Universidad Autónoma de Nuevo León

Facultad de Ingeniería Mecánica y Eléctrica

Flujo en redes

PORTAFOLIO

Héctor Hugo García López
1484166

Enero-Junio 2019

Dra. Elisa Schaeffer

1. Tarea uno original

En esta tarea fuí corregido por el tiempo en el que se escribió el documento, por acentos, por referencias y por no usar el comando `newpage` para saltar de pagina en los códigos.

Introducción

Un grafo esta formado por un conjunto de nodos que llamaremos N, y un conjunto de la forma (u, v) llamado aristas al que nos referiremos por A. Dada su composición es conveniente ver un grafo como un dibujo. Los grafos se pueden usar para ilustrar procesos, rutas, cambios de estados en ciertos procesos y hasta un árbol genealógico. En este documento se mostrará los tipos grafos, cuales son las condiciones que tienen que cumplir y un ejemplo de su aplicación.

Para hacer una mejor referencia a los grafos, los vamos a dibujar en python y para eso necesitamos hacer uso de dos librerias, como se presenta a continuación.

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

PrimerProyecto.py

1.1. Grafo simple no dirigido acíclico

Primeramente se definirá ciertos conceptos. Un camino es un subconjunto de A (que puede ser A) y recorre un subconjunto de N (que puede incluir todo N). Un ciclo es un camino que une por lo menos 3 nodos distintos, y el nodo de inicio es el mismo que el nodo final.

En la figura 25 se representa un isómero de carbono, en el cual los nodos son los diferentes elementos del compuesto, y las aristas son los enlaces que tienen un elemento con otro.

El código que da paso a la figura antes mencionada es el siguiente.

```
1 #Primero simple no dirigido aciclico
2 H=nx.Graph()
3
4 H.add_edges_from([('C','Cl'), ('C','H'), ('C','F'),('C','Br')])
5
6 nx.draw(H, node_color="white", node_size=800, with_labels=True,
7          font_weight="bold", edgecolors="black")
8
9 plt.savefig('Primero.eps', format='eps', dpi=1000)
```

PrimerProyecto.py

La última linea del código se usa para guardar una imagen tipo extensión eps.

1.2. Grafo simple no dirigido cíclico

En la sección anterior se vió que es un ciclo en un grafo, en la figura 27 se observa un grafo que representa la caja de cambios de un carro, donde el nodo Öës estar en neutral.

Es no dirigido porque es lo mismo si pasa de la primera marcha a la segunda, análogamente con las demás marchas; es cíclica ya que eso se repite muchas veces durante un mismo trayecto.

1.3. Grafo simple no dirigido reflexivo

Un nodo reflexivo es el que tiene un arista que va hacia sí mismo, es decir, para cualquier $u \in V$ existe $(u, u) \in A$. Entonces se puede decir que un grafo reflexivo es el que tiene nodos reflexivos. En este documento los nodos reflexivos son de color rojo.

En el siguiente ejemplo 29 se tiene el horario de cierto profesor, los nodos muestran las materias y las aristas las horas que da de clase en una semana especifica. Se puede apreciar que las materias de Calculo 1, Algebra y Conjuntos hay repeticiones (porque están de color rojo) lo que quiere decir que después de ir, por ejemplo, a Algebra, la siguiente clase podría ser Algebra, o salir de esa clase e ir a otra distinta.

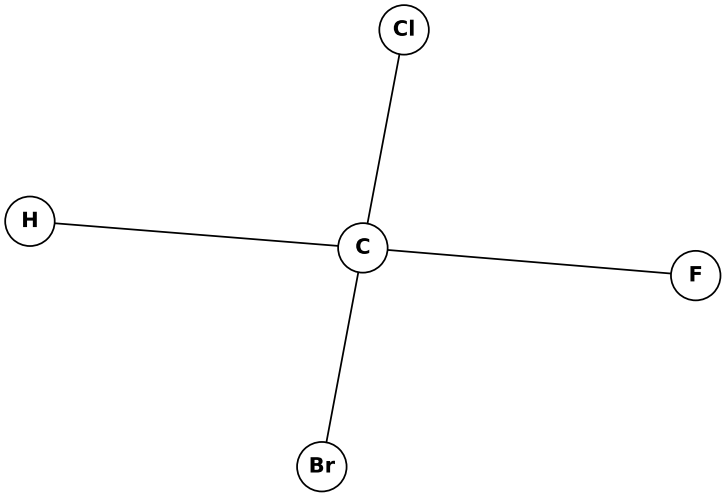


Figura 1: Grafo no dirigido acíclico

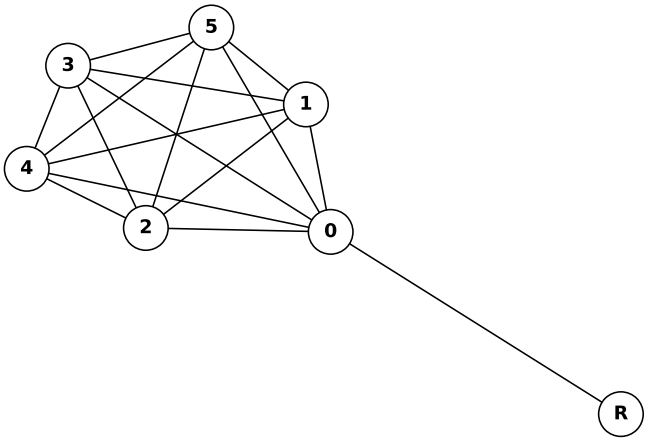


Figura 2: Grafo no dirigido cíclico

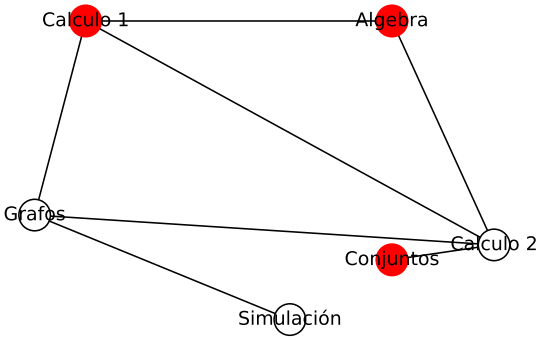


Figura 3: Grafo no dirigido reflexivo

Para llevar a cabo este grafo se hizo dos conjuntos llamados nodos1 y nodos2, en las siguientes lineas (15) y (17) al dibujar nodos1 y nodos2 se le asignan los colores y las posiciones ya establecidas en el diccionario pos

```
#Tercero simple no dirigido reflexivo
G=nx.Graph()
G.add_edges_from([('Calculo 1','Algebra'),('Calculo 1','Calculo 2'),
                  ('Calculo 1','Grafos'),('Algebra','Calculo 2'),
                  ('Calculo 2','Grafos'),('Simulaci n','Grafos'),
                  ('Calculo 2','Conjuntos')])

nodos1 = {'Calculo 1','Algebra','Conjuntos'}
nodos2 = {'Calculo 2','Simulaci n','Grafos'}

pos = {'Calculo 1':(20, 35), 'Algebra':(50,35),
       'Calculo 2':(60, 20), 'Simulaci n':(40,15),
       'Grafos':(15,22), 'Conjuntos':(50,19)}

nx.draw_networkx_nodes(G, pos, nodelist=nodos1,node_size=400,
                       node_color='r', node_shape='o')
nx.draw_networkx_nodes(G, pos, nodelist=nodos2,node_size=400,
                       node_color='w', node_shape='o',
                       edgecolors="k")

nx.draw_networkx_edges(G, pos)
nx.draw_networkx_labels(G, pos)
plt.axis('off')

plt.savefig('Tercero.eps', format='eps', dpi=1000)
```

PrimerProyecto.py

1.4. Grafo simple dirigido acíclico

Un grafo dirigido es aquel que se especifica la dirección de la arista, es decir, si $(u, v) \in A$ entonces $(v, u) \notin A$.

