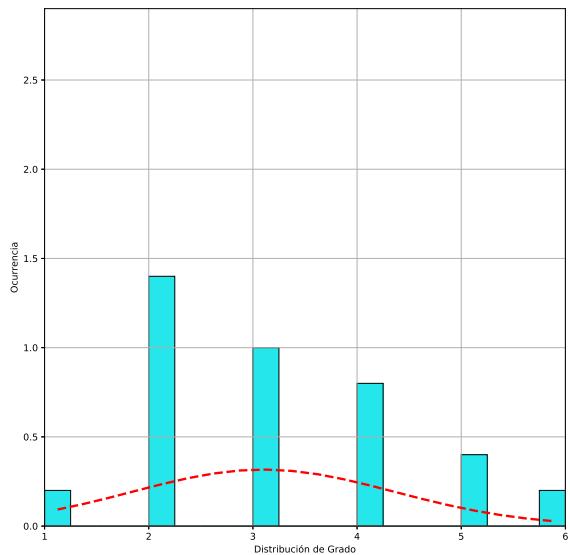
Dado que la distribución de los valores de las características no es normal, se hizo una categorización por rangos para hacer cómoda la visualización de la relación con las variables dependientes, estos rangos se obtuvieron a través de los contenedores que devolvió histogramas que se le aplicaron a cada propiedad.

Con esta categorización de las características se realizaron las pruebas estadísticas para analizar la influencia de las características estructurales de los grafos en el tiempo de ejecución del algoritmo de flujo máximo y el valor de flujo máximo.

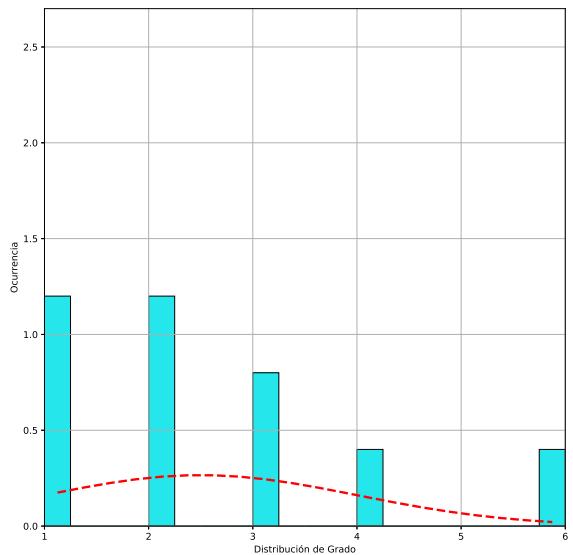
El siguiente fragmento de código nos muestra la realización de las pruebas antes mencionadas.

```
1 logX = np.log1p(df[ 'Mediana' ])
2 df = df.assign(mediana_log=logX.values)
3 df.drop(['Mediana'], axis= 1, inplace= True)
4
5 factores=[ "Grado" , "CoefAg" , "CentCer" , "CentCag" , "Excentricidad" , "PageRag" ]
6 plt.figure(figsize=(8, 6))
7 for i in factores:
8     print(rp.summary_cont(df[ 'FlujoMax' ].groupby(df[i])))
9
10 anova = pg.anova (dv='FlujoMax' , between=i, data=df, detailed=True , )
11 pg.export_table (anova ,("ANOVAAsFlujoMax"+i+".csv"))
12
13 ax=sns.boxplot(x=df[ "FlujoMax" ] , y=df[i] , data=df, palette="cubeHelix")
14
15 plt.savefig("boxplot_FlujoMax" + i + ".eps" , bbox_inches='tight')
16 tukey = pairwise_tukeyhsd(endog = df["FlujoMax"] , groups= df[i] , alpha=0.05)
17
18 tukey.plot_simultaneous(xlabel='Flujo Maximo' , ylabel=i)
19
20 plt.savefig("simultaneous_tukey" + i + ".eps" , bbox_inches='tight')
21
22 print(tukey.summary())
23 t_csv = open("TukeyFlujoMax"+i+".csv" , 'w')
24 with t_csv:
25     writer = csv.writer(t_csv)
26     writer.writerows(tukey.summary())
27 plt.show()
28
29 modelo = ols('FlujoMax ~ Grado + CoefAg + CentCer + CentCag + Excentricidad + PageRag + '
30                 'Grado*CoefAg + Grado*CentCer+ Grado*CentCag + Grado*Excentricidad + Grado*PageRag ,
31                 '+CoefAg*CentCer + CoefAg*CentCag + CoefAg*Excentricidad + CoefAg*PageRag +
32                 'CentCer*CentCag + CentCer*Excentricidad + CentCer*PageRag + CentCag*Excentricidad ,
33                 '+ CentCag*PageRag + Excentricidad*PageRag' , data=df).fit()
34 print(modelo.summary())
35 modelo.csv = open("Anova_MultFlujoMax.csv" , 'w')
36 aov_table = sm.stats.anova_lm(modelo , typ=2)
37 df1=pd.DataFrame(aov_table)
38 df1.to_csv("modeloFlujoMax.csv")
```