Un ejemplo para la figura 31 puede ser un centro de distribución de materiales, que entrega a varios proveedores que simbolizan los nodos y cada arista es el flujo de material.

Para que este ejemplo sea considerado como un grafo simple solo se puede enviar material.

Para realizar el grafo se hizo uso de la función DiGraph() y con ella las aristas se vuelven dirigidas

```
#Cuarto simple dirigido aciclico
H=nx.DiGraph()
H.add_edges_from([('a','b'), ('a','c'), ('a','d'),('d','e'),
                  ('e','f'), ('c','a')])
nx.draw(H, node_color="white", node_size=800, with_labels=True,
        font_weight="bold", edgecolors="black")
plt.savefig('Cuarto.eps', format='eps', dpi=1000)
```

PrimerProyecto.py

1.5. Grafo simple dirigido cíclico

Al igual que en el grafo no dirigido, este grafo tiene la característica de que contiene un ciclo. Si queremos representar como diagrama de flujo, un videojuego que se elige entrar solo (solitario) o en grupo (cooperativo) y después de esa opción se va a jugar, y cuando termina el juego, se devuelve al menú principal.

En la figura 33 se aprecia el ciclo y las flechas que nos dice que es un grafo dirigido

1.6. Grafo simple dirigido reflexivo

Supongamos que se tiene el diagrama de flujo de los procesos para la elaboración de productos en una empresa. El primer paso es la recepción donde se verifica la calidad de los productos que

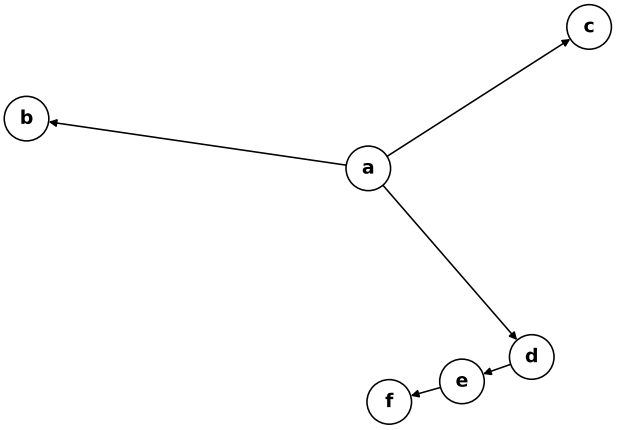


Figura 4: Grafo dirigido acíclico

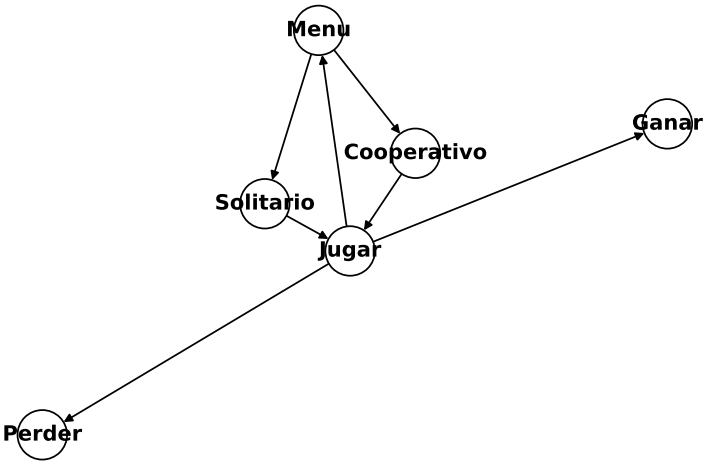


Figura 5: Grafo dirigido cíclico

entran y se cuentan los mismos con el fin de ver si lo que llegó es lo que se pidió. En caso de no cumplir con el conteo, se vuelve a verificar. Si está en mal estado se puede llevar a calidad e intentar recuperar el producto, pero si el producto no queda del todo bien, sigue entrando a calidad.

Este caso puede ser representado mediante un grafo, como lo muestra la figura 35 donde los nodos representan los procesos y las aristas es el flujo de los productos. Los nodos reflexivos son como el nodo 1 o calidad, que pueden entrar varias veces al mismo proceso.

1.7. Multigrafo no dirigido acíclico

Un multigrafo es un grafo en el cual puede haber mas de un arista entre cada par de nodos, esto es, tiene multiplicidades en sus nodos. Gráficamente se representa con dos lineas, pero en los siguientes gráficos solo se muestra una linea.

Este tipo de grafos se pueden usar para representar los vuelos de cierto tipo de aerolínea entre algunos países, como lo representa en la figura 19 donde los nodos son los países y las aristas son la cantidad de vuelos.

Para dibujar estos grafos se usa la función de networkx MultiGraph() que te permite agrgar aristas que, aunque vistos como conjuntos son iguales, importan porque representan flujos (cantidades de vuelos en nuestro ejemplo) diferentes

```
#Septimo multigrafo no dirigido aciclico
H=nx.MultiGraph()
H.add_edges_from([('Mexico','EUA'), ('EUA','Mexico'),
                  ('EUA','Espa a'), ('Espa a','EUA'),
                  ('Mexico','Venezuela'), ('Venezuela','Mexico'),
                  ('Cuba','Mexico'), ('Mexico','Cuba'),
                  ('EUA','Alemania'), ('Alemania','EUA')])
nx.draw(H, node_color="white", node_size=800, with_labels=True,
        font_weight="bold", edgecolors="black")
plt.savefig('Septimo.eps', format='eps', dpi=1000)
```

PrimerProyecto.py

1.8. Multigrafo no dirigido cíclico

Al igual que en el grafo simple, este grafo contiene por lo menos un ciclo. Para visualizarlo mejor se hace referencia al ejemplo anterior, en la figura 20 vemos que ahora, Venezuela y Cuba tienen vuelos entre sí, lo mismo que Venezuela y España, formándose dos ciclos

Lo diferente de este gráfico con el anterior consiste en que se agregan los aristas para conectar Cuba-Venezuela y España-Venezuela

1.9. Multigrafo no dirigido reflexivo

Como ya se ha comentado, los nodos reflexivos se pintan de rojo. El ejemplo que se tomó para este tipo de grafos es donde cada nodo representa una ciudad y un arco representa la capacidad del drenaje pluvial, como la capacidad es la misma en ambas direcciones es no dirigido, ya que hay drenaje que conecta con otros ductos dentro de la misma ciudad entonces el grafo es reflexivo.

1.10. Multigrafo dirigido acíclico

En la figura 22 se representa a 5 estaciones de trabajo (work station) las cuales están conectadas a un servidor. Los aristas representan el flujo de información que existe entre las estaciones de trabajo y el servidor, cada computadora es independiente, salvo por el servidor, por lo que toda conexión entre computadoras es a través del mismo.

El código necesario para la gráfica es el siguiente.

```
#Decimo multigrafo dirigido aciclico
H=nx.MultiDiGraph()
H.add_edges_from([('Jose','Claudia'), ('Claudia','Jose'),
                  ('Claudia','Agustin'), ('Agustin','Claudia'),
                  ('Agustin','Priscila'), ('Agustin','Felipe'),
```

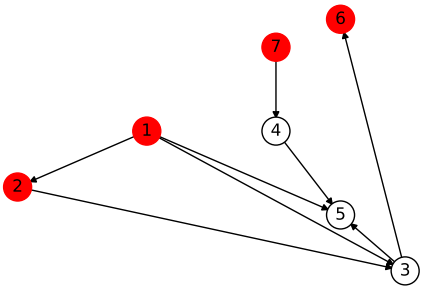


Figura 6: Grafo dirigido reflexivo

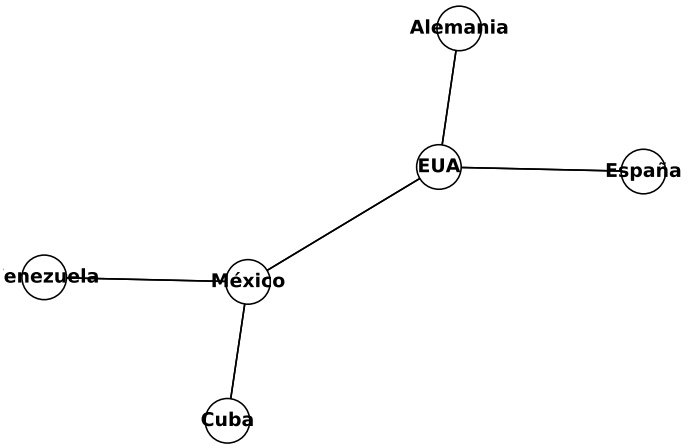


Figura 7: Multigrafo no dirigido acíclico

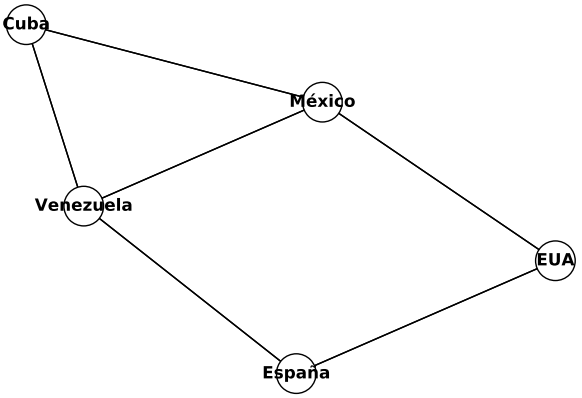


Figura 8: Multigrafo no dirigido cíclico

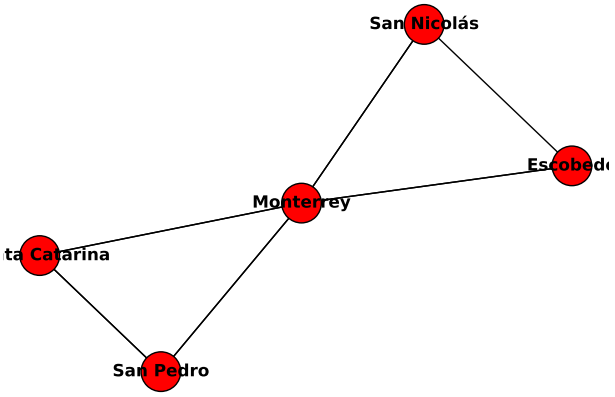


Figura 9: Multigrafo no dirigido reflexivo

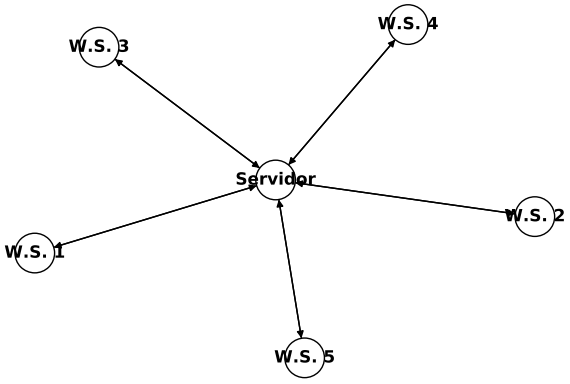


Figura 10: Multigrafo dirigido acíclico

```
                    ('Agustin','Cristina'), ('Cristina','Carlos'),
                    ('Carlos','Pedro'), ('Pedro','Carlos'),
                    ('Priscila','Carlos'), ('Carlos','Priscila')])
nx.draw(H, node_size=800, with_labels=True,
        font_weight="bold", edgecolors="black", node_color='white')
plt.savefig('Decimo.eps', format='eps', dpi=1000)
```

PrimerProyecto.py

Donde vemos que se usa la función MultiDiGraph(), al igual que en DiGraph() un par ordenado de vertices se convierte en una flecha.

1.11. Multigrafo dirigido cíclico

Para este ejemplo se tiene la transmisión de enfermedades en una oficina, donde los nodos son las personas y los aristas representan la probabilidad de transmitir la enfermedad, que puede ser distinta si es de Felipe a Agustín o de Agustín a Felipe, lo mismo ocurre con los demás nodos

1.12. Multigrafo dirigido reflexivo

Supóngase que se tiene un evento probabilístico, donde cada evento tiene una cierta probabilidad dado el evento anterior, y un evento se puede repetir. En la figura 24, se puede ver que los nodos son los eventos, ya que se puede repetir el evento entonces el grafo es reflexivo, y las aristas son las probabilidades de ocurrencia del evento j dado que ocurrió el evento i

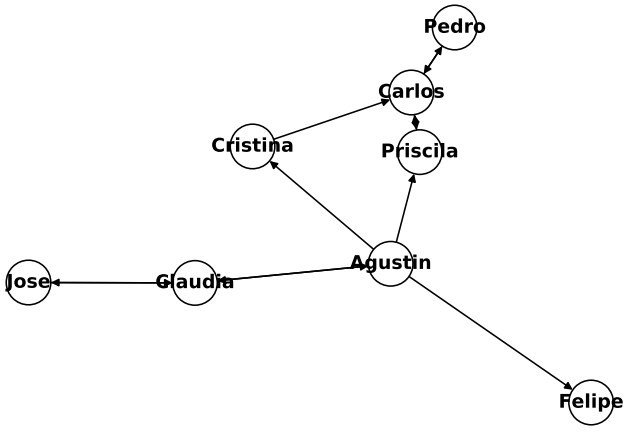


Figura 11: Multigrafo dirigido cíclico

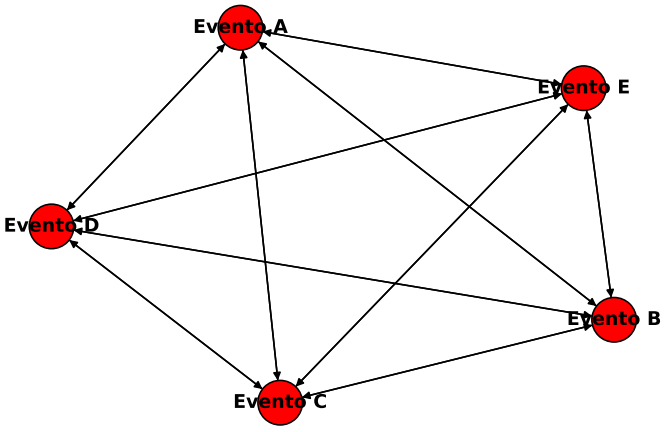


Figura 12: Multigrafo dirigido reflexivo

1.13. Tarea uno corregida

Introducción

Un grafo está formado por un conjunto de nodos denotados por N, y un conjunto de la forma (u, v) llamado aristas al que nos referiremos por A. Dada su composición es conveniente ver un grafo como un dibujo. Los grafos se pueden usar para ilustrar procesos, rutas, cambios de estados en ciertos procesos y hasta un árbol genealógico. En este documento se muestra los tipos grafos, cuales son las condiciones que tienen que cumplir y un ejemplo de su aplicación.

Para hacer una mejor referencia a los grafos, los vamos a dibujar en python y para eso necesitamos hacer uso de dos librerias, como se presenta a continuación.

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

PrimerProyecto.py

1.14. Grafo simple no dirigido acíclico

Primeramente se definen ciertos conceptos. Un camino es un subconjunto de A (que puede ser A) y recorre un subconjunto de N (que puede incluir todo N). Un ciclo es un camino que une por lo menos tres nodos distintos, y el nodo de inicio es el mismo que el nodo final.

En la figura 25 se representa un isómero de carbono, en el cual los nodos son los diferentes elementos del compuesto, y las aristas son los enlaces que tienen un elemento con otro.