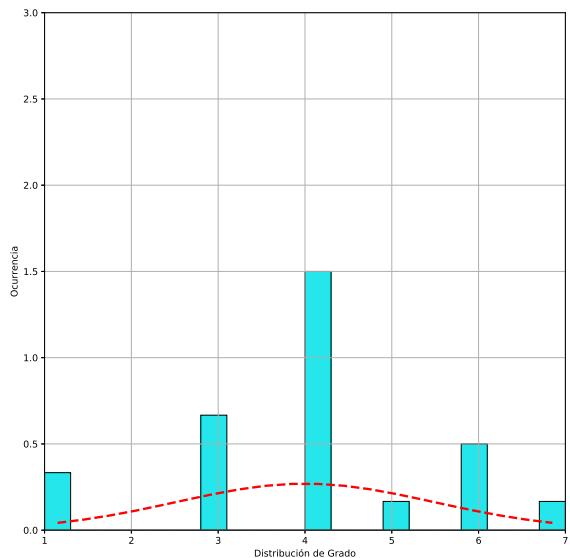
Analisis_datos1.py



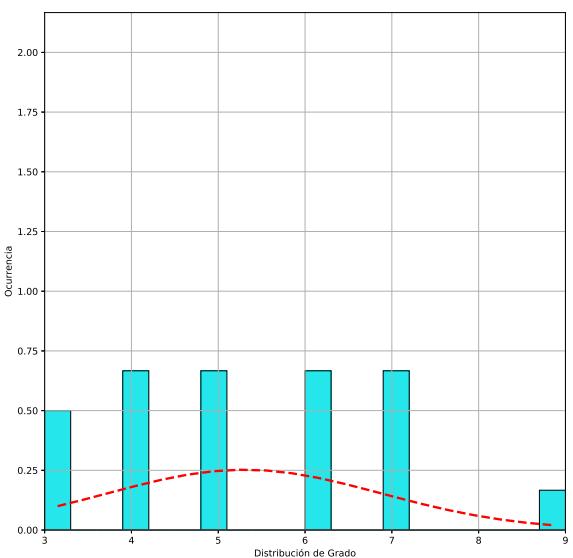
(a) *Grafo1*



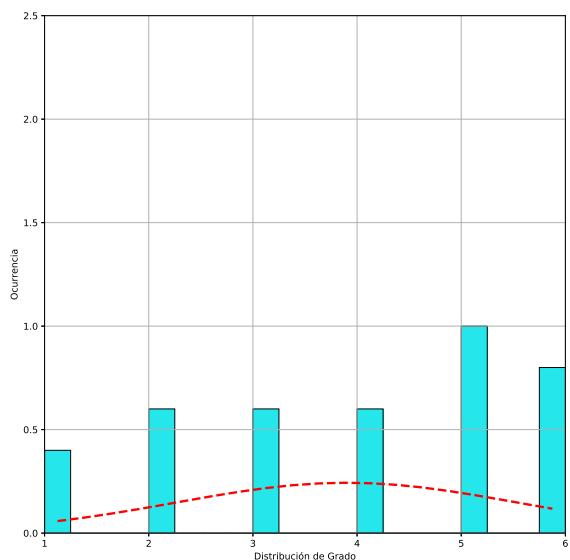
(b) *Grafo2*



(c) *Grafo3*

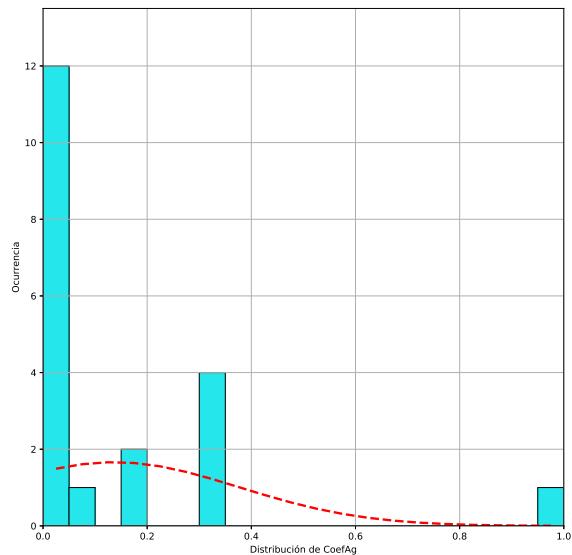


(d) *Grafo4*

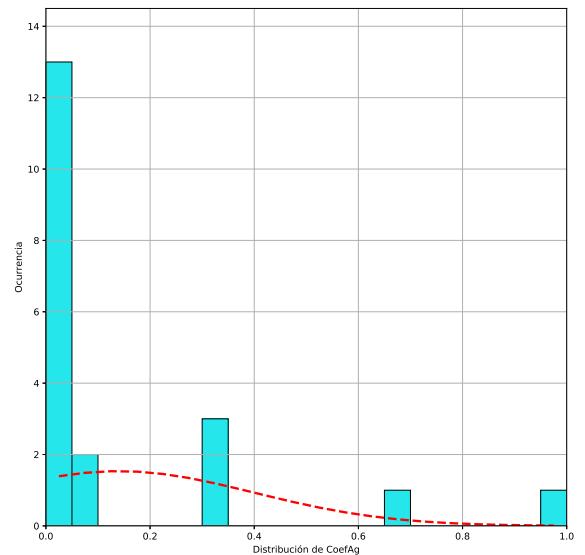


(e) *Grafo5*

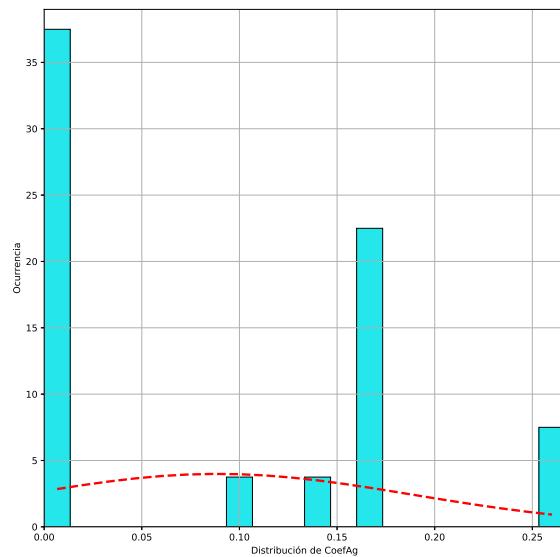
Figura 7: Histogramas que muestran la distribución de los valores de la característica distribución de grado.



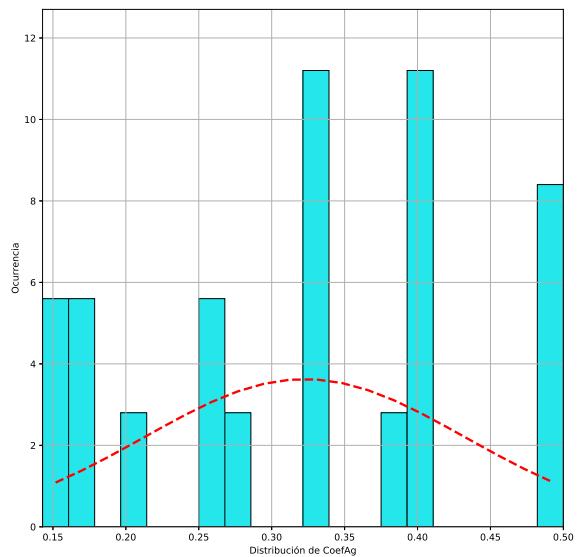
(a) *Grafo1*



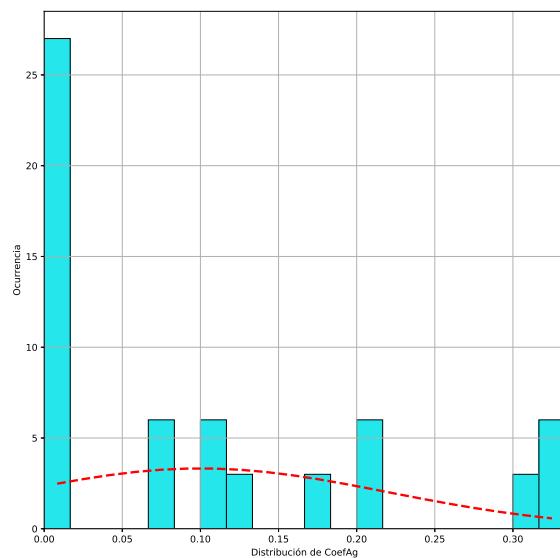
(b) *Grafo2*



(c) *Grafo3*

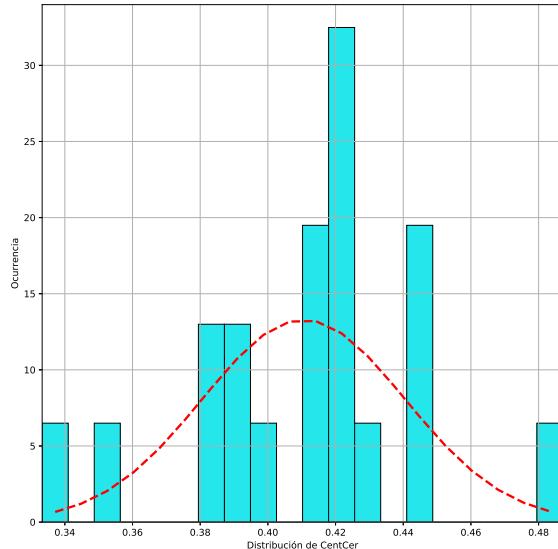


(d) *Grafo4*

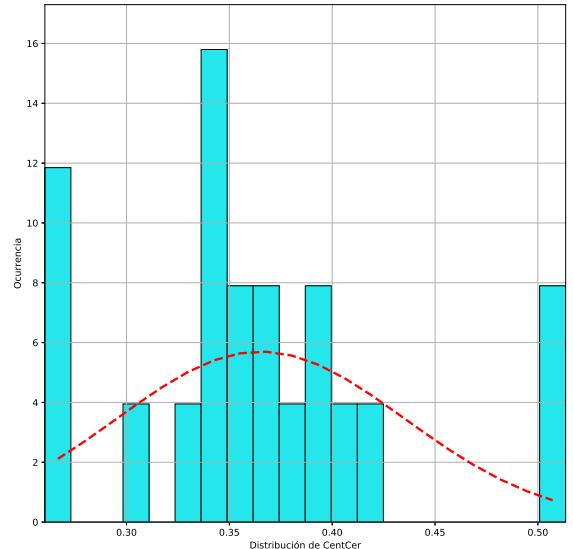


(e) *Grafo5*

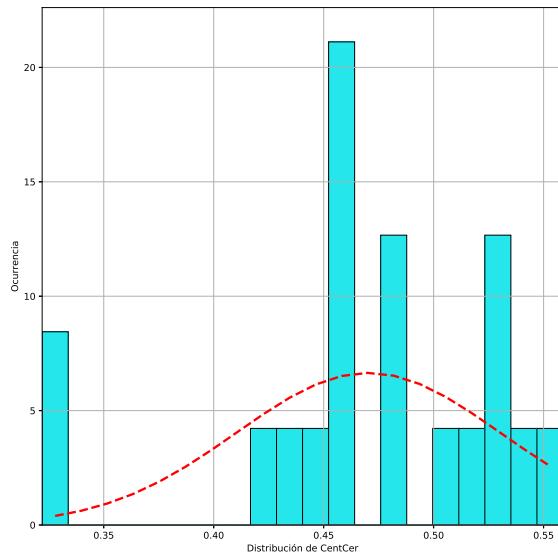
Figura 8: Histogramas que muestran la distribución de los valores de la característica coeficiente de agrupamiento.