El código que da paso a la figura antes mencionada es el siguiente.

```
1 #Primero simple no dirigido aciclico
2 H=nx.Graph()
3
4 H.add_edges_from([('C','Cl'), ('C','H'), ('C','F'),('C','Br')])
5
6 nx.draw(H, node_color="white", node_size=800, with_labels=True,
7         font_weight="bold", edgecolors="black")
8
9 plt.savefig('Primero.eps', format='eps', dpi=1000)
```

PrimerProyecto.py

La última línea del código se usa para guardar una imagen tipo extensión eps.

1.15. Grafo simple no dirigido cíclico

Un ejemplo de un grafo ciclico es el de la figura 27, donde se observa un grafo que representa la caja de cambios de un carro, donde el nodo O es estar en neutral.

Es no dirigido porque es lo mismo si pasa de la primera marcha a la segunda, análogamente con las demás marchas; es cíclica ya que eso se repite muchas veces durante un mismo trayecto.

1.16. Grafo simple no dirigido reflexivo

Un nodo reflexivo es el que tiene un arista que va hacia sí mismo, es decir, para cualquier $u \in V$ existe $(u, u) \in A$. Entonces se puede decir que un grafo reflexivo es el que tiene nodos reflexivos. En este documento los nodos reflexivos son de color rojo.

En el ejemplo 29 se tiene el horario de cierto profesor, los nodos muestran las materias y las aristas las horas que da de clase en una semana especifica. Se puede apreciar que las materias de Cálculo 1, Álgebra y Conjuntos hay repeticiones (porque están de color rojo) lo que quiere decir que después de ir, por ejemplo, a Álgebra, la siguiente clase podría ser Álgebra, o salir de esa clase e ir a otra distinta.

Para llevar a cabo este grafo se hizo dos conjuntos llamados `nodos1` y `nodos2`, en las siguientes líneas (15) y (17) al dibujar `nodos1` y `nodos2` se le asignan los colores y las posiciones ya establecidas en el diccionario `pos`.

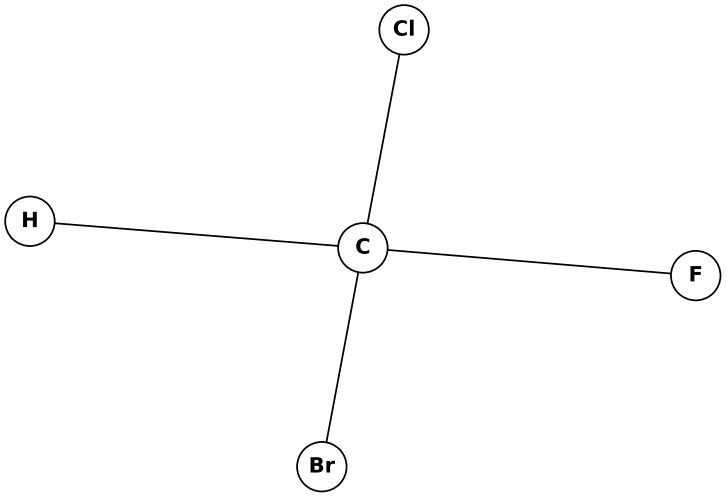


Figura 13: Grafo no dirigido acíclico

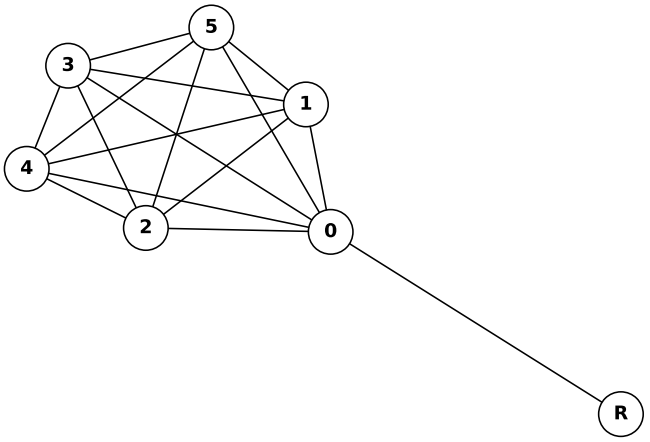


Figura 14: Grafo no dirigido cíclico

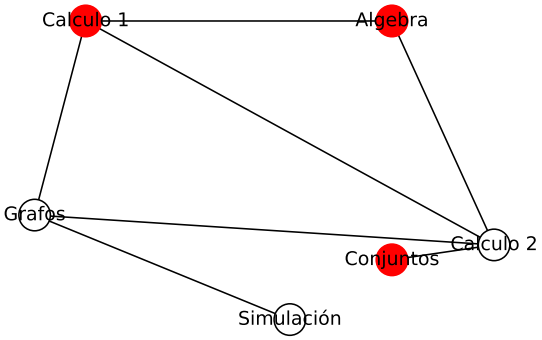


Figura 15: Grafo no dirigido reflexivo

```
#Tercero simple no dirigido reflexivo
G=nx.Graph()
G.add_edges_from([('Calculo 1','Algebra'),('Calculo 1','Calculo 2'),
                  ('Calculo 1','Grafos'),('Algebra','Calculo 2'),
                  ('Calculo 2','Grafos'),('Simulaci n','Grafos'),
                  ('Calculo 2','Conjuntos')])

nodos1 = {'Calculo 1','Algebra','Conjuntos'}
nodos2 = {'Calculo 2','Simulaci n','Grafos'}

pos = {'Calculo 1':(20, 35), 'Algebra':(50,35),
       'Calculo 2':(60, 20), 'Simulaci n':(40,15),
       'Grafos':(15,22), 'Conjuntos':(50,19)}

nx.draw_networkx_nodes(G, pos, nodelist=nodos1,node_size=400,
                       node_color='r', node_shape='o')
nx.draw_networkx_nodes(G, pos, nodelist=nodos2,node_size=400,
                       node_color='w', node_shape='o',
                       edgecolors="k")

nx.draw_networkx_edges(G, pos)
nx.draw_networkx_labels(G, pos)
plt.axis('off')

plt.savefig('Tercero.eps', format='eps', dpi=1000)
```

PrimerProyecto.py

1.17. Grafo simple dirigido acíclico

Un grafo dirigido es aquel que se especifica la dirección de la arista, es decir, si $(u, v) \in A$ entonces $(v, u) \notin A$.

Un ejemplo para la figura 31 puede ser un centro de distribución de materiales, que entrega a varios proveedores que simbolizan los nodos y cada arista es el flujo de material.

Para que este ejemplo sea considerado como un grafo simple solo se puede enviar material.

Para realizar el grafo se hizo uso de la función `DiGraph()` y con ella las aristas se vuelven dirigidas.

```
#Cuarto simple dirigido aciclico
H=nx.DiGraph()
H.add_edges_from([('a','b'), ('a','c'), ('a','d'),('d','e'),
                  ('e','f'), ('c','a')])
nx.draw(H, node_color="white", node_size=800, with_labels=True,
        font_weight="bold", edgecolors="black")
plt.savefig('Cuarto.eps', format='eps', dpi=1000)
```

PrimerProyecto.py

1.18. Grafo simple dirigido cíclico

Al igual que en el grafo no dirigido, este grafo tiene la característica de que contiene un ciclo. Si queremos representar como diagrama de flujo, un videojuego que se elige entrar solo (solitario) o en grupo (cooperativo) y después de esa opción se va a jugar, y cuando termina el juego, se devuelve al menú principal.

En la figura 33 se aprecia el ciclo y las flechas que nos dice que es un grafo dirigido.

1.19. Grafo simple dirigido reflexivo

Supongamos que se tiene el diagrama de flujo de los procesos para la elaboración de productos en una empresa. El primer paso es la recepción donde se verifica la calidad de los productos que entran y se cuentan los mismos con el fin de ver si lo que llegó es lo que se pidió. En caso de no

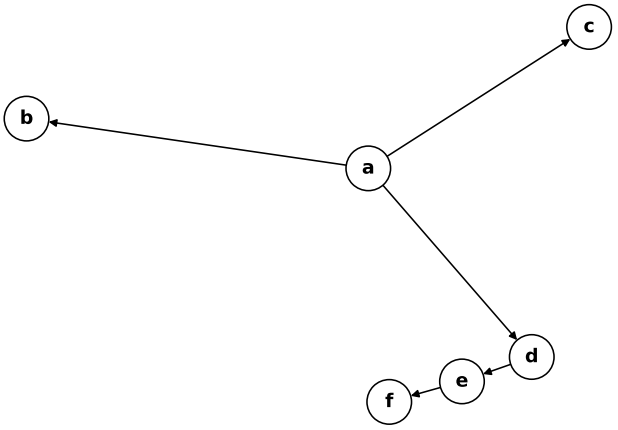


Figura 16: Grafo dirigido acíclico

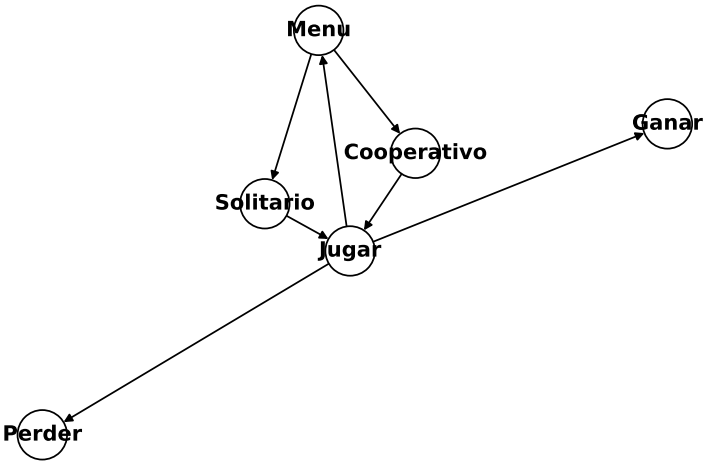


Figura 17: Grafo dirigido cíclico

cumplir con el conteo, se vuelve a verificar. Si está en mal estado se puede llevar a calidad e intentar recuperar el producto, pero si el producto no queda del todo bien, sigue entrando a calidad.

Este caso puede ser representado mediante un grafo, como lo muestra la figura 35 donde los nodos representan los procesos y las aristas es el flujo de los productos. Los nodos reflexivos son como el nodo 1 o calidad, que pueden entrar varias veces al mismo proceso.

1.20. Multigrafo no dirigido acíclico

Un multigrafo es un grafo en el cual puede haber más de un arista entre cada par de nodos, esto es, tiene multiplicidades en sus nodos. Gráficamente se representa con dos líneas, pero en los siguientes gráficos solo se muestra una línea.

Este tipo de grafos se pueden usar para representar los vuelos de cierto tipo de aerolínea entre algunos países, como lo representa en la figura 19 donde los nodos son los países y las aristas son la cantidad de vuelos.

Para dibujar estos grafos se usa la función de networkx `MultiGraph()` que te permite agrgar aristas que, aunque vistos como conjuntos son iguales, importan porque representan flujos (cantidades de vuelos en nuestro ejemplo) diferentes

```
#Septimo multigrafo no dirigido aciclico
H=nx.MultiGraph()
H.add_edges_from([('Mexico','EUA'), ('EUA','Mexico'),
                  ('EUA','Espa a'), ('Espa a','EUA'),
                  ('Mexico','Venezuela'), ('Venezuela','Mexico'),
                  ('Cuba','Mexico'), ('Mexico','Cuba'),
                  ('EUA','Alemania'), ('Alemania','EUA')])
nx.draw(H, node_color="white", node_size=800, with_labels=True,
        font_weight="bold", edgecolors="black")
plt.savefig('Septimo.eps', format='eps', dpi=1000)
```

PrimerProyecto.py

1.21. Multigrafo no dirigido cíclico

Al igual que en el grafo simple, este grafo contiene por lo menos un ciclo. Para visualizarlo mejor se hace referencia al ejemplo anterior, en la figura 20 se aprecia que ahora, Venezuela y Cuba tienen vuelos entre sí, lo mismo que Venezuela y España, formándose dos ciclos

Lo diferente de este gráfico con el anterior consiste en que se agregan los aristas para conectar Cuba-Venezuela y España-Venezuela

1.22. Multigrafo no dirigido reflexivo

Como ya se ha comentado, los nodos reflexivos se pintan de rojo. El ejemplo que se tomó para este tipo de grafos es donde cada nodo representa una ciudad y un arco representa la capacidad del drenaje pluvial, como la capacidad es la misma en ambas direcciones es no dirigido, ya que hay drenaje que conecta con otros ductos dentro de la misma ciudad entonces el grafo es reflexivo.

1.23. Multigrafo dirigido acíclico

En la figura 22 se representa a cinco estaciones de trabajo (work station) las cuales están conectadas a un servidor. Los aristas representan el flujo de información que existe entre las estaciones de trabajo y el servidor, cada computadora es independiente, salvo por el servidor, por lo que toda conexión entre computadoras es a través del mismo.

El código necesario para la gráfica es el siguiente, donde vemos que se usa la función `MultiDiGraph()`, al igual que en `DiGraph()` un par ordenado de vertices se convierte en una flecha.