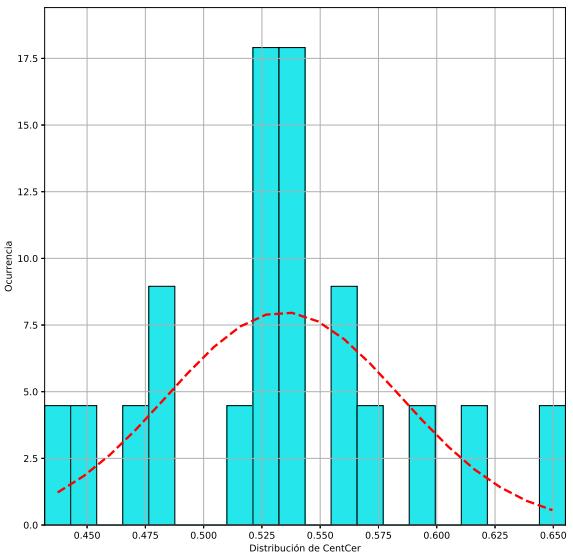
(a) *Grafo 1*



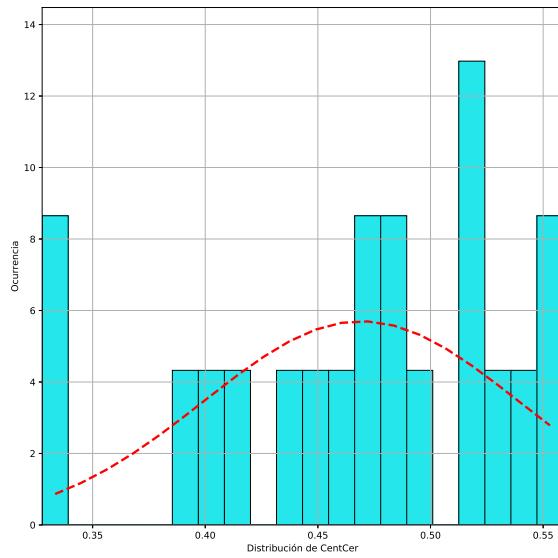
(b) *Grafo 2*



(c) *Grafo 3*

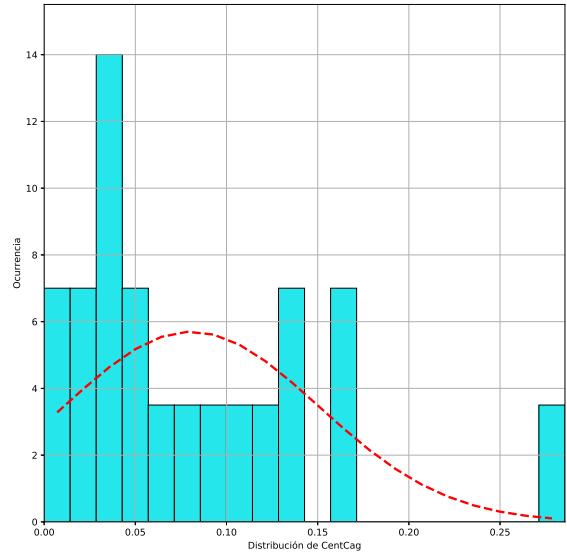


(d) *Grafo 4*

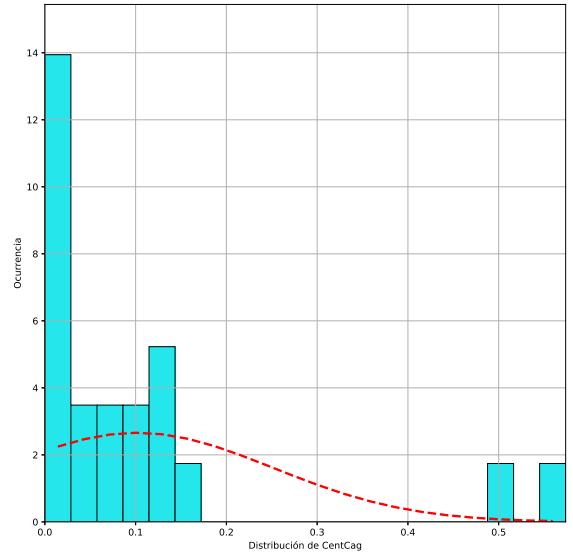


(e) *Grafo 5*

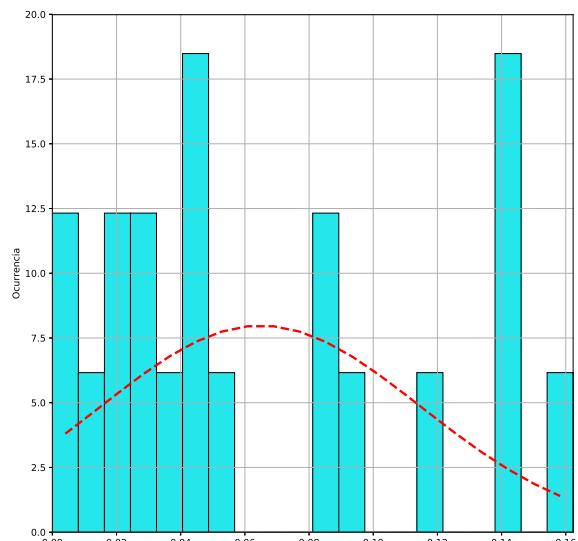
Figura 9: Histogramas que muestran la distribución de los valores de la característica centralidad de cercanía.