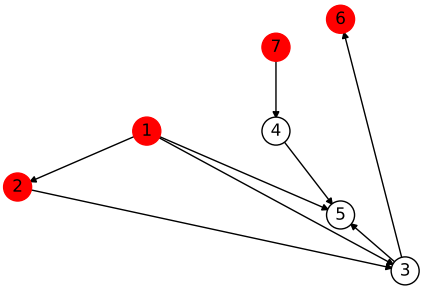


Figura 18: Grafo dirigido reflexivo

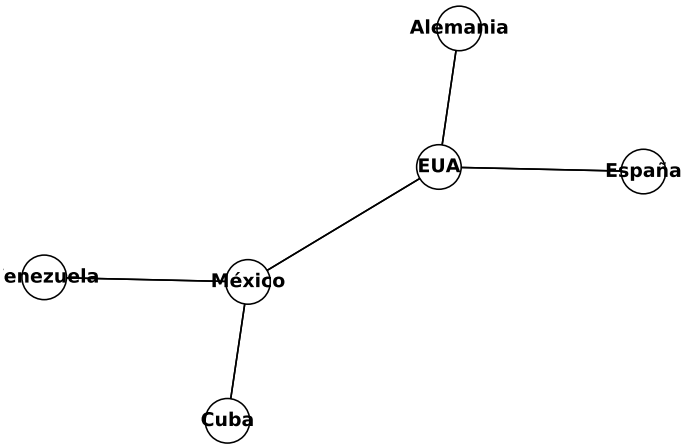


Figura 19: Multigrafo no dirigido acíclico

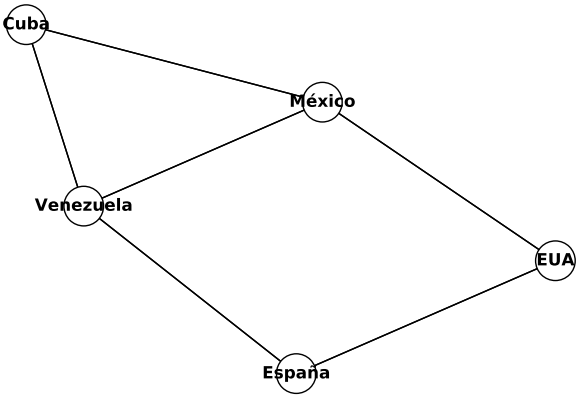


Figura 20: Multigrafo no dirigido cíclico

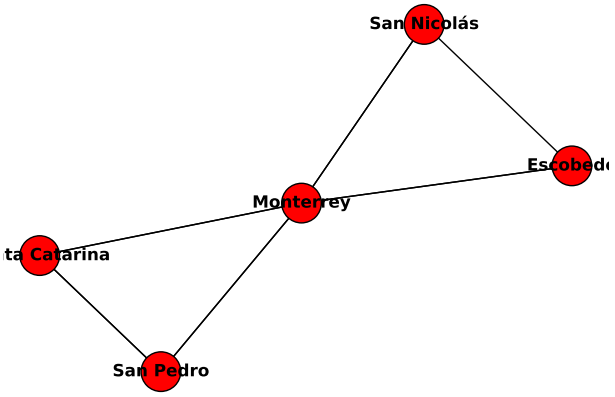


Figura 21: Multigrafo no dirigido reflexivo

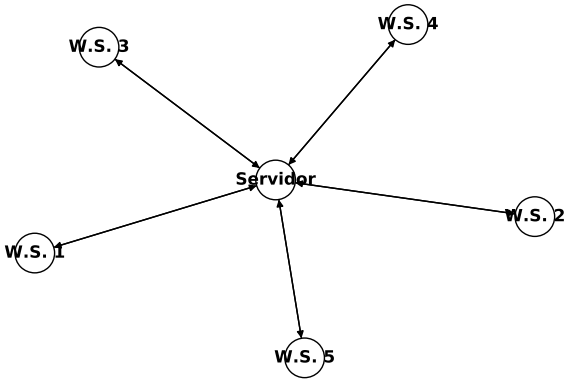


Figura 22: Multigrafo dirigido acíclico

```
#Decimo multigrafo dirigido aciclico
H=nx.MultiDiGraph()
H.add_edges_from([('Jose','Claudia'), ('Claudia','Jose'),
                  ('Claudia','Agustin'), ('Agustin','Claudia'),
                  ('Agustin','Priscila'), ('Agustin','Felipe'),
                  ('Agustin','Cristina'), ('Cristina','Carlos'),
                  ('Carlos','Pedro'), ('Pedro','Carlos'),
                  ('Priscila','Carlos'), ('Carlos','Priscila')])
nx.draw(H, node_size=800, with_labels=True,
        font_weight="bold", edgecolors="black", node_color='white')
plt.savefig('Decimo.eps', format='eps', dpi=1000)
```

PrimerProyecto.py

1.24. Multigrafo dirigido cíclico

Para este ejemplo se tiene la transmisión de enfermedades en una oficina, donde los nodos son las personas y los aristas representan la probabilidad de transmitir la enfermedad, que puede ser distinta si es de Felipe a Agustín o de Agustín a Felipe, lo mismo ocurre con los demás nodos

1.25. Multigrafo dirigido reflexivo

Supóngase que se tiene un evento probabilístico, donde cada evento tiene una cierta probabilidad dado el evento anterior, y un evento se puede repetir. En la figura 24 se puede ver que los nodos son los eventos, ya que se puede repetir el evento entonces el grafo es reflexivo, y las aristas son las probabilidades de ocurrencia del evento j dado que ocurrió el evento i

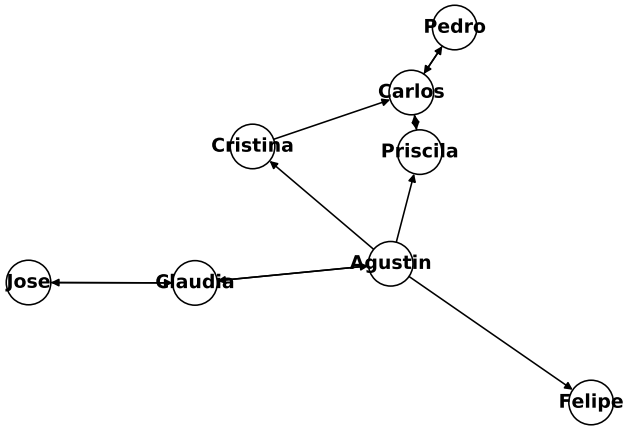


Figura 23: Multigrafo dirigido cíclico

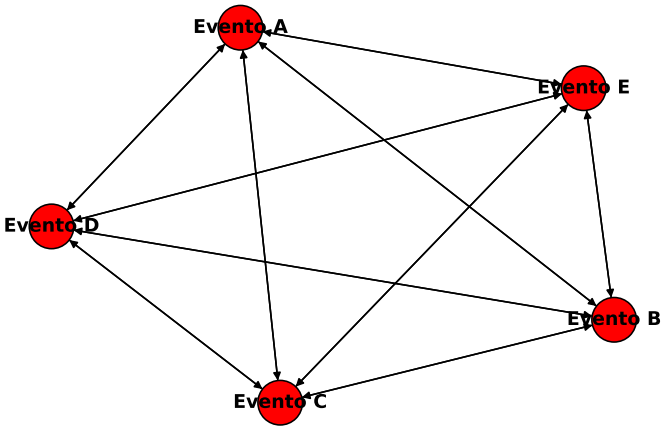


Figura 24: Multigrafo dirigido reflexivo

2. Tarea dos original

Introducción

Un grafo se puede dibujar de muchas maneras ya que por lo que representan aveces es conveniente verlo de cierto modo. En este documento se muestra algunas opciones presentes en la librería de `Networkx`

2.1. Diseño bipartito

Un grafo bipartito es aquel donde sus nodos se pueden repartir en dos conjuntos disjuntos.

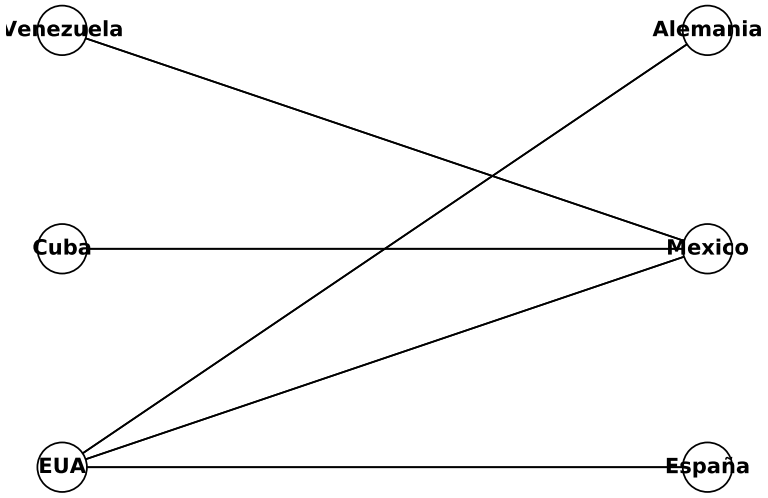
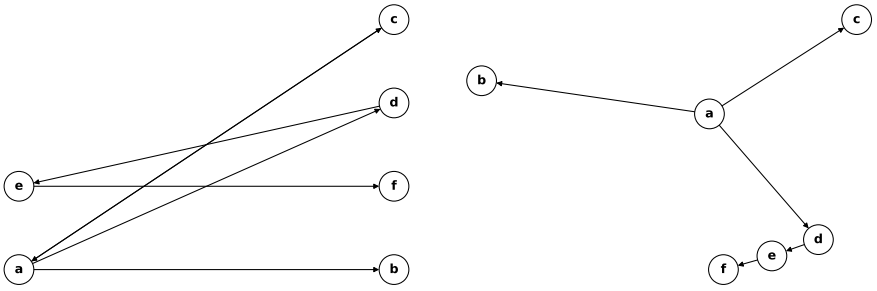


Figura 25: Grafo bipartito



(a) Grafo bipartito (b) Grafo dirigido acíclico

Figura 26: Comparativa entre grafo con acomodo bipartito `vs` grafo con acomodo por default