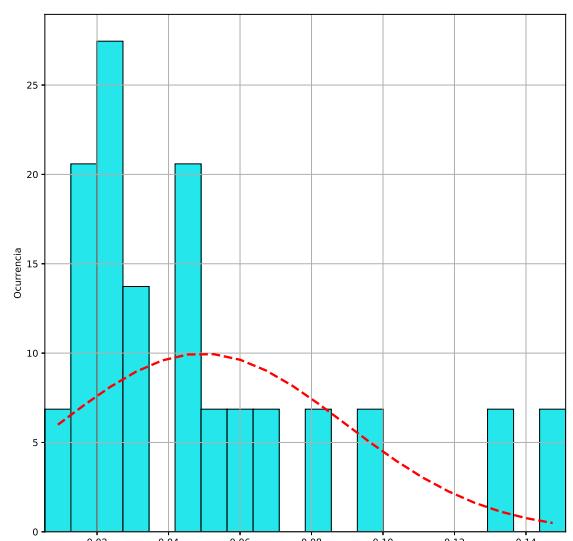
(a) *Grafo1*



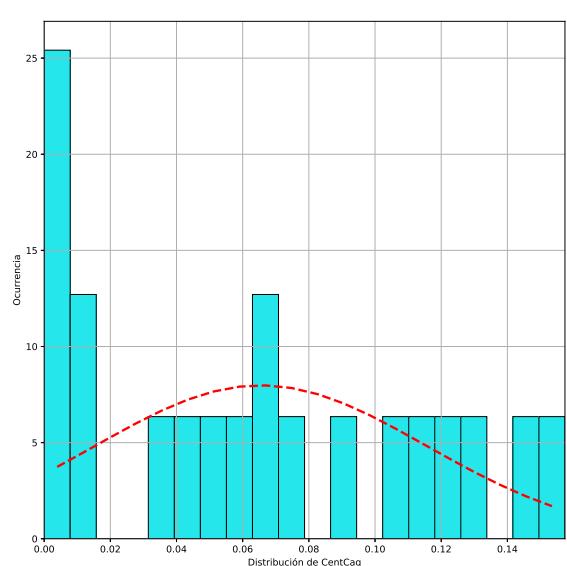
(b) *Grafo2*



(c) *Grafo3*

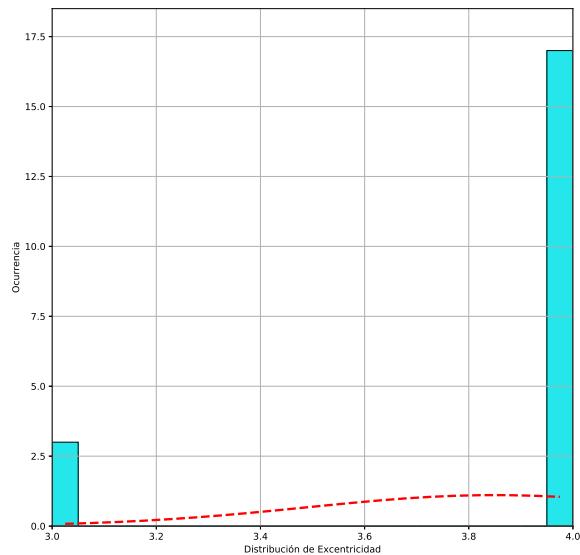


(d) *Grafo4*

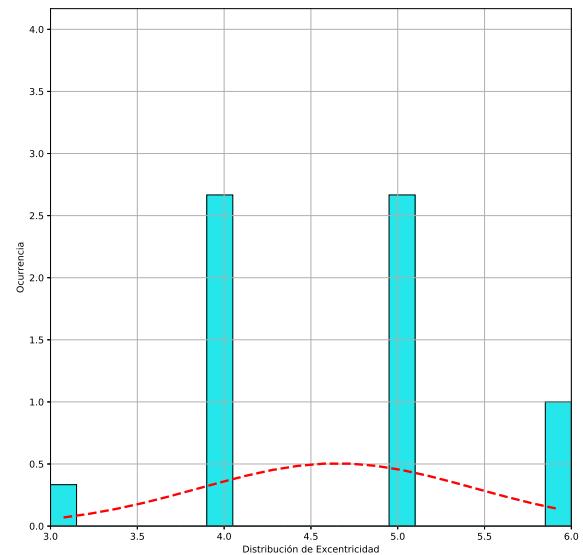


(e) *Grafo5*

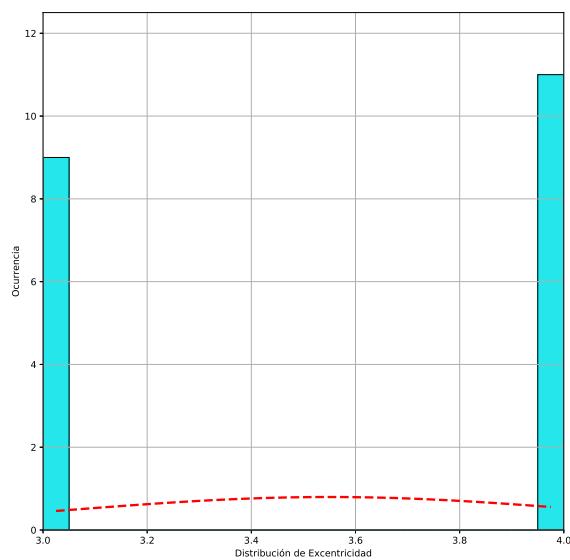
Figura 10: Histogramas que muestran la distribución de los valores de la característica centralidad de carga.



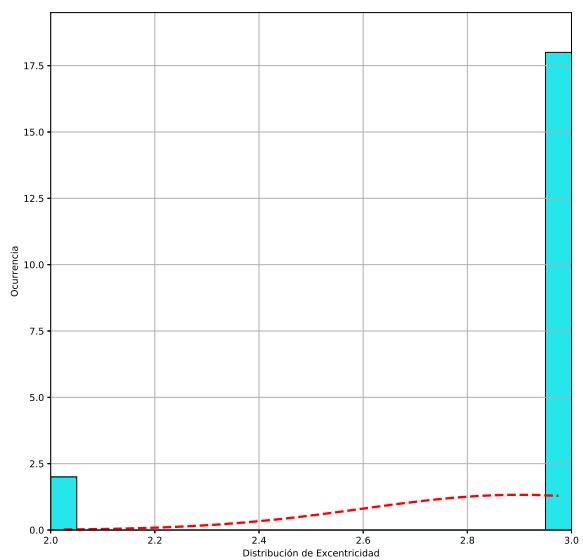
(a) *Grafo1*



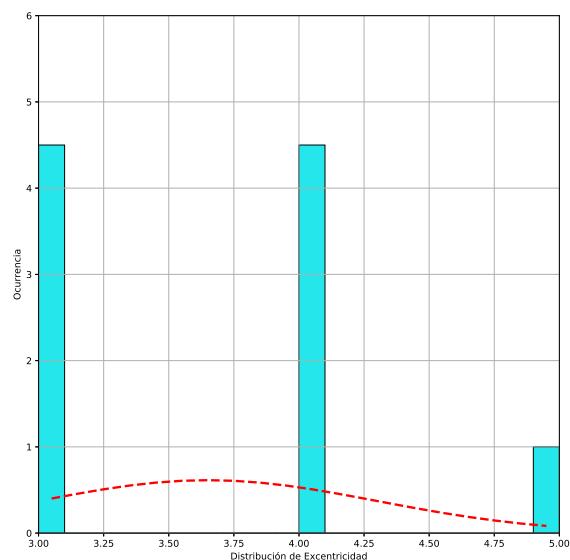
(b) *Grafo2*



(c) *Grafo3*

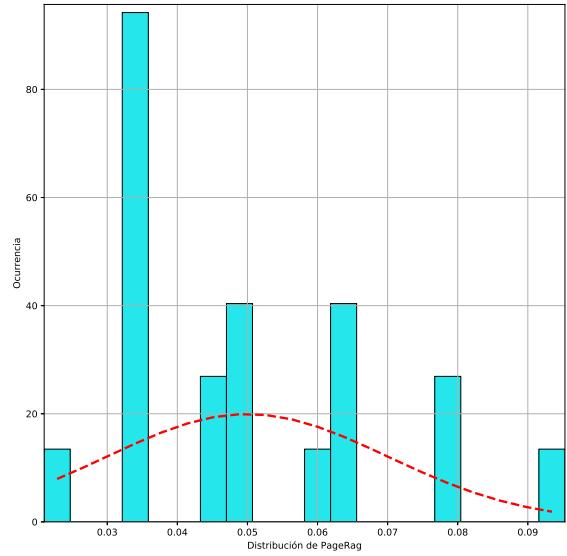


(d) *Grafo4*

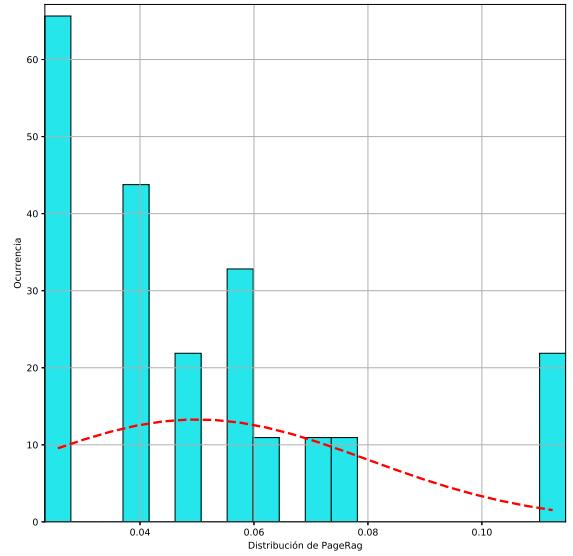


(e) *Grafo5*

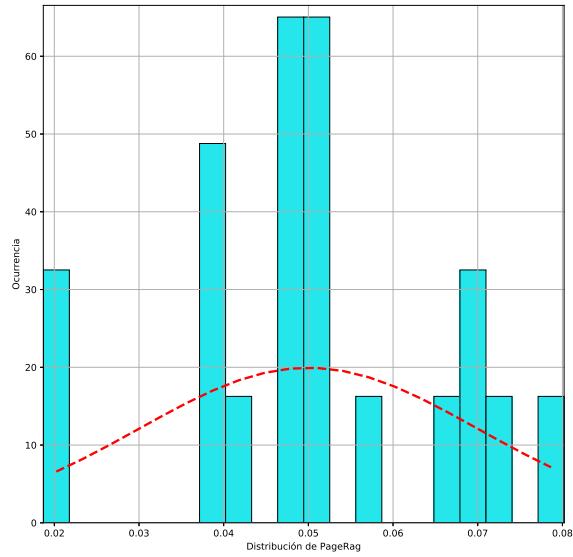
Figura 11: Histogramas que muestran la distribución de los valores de la característica excentricidad.



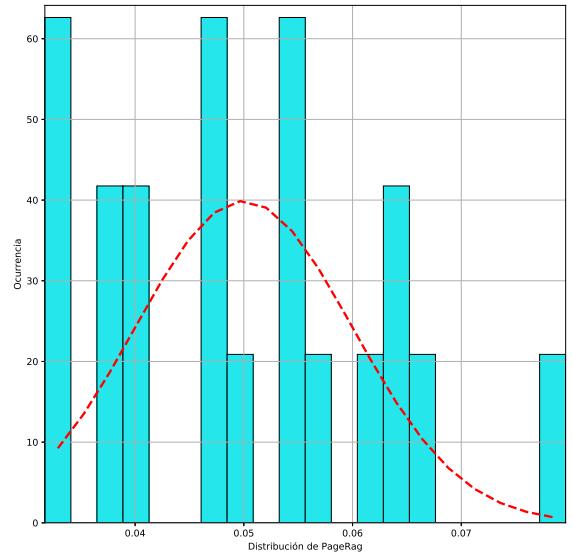
(a) *Grafo1*



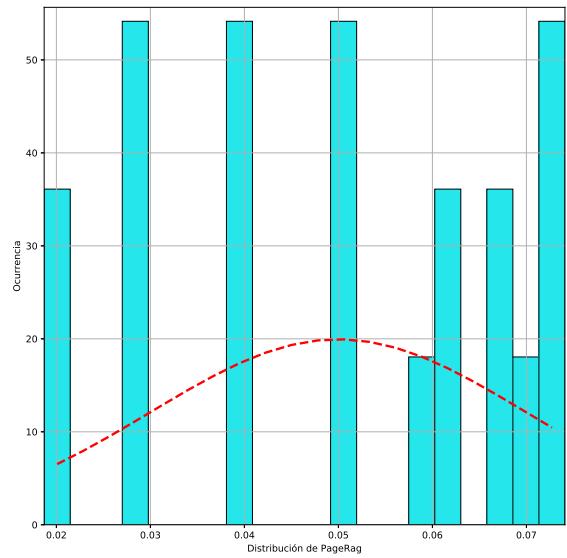
(b) *Grafo2*



(c) *Grafo3*



(d) *Grafo4*



(e) *Grafo5*

Figura 12: Histogramas que muestran la distribución de los valores de la característica *PageRank*.

5.1. Análisis de varianza (ANOVA)

El análisis de varianza (ANOVA) es la técnica central en el análisis de datos experimentales. La idea general de esta técnica es separar la variación total en las partes con las que contribuye cada fuente de variación en el experimento. En el caso de los diseños completamente al azar se separan la variabilidad debida a los tratamientos y la debida al error. Cuando la primera predomina sobre la segunda, es cuando se concluye que las medias son diferentes. Cuando los tratamientos no dominan contribuyen igual o menos que el error, por lo que se concluye que las medias son iguales [2]. Para analizar si los diferentes factores (distribución de grado, coeficiente de agrupamiento, centralidad de cercanía, centralidad de carga, excentricidad, *pagerank*) influyen en la variable dependiente *tiempo de ejecución* y valores de flujo máximo se realizó un ANOVA de un factor para cada caso.

5.1.1. Influencia de la distribución de grado en el tiempo de ejecución

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 3: Influencia de la distribución de grado en el tiempo de ejecución (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>Grado</i>	209597.452	2.000	104798.726	662.186	0.000	0.411
<i>Within</i>	300222.666	1897.000	158.262	-	-	-

En el cuadro 3 se muestra que existen diferencia entre las medianas de los grupos de factores ya que el **p-unc** es menor que 0,05 por lo que se rechaza la hipótesis de que la distribución de grado no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 13 de la página 20.

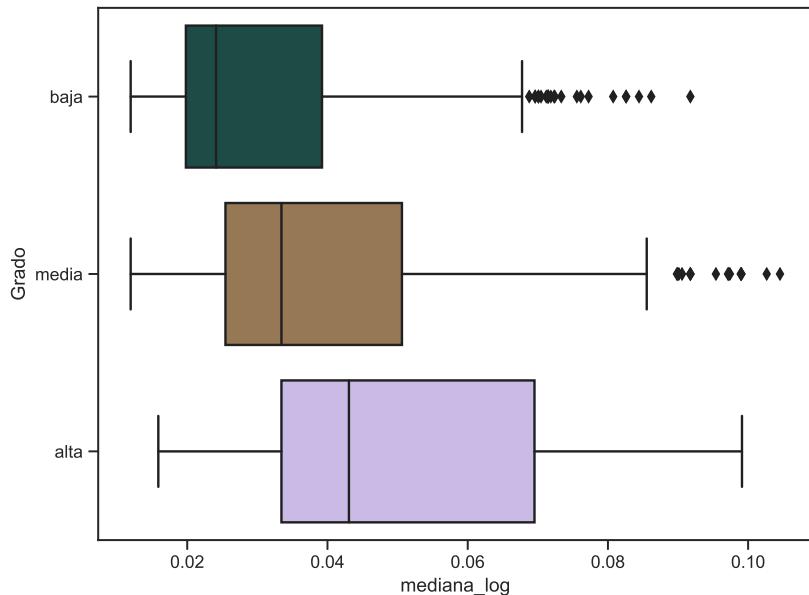


Figura 13: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con la distribución de grado.

En el cuadro 3 se muestra que existen grandes diferencia entre las medianas de los grupos de factores ya que el **p-unc** es menor que 0,05 por lo que se rechaza la hipótesis de que la distribución de grado no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 4 de la página 10. Por tal motivo se realiza la prueba de Tukey mostrara las diferencias entre las medianas de los factores, lo que se evidencia en el cuadro 4 de la página 21.

Cuadro 4: Influencia de la distribución de grado en el tiempo de ejecución (*Tukey*)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	-0.020	-0.024	-0.016	True
alta	media	-0.011	-0.015	-0.007	True
baja	media	0.009	0.007	0.011	True

5.1.2. Influencia del coeficiente de agrupamiento en el tiempo de ejecución

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 5: Influencia del coeficiente de agrupamiento en el tiempo de ejecución (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>CoefAg</i>	0.013	2.000	0.006	19.962	0.000	0.021
<i>Within</i>	0.610	1897.000	0.000			

En el cuadro 5 se muestra que existen diferencias entre las medianas de los grupos de factores ya que el **p-unc** es menor que 0,05 por lo que se rechaza la hipótesis de que el coeficiente de agrupamiento no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 14 de la página 21.

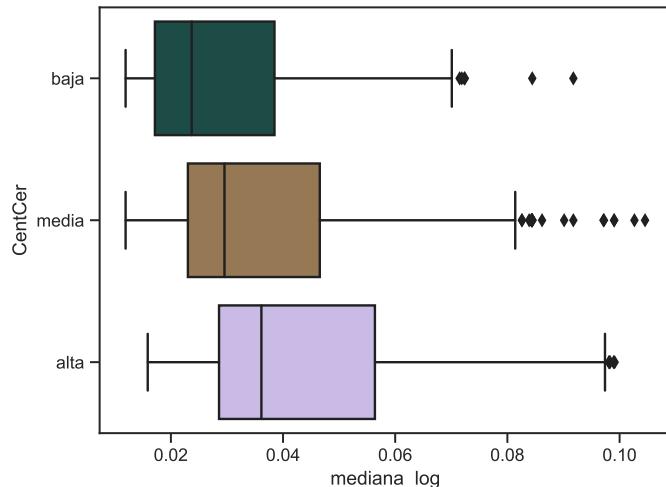


Figura 14: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con el coeficiente de agrupamiento.

En el cuadro 5 se muestra que existen diferencias entre las medianas de los grupos de factores **p-unc** es menor que 0,05 por lo que se rechaza la hipótesis de que el coeficiente de agrupamiento no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 14 de la página 21. Por tal motivo se realiza la prueba de Tukey mostrando las diferencias entre las medianas de los factores, lo que se evidencia en el cuadro 6 de la página 22.

Cuadro 6: Influencia del coeficiente de agrupamiento en el tiempo de ejecución (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	0.005	-0.001	0.010	False
alta	media	0.014	0.007	0.020	True
baja	media	0.009	0.005	0.013	True

5.1.3. Influencia de la centralidad de cercanía en el tiempo de ejecución

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 7: Influencia de la centralidad de cercanía en el tiempo de ejecución

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>CentCer</i>	0.054	2.000	0.027	89.249	0.000	0.086
<i>Within</i>	0.570	1897.000	0.000			

En el cuadro 7 se muestra que existen diferencias entre las medianas de los grupos de factores ya que el **p-unc** es menor que 0,05 por lo que se rechaza la hipótesis de que la centralidad de cercanía no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 15 de la página 22.

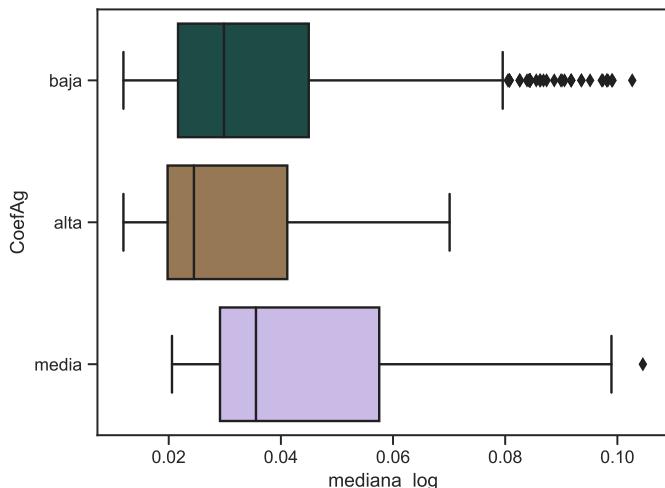


Figura 15: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con la centralidad de cercanía.