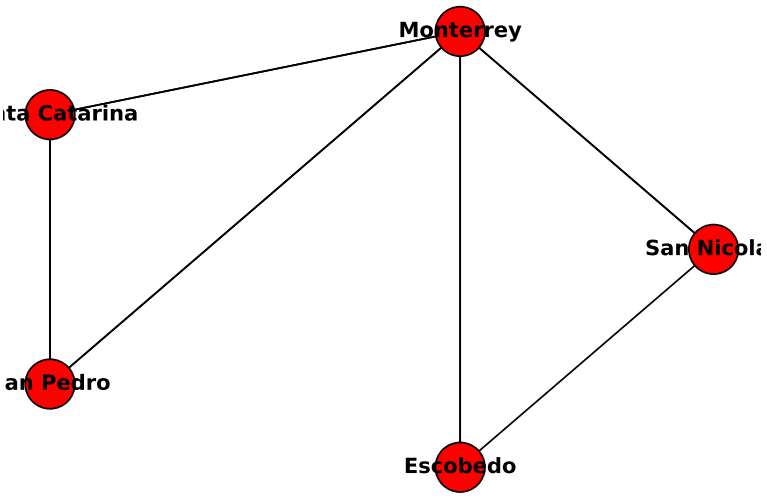
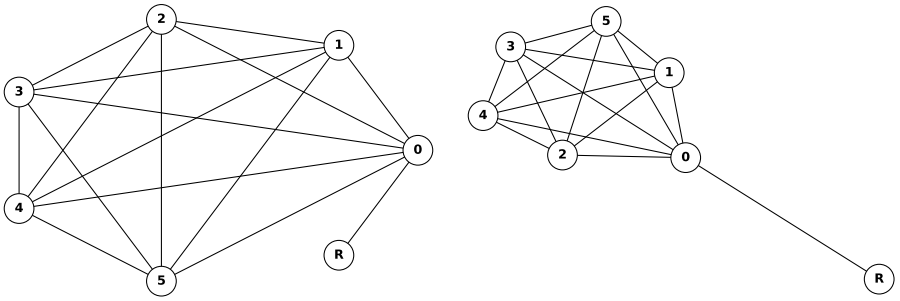


Figura 27: Grafo circular



(a) Grafo circular (b) Grafo no dirigido ciclico

Figura 28: Comparativa entre grafo con acomodo circular `vs` grafo acomodo por default

El código que da paso a la figura 25 es el siguiente.

```
H=nx.MultiGraph()
H.add_nodes_from(['Venezuela','Mexico','Cuba'], bipartite=0)
H.add_nodes_from(['Espa a','EUA','Alemania'], bipartite=1)
H.add_edges_from([('Mexico','EUA'), ('EUA','Mexico'),
                  ('EUA','Espa a'), ('Espa a','EUA'),
                  ('Mexico','Venezuela'), ('Venezuela','Mexico'),
                  ('Cuba','Mexico'), ('Mexico','Cuba'),
                  ('EUA','Alemania'), ('Alemania','EUA')])
X,Y=bipartite.sets(H)
pos=dict()
pos.update( (n, (1, i)) for i, n in enumerate(X) )
pos.update( (n, (2, i)) for i, n in enumerate(Y) )
nx.draw(H, pos, node_color="white", node_size=800, with_labels=True,
        font_weight="bold", edgecolors="black")
plt.savefig('SSeptimo.eps', format='eps', dpi=1000)
```

Grafos_layout.py

2.2. Diseño circular

Como lo dice su nombre, un grafo de este tipo es el que tiene una distribución circular. Para hacer estos grafos se usa la función `circular_layout()`

Para graficar la red de la figura 28, grafo con red circular se utilizó el siguiente código.

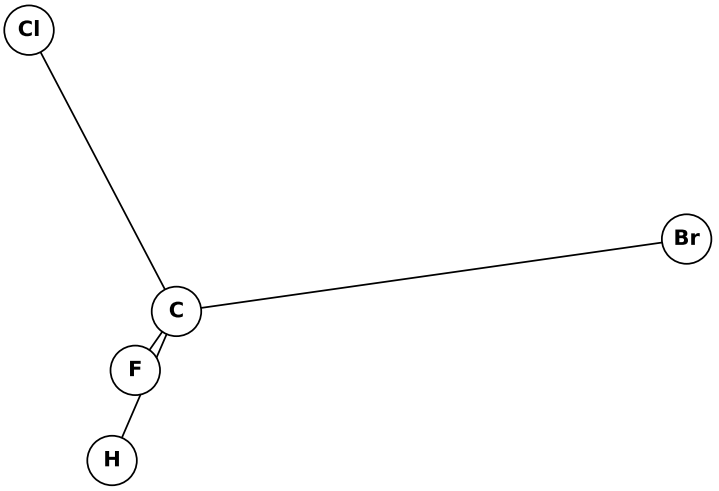
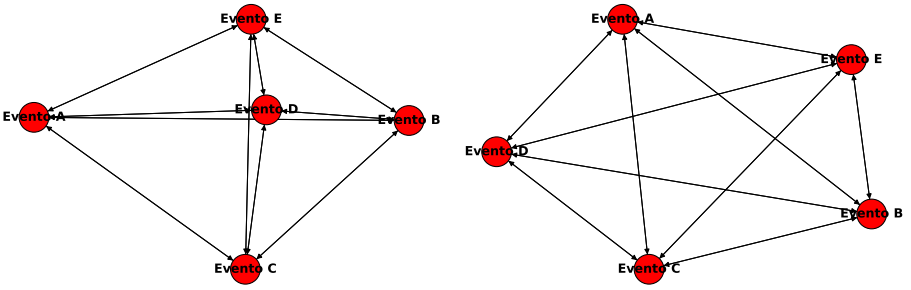


Figura 29: Grafo spectral



(a) Grafo spectral (b) Multigrafo dirigido reflexivo

Figura 30: Comparativa entre grafo con acomodo espectral `vs` grafo con acomodo por default

```
H=nx.complete_graph(6)
H.add_nodes_from([1,2,3,4,5])
H.add_edge(0,'R')
pos = nx.circular_layout(H)
nx.draw(H, pos, node_color="white", node_size=800, with_labels=True,
        font_weight="bold", edgecolors="black")
plt.savefig('SSegundo.eps', format='eps', dpi=1000)
```

Grafos_layout.py

2.3. Diseño spectral

En este acomodo del grafo se utiliza los vectores propios de una matriz como coordenadas cartesianas de los nodos de la gráfica. Para acceder a este diseño de grafos basta con ingresar la función `spectral_layout()`.