En el cuadro 7 se muestra que existen diferencias entre las medianas de los grupos de factores **p-unc** es menor que 0,05 por lo que se rechaza la hipótesis de que la centralidad de cercanía no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 15 de la página 22. Por tal motivo se realiza la prueba de Tukey mostrando las diferencias entre las medianas de los factores, lo que se evidencia en el cuadro 8 de la página 23.

Cuadro 8: Influencia de la centralidad de cercanía en el tiempo de ejecución (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	-0.016	-0.018	-0.013	<i>True</i>
alta	media	-0.008	-0.011	-0.006	<i>True</i>
baja	media	0.007	0.005	0.010	<i>True</i>

5.1.4. Influencia de la centralidad de carga en el tiempo de ejecución

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 9: Influencia de la centralidad de carga en el tiempo de ejecución (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>CentCag</i>	0.000	2.000	0.000	0.476	0.621	0.001
<i>Within</i>	0.623	1897.000	0.000			

En el cuadro 9 se muestra que no existen diferencias entre las medianas de los grupos de factores ya que el **p-unc** es menor que 0,05 por lo que se acepta la hipótesis de que la centralidad de carga no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 16 de la página 23.

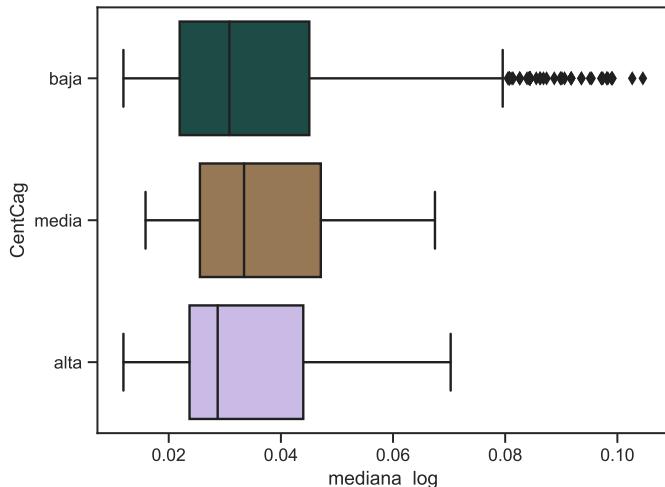


Figura 16: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con la centralidad de carga.

5.1.5. Influencia de la excentricidad en el tiempo de ejecución

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

Cuadro 10: Influencia de la excentricidad en el tiempo de ejecución (ANOVA)

Source	SS	DF	MS	F	p-unc	np2
Excentricidad	0.043	2.000	0.021	70.117	0.000	0.069
Within	0.580	1897.000	0.000			

En el cuadro 10 se muestra que existen diferencia entre las medianas de los grupos de factores ya que el **p-unc** es menor que 0,05 por lo que se rechaza la hipótesis de que la excentricidad no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 17 de la página 24.

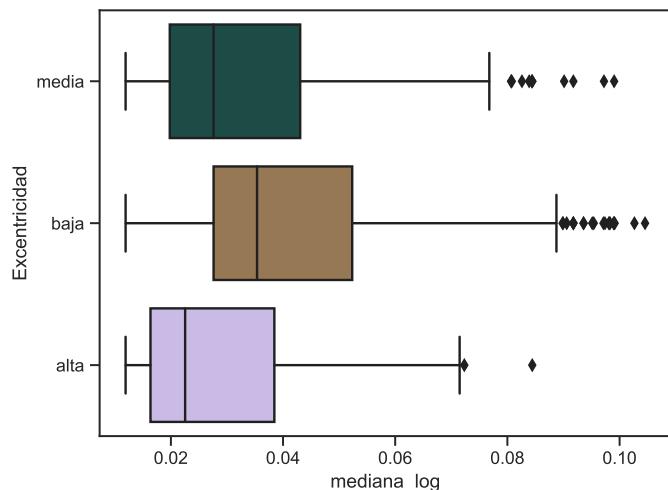


Figura 17: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con la excentricidad.

En el cuadro 10 se muestra que existen diferencia entre las medianas de dos de los grupos de factores **p-unc** es menor que 0,05 por lo que se rechaza la hipótesis de que la excentricidad no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 17 de la página 24. Por tal motivo se realiza la prueba de Tukey mostrara las diferencias entre las medianas de los factores, lo que se evidencia en el cuadro 11 de la página 24.

Cuadro 11: Influencia de la excentricidad en el tiempo de ejecución (Tukey)

group1	group2	meandiff	lower	upper	reject
alta	baja	0.013	0.010	0.016	True
alta	media	0.004	0.001	0.007	True
baja	media	-0.008	-0.010	-0.006	True

5.1.6. Influencia del *PageRank* en el tiempo de ejecución

El siguiente cuadro muestra el resultado de la aplicación del ANOVA.

En el cuadro 12 se muestra que existen diferencia entre las medianas de los grupos de factores ya que el *p - unc* es menor que 0,05 por lo que se rechaza la hipótesis de que el *PageRank* no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 18 de la página 25

Cuadro 12: Influencia del *PageRank* en el tiempo de ejecución (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>PageRag</i>	0.016	2.000	0.008	25.358	0.000	0.026
<i>Within</i>	0.607	1897.000	0.000			

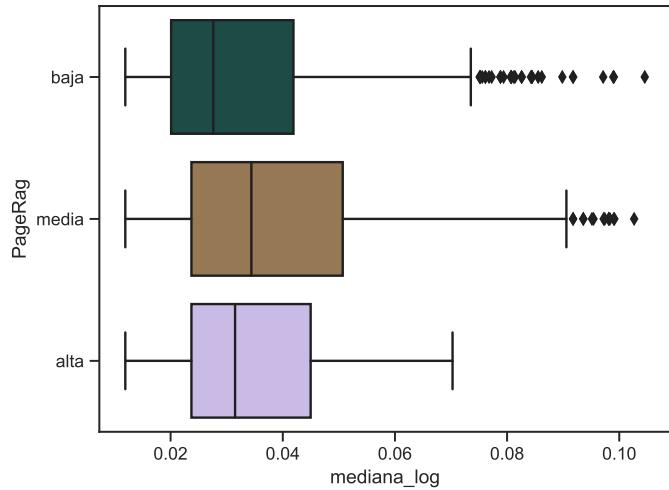


Figura 18: Diagrama de caja y bigotes que relaciona los tiempos de ejecución con el *PageRank*.

En el cuadro 12 se muestra que existen diferencia entre las medianas de dos de los grupos de factores ***p-unc*** es menor que 0,05 por lo que se rechaza la hipótesis de que el *PageRank* no influye en el *tiempo de ejecución*. Esto se puede observar en la figura 17 de la página 24. Por tal motivo se realiza la prueba de Tukey mostrara las diferencias entre las medianas de los factores, lo que se evidencia en el cuadro 13 de la página 25.

Cuadro 13: Influencia del *PageRank* en el tiempo de ejecuciónAdd (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	-0.001	-0.006	0.005	<i>False</i>
alta	media	0.005	-0.001	0.011	<i>False</i>
baja	media	0.006	0.004	0.008	<i>True</i>

5.1.7. Influencia de los seis factores en el tiempo de ejecución

Para analizar este caso se realizó ANOVA multifactorial teniendo en cuenta las seis características (distribución de grado, coeficiente de agrupamiento, centralidad de cercanía, centralidad de carga, excentricidad, *pagerank*) dando como resultado el cuadro 14 de la página 26.

Cuadro 14: Influencia de los seis factores (distribución de grado, coeficiente de agrupamiento, centralidad de cercanía, centralidad de carga, excentricidad, *pagerank*) en el tiempo de ejecución (ANOVA)

	<i>sum_sq</i>	<i>df</i>	<i>F</i>	<i>PR(>F)</i>
Grado	0.000	2.000	0.000	1.000
CoefAg	0.000	2.000	0.000	1.000
CentCer	0.000	2.000	-0.409	1.000
CentCag	0.000	2.000	0.000	1.000
Excentricidad	0.000	2.000	0.000	1.000
PageRag	0.000	2.000	0.000	1.000
Grado:CoefAg	0.001	4.000	0.594	0.619
Grado:CentCer	0.001	4.000	1.048	0.381
Grado:CentCag	0.001	4.000	0.854	0.491
Grado:Excentricidad	0.001	4.000	1.064	0.373
Grado:PageRag	0.001	4.000	0.858	0.489
CoefAg:CentCer	0.001	4.000	0.560	0.571
CoefAg:CentCag	0.001	4.000	0.525	0.665
CoefAg:Excentricidad	0.000	4.000	0.203	0.816
CoefAg:PageRag	0.001	4.000	0.430	0.731
CentCer:CentCag	0.001	4.000	0.758	0.517
CentCer:Excentricidad	0.001	4.000	0.543	0.653
CentCer:PageRag	0.001	4.000	0.755	0.519
CentCag:Excentricidad	0.001	4.000	0.584	0.626
CentCag:PageRag	0.001	4.000	0.640	0.527
Excentricidad:PageRag	0.001	4.000	0.677	0.608
Residual	0.549	1876.000		

En el cuadro 7 se puede observar que los factores que más influyen en el tiempo de ejecución son el número de vértices y la densidad de los grafos, que además existe una relación entre el número de nodos y la densidad de los grafos.

5.1.8. Influencia de las seis propiedades en el flujo máximo

Los cuadro 15, 16, 17, 18, 19, 19 de las páginas 26, 26, 27, 27, 27 respectivamente, muestran los resultados de la aplicación del ANOVA en las seis propiedades con respecto los valores de flujo máximo.

Cuadro 15: Influencia de la distribución de grado en el flujo máximo (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
Grado	209597.452	2.000	104798.726	662.186	0.000	0.411
Within	300222.666	1897.000	158.262			

Cuadro 16: Influencia de la distribución de coeficiente de agrupamiento en el flujo máximo (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
CoefAg	59963.814	2.000	29981.907	126.431	0.000	0.118
Within	449856.304	1897.000	237.141			

En los cuadro anteriores se muestra que existen diferencia entre las medianas de los grupos de factores ya que el *p-unc* es menor que 0,05 por lo que se rechaza la hipótesis de que las propiedades no influye en el *flujo máximo*. Esto se puede observar claramente en las figuras 19, 20, 21, 22, 23, 23 de las páginas 27, 28, 28, 29, 29 respectivamente.

Cuadro 17: Influencia de la distribución de centralidad de cercanía en el flujo máximo (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>CentCer</i>	256496.045	2.000	128248.023	960.377	0.000	0.503
<i>Within</i>	253324.073	1897.000	133.539			

Cuadro 18: Influencia de la centralidad de carga en el flujo máximo (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>CentCag</i>	6469.019	2.000	3234.510	12.190	0.000	0.013
<i>Within</i>	503351.098	1897.000	265.341			

Cuadro 19: Influencia de la excentricidad en el flujo máximo (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
Excentricidad	200853.698	2.000	100426.849	616.603	0.000	0.394
Within	308966.420	1897.000	162.871	-	-	-

Cuadro 20: Influencia de el *pagerank* en el flujo máximo (ANOVA)

<i>Source</i>	<i>SS</i>	<i>DF</i>	<i>MS</i>	<i>F</i>	<i>p-unc</i>	<i>np2</i>
<i>PageRag</i>	67070.251	2.000	33535.125	143.684	0.000	0.132
<i>Within</i>	442749.867	1897.000	233.395			

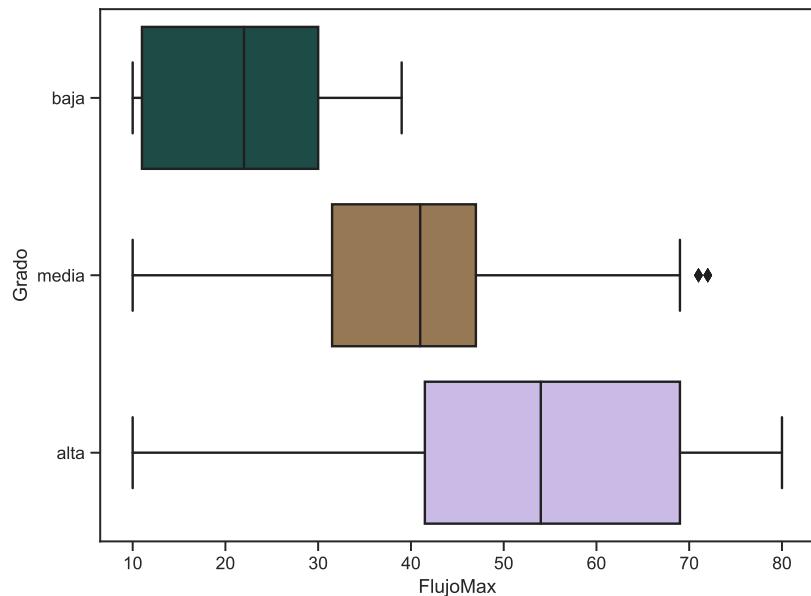


Figura 19: Diagrama de caja y bigotes que relaciona los flujos máximos con la distribución de grado.

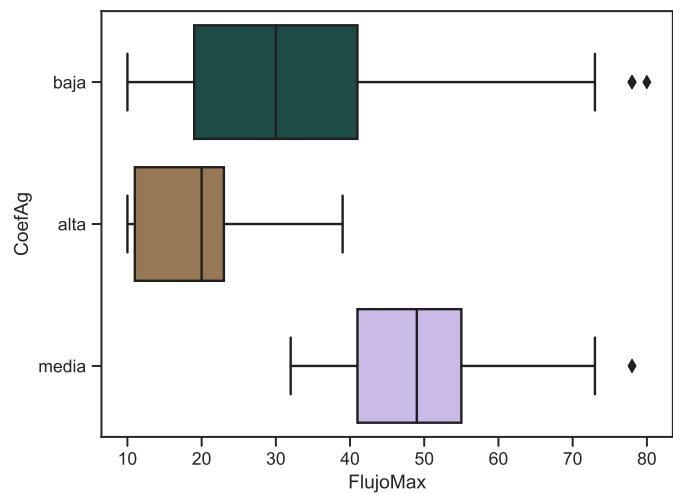


Figura 20: Diagrama de caja y bigotes que relaciona los flujos máximos con el coeficiente de agrupamiento.

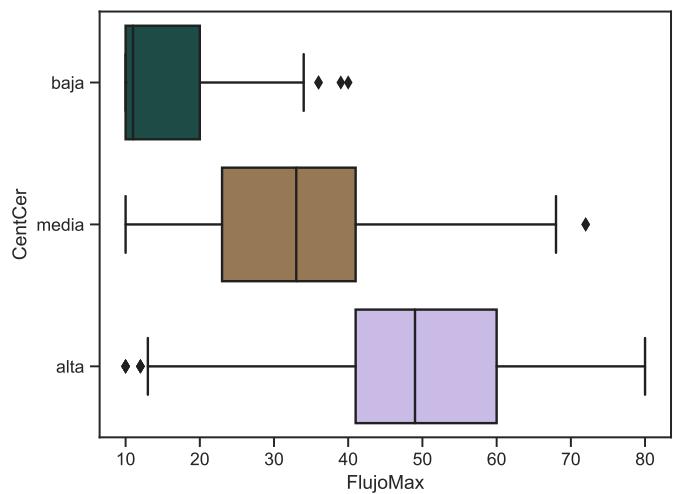


Figura 21: Diagrama de caja y bigotes que relaciona los flujos máximos con la centralidad de cercanía.

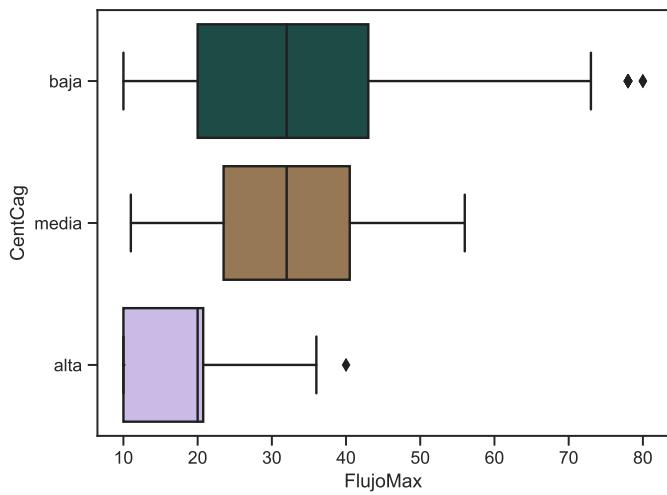


Figura 22: Diagrama de caja y bigotes que relaciona los flujos máximos con la centralidad de carga.

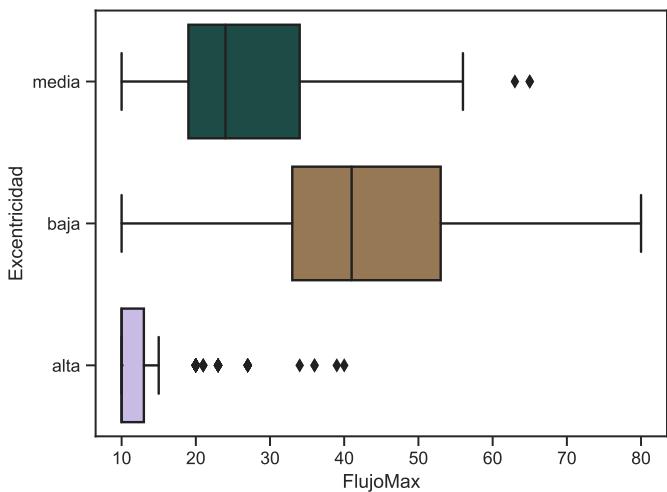


Figura 23: Diagrama de caja y bigotes que relaciona los flujos máximos con la excentricidad.

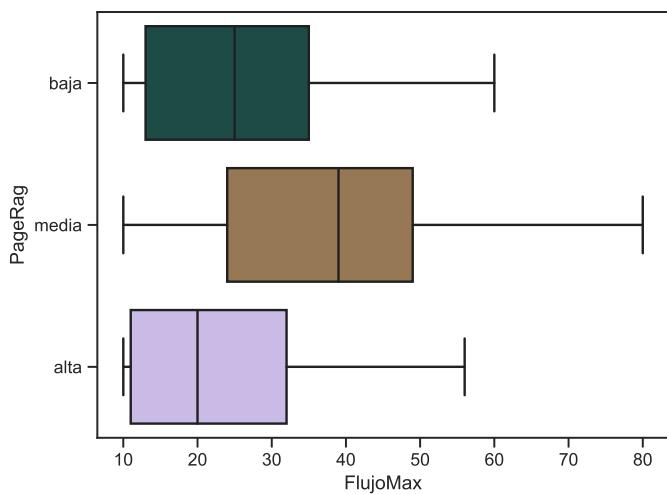


Figura 24: Diagrama de caja y bigotes que relaciona los flujos máximos con el *pagerank*.

Debido a las diferencias obtenidas en todas las ANOVAs realizadas, procedemos a relizar la prueba de Tukey cuyos resultados se muestran en los cuadros

Cuadro 21: Influencia de la distribución de grado en el flujo máximo (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	-33.721	-36.659	-30.782	<i>True</i>
alta	media	-15.682	-18.613	-12.751	<i>True</i>
baja	media	18.039	16.642	19.436	<i>True</i>

Cuadro 22: Influencia de el coeficiente de agrupamiento en el flujo máximo (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	11.253	6.389	16.118	<i>True</i>
alta	media	30.351	24.741	35.961	<i>True</i>
baja	media	19.098	16.039	22.156	<i>True</i>

Cuadro 23: Influencia de la centralidad de cercanía en el flujo máximo (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	-33.901	-35.715	-32.086	<i>True</i>
alta	media	-17.016	-18.568	-15.463	<i>True</i>
baja	media	16.885	15.355	18.415	<i>True</i>

Cuadro 24: Influencia de la centralidad de carga en el flujo máximo (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	13.165	6.904	19.427	<i>True</i>
alta	media	13.816	3.080	24.551	<i>True</i>
baja	media	0.651	-8.160	9.461	<i>False</i>

Cuadro 25: Influencia de la excentricidad en el flujo máximo (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	29.093	26.913	31.272	<i>True</i>
alta	media	12.855	10.692	15.017	<i>True</i>
baja	media	-16.238	-17.711	-14.764	<i>True</i>

Cuadro 26: Influencia del *pagerank* en el flujo máximo (Tukey)

<i>group1</i>	<i>group2</i>	<i>meandiff</i>	<i>lower</i>	<i>upper</i>	<i>reject</i>
alta	baja	3.867	-0.998	8.733	<i>False</i>
alta	media	15.846	10.916	20.776	<i>True</i>
baja	media	11.979	10.269	13.689	<i>True</i>

En los cuadros 21, 22, 23, 24, 25, 26 de las páginas 30, 30, 30, 31, 31, 31 se pueden observar claramente que las seis características influye en el valor máximo de flujo.

5.1.9. Influencia de los seis factores en el flujo máximo

Para analizar este caso se realizó ANOVA multifactorial teniendo en cuenta las seis características (distribución de grado, coeficiente de agrupamiento, centralidad de cercanía, centralidad de carga, excentricidad, *pagerank*) dando como resultado el cuadro 27 de la página 32.

En el cuadro 27 de la página 32 se observa claramente que todos las características influye en los valores del flujo máximo corroborando lo arrojado en las pruebas anteriores.

Cuadro 27: Influencia de los seis factores (distribución de grado, coeficiente de agrupamiento, centralidad de cercanía, centralidad de carga, excentricidad, *pagerank*) en el flujo máximo (ANOVA)

	<i>sum_sq</i>	<i>df</i>	<i>F</i>	<i>PR(>F)</i>
Grado	0.001	2.000	0.000	1.000
CoefAg	-45.503	2.000	-0.210	1.000
CentCer	2523.680	2.000	11.667	0.000
CentCag	0.000	2.000	0.000	1.000
Excentricidad	0.000	2.000	0.000	1.000
PageRag	0.000	2.000	0.000	1.000
Grado:CoefAg	34.118	4.000	0.079	0.971
Grado:CentCer	37.572	4.000	0.087	0.987
Grado:CentCag	54.632	4.000	0.126	0.973
Grado:Excentricidad	35.792	4.000	0.083	0.988
Grado:PageRag	59.651	4.000	0.138	0.968
CoefAg:CentCer	55.193	4.000	0.128	0.880
CoefAg:CentCag	38.260	4.000	0.088	0.966
CoefAg:Excentricidad	34.182	4.000	0.079	0.924
CoefAg:PageRag	45.273	4.000	0.105	0.957
CentCer:CentCag	46.187	4.000	0.107	0.956
CentCer:Excentricidad	41.053	4.000	0.095	0.963
CentCer:PageRag	74.952	4.000	0.173	0.915
CentCag:Excentricidad	90.135	4.000	0.208	0.891
CentCag:PageRag	79.931	4.000	0.185	0.831
Excentricidad:PageRag	51.771	4.000	0.120	0.976
Residual	202895.688	1876.000		

6. Conclusiones

Después del análisis realizado se concluyó que las características estructurales sí influye tanto en el tiempo de ejecución como en el valor máximo de flujo, en los grafos generados para esta tarea la características que influyen en mayor medida es la distribución de grado y *pagerank*.

Referencias

- [1] Bartłomiej Bosek, Dariusz Leniowski, Piotr Sankowski, and Anna Zych-Pawlewicz. Shortest augmenting paths for online matchings on trees. *Theory of Computing Systems*, 62(2):337–348, Feb 2018.
- [2] Humberto Gutiérrez and Román de la Vara. *Análisis y diseño de experimentos*. The McGraw-Hill Companies, Inc., segunda edición edition, 2008. 60–74.
- [3] Desarrolladores NetworkX. https://networkx.github.io/documentation/stable/reference/generated/networkx.generators.random_graphs.erdos_renyi_graph.html#networkx.generators.random_graphs.erdos_renyi_graph. Accessed: 01-04-2019.
- [4] Desarrolladores NetworkX. https://networkx.github.io/documentation/stable/reference/generated/networkx.generators.random_graphs.fast_gnp_random_graph.html#networkx.generators.random_graphs.fast_gnp_random_graph. Accessed: 01-04-2019.
- [5] Desarrolladores NetworkX. https://networkx.github.io/documentation/stable/reference/generated/networkx.generators.random_graphs.binomial_graph.html#networkx.generators.random_graphs.binomial_graph. Accessed: 01-04-2019.
- [6] Desarrolladores NetworkX. https://networkx.github.io/documentation/stable/reference/algorithms/generated/networkx.algorithms.flow.maximum_flow.html#networkx.algorithms.flow.maximum_flow. Accessed: 01-04-2019.
- [7] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.coloring.greedy_color.html. Accessed: 18-03-2019.
- [8] Sadegh Nobari, Xuesong Lu, Panagiotis Karras, and Stéphane Bressan. Fast random graph generation. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT ’11, pages 331–342, New York, NY, USA, 2011. ACM.