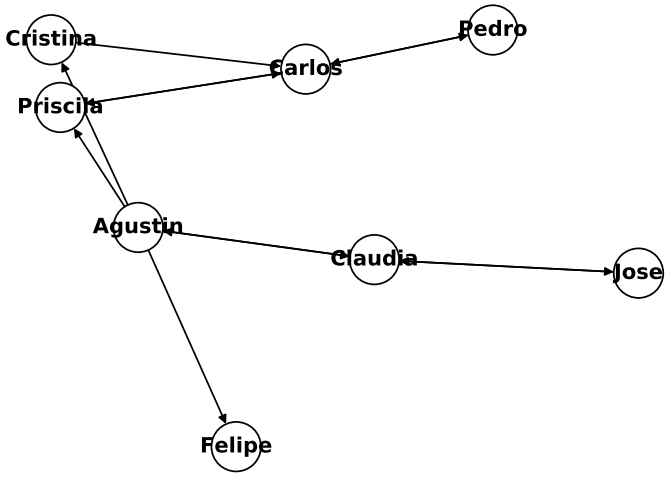


Figura 31: Grafo spring

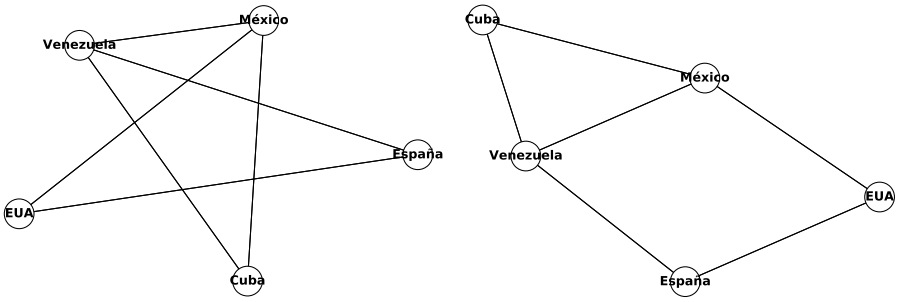
El código con el que se hizo el grafo de la figura 30 con acomodo espectral es.

```
G=nx.MultiDiGraph()
G.add_weighted_edges_from([('Evento A','Evento B',1),
                           ('Evento B','Evento A',2),
                           ('Evento A','Evento C',3),
                           ('Evento C','Evento A',2),
                           ('Evento A','Evento D',4),
                           ('Evento D','Evento A',1),
                           ('Evento A','Evento E',3),
                           ('Evento E','Evento A',2),
                           ('Evento B','Evento C',4),
                           ('Evento C','Evento B',1),
                           ('Evento B','Evento D',5),
                           ('Evento D','Evento B',1),
                           ('Evento B','Evento E',2),
                           ('Evento E','Evento B',3),
                           ('Evento C','Evento D',4),
                           ('Evento D','Evento C',2),
                           ('Evento C','Evento E',3),
                           ('Evento E','Evento C',1),
                           ('Evento D','Evento E',5),
                           ('Evento E','Evento D',2)])
pos=nx.spectral_layout(G)
nx.draw(G, pos, node_size=800, with_labels=True,
        font_weight="bold", edgecolors="black", node_color='red')
plt.savefig('DDoceavo.eps', format='eps', dpi=1000)
```

Grafos_layout.py

2.4. Diseño spring

Esta función hace una analogía de las fuerzas de física, el objetivo de este algoritmo es encontrar un acomodo de los nodos donde se minimice la suma de las fuerzas. Para obtener este tipo de grafo se utiliza la función `spring_layout()`. Algunos ejemplos de muestran a continuación.



(a) Grafo spring (b) Multigrafo no dirigido ciclico

Figura 32: Comparativa entre grafo con acomodo spring `vs` grafo con acomodo por default

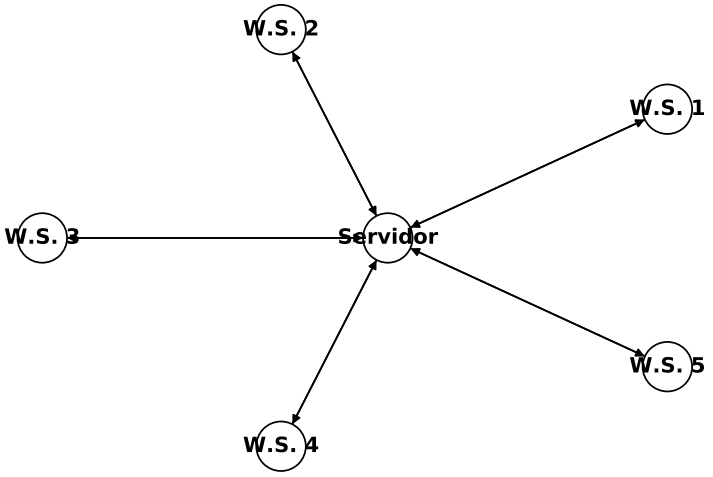


Figura 33: Grafo con diseño Kamada Kawai

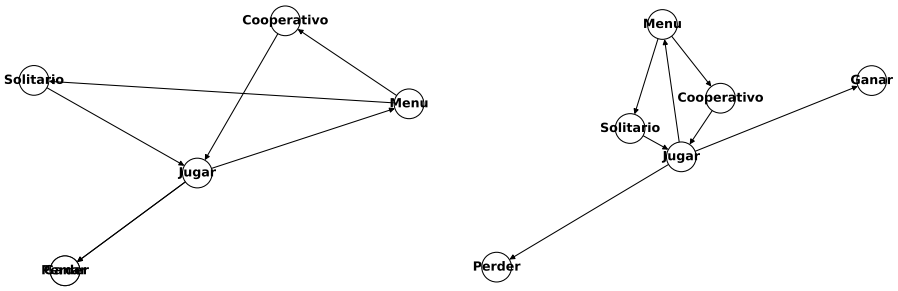
El siguiente código muestra como se hace el grafo de la figura 32.

```
H=nx.MultiGraph()
H.add_edges_from([('México','EUA'), ('EUA','México'),('EUA','España'),
                  ('España','EUA'), ('México','Venezuela'), ('Venezuela','México'),
                  ('Cuba','México'), ('México','Cuba'),('Cuba','Venezuela'), ('Venezuela','Cuba'),
                  ('Venezuela','España'), ('España','Venezuela')])
pos=nx.spring_layout(H)
nx.draw(H, pos, node_color="white", node_size=800, with_labels=True,
        font_weight="bold", edgecolors="black")
plt.savefig('OOctavo.eps', format='eps', dpi=1000)
```

Grafos_layout.py

2.5. Diseño Kamada Kawai

Este tipo diseño se utiliza cuando se quiere reducir el número de cruces de los aristas, como se muestra en los siguientes gráficos. Para hacer uso de este diseño se usa la función `kamada_kawai_layout()`.



(a) Grafo diseño Kamada Kawai (b) Grafo dirigido cíclico

Figura 34: Comparativa entre grafo con acomodo Kamada Kawai `vs` grafo acomodo por default

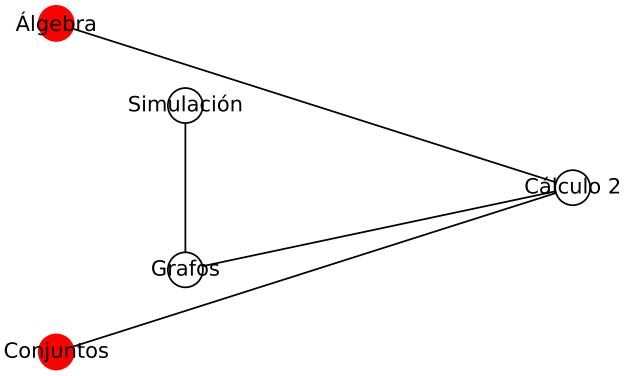


Figura 35: Grafo dirigido reflexivo

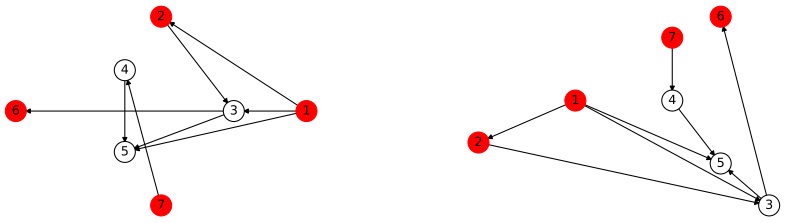
El código con el que se hizo la red de la figura 34 con el acomodo Kamada Kawai es.

```
H=nx.DiGraph()
H.add_edges_from([('Menu','Cooperativo'),('Menu','Solitario'),
                  ('Solitario','Jugar'),('Cooperativo','Jugar'),
                  ('Jugar','Menu'),('Jugar','Ganar'),
                  ('Jugar','Perder')])
pos=nx.kamada_kawai_layout(H)
nx.draw(H, pos, node_color="white", node_size=800, with_labels=True,
        font_weight="bold", edgecolors="black")
plt.savefig('QQuinto.eps', format='eps', dpi=1000)
```

Grafos_layout.py

2.6. Diseño shell

Este algoritmo se basa en acomodar los nodos en circulos concéntricos.



(a) Grafo diseño shell (b) Grafo dirigido cíclico

Figura 36: Comparativa entre grafo con acomodo shell `vs` grafo acomodo por default

Para hacer la red de la figura 35 con el acomodo shell es.

```
G=nx.Graph()
G.add_edges_from([('  lgebra  ','C lculo 2'),('C lculo 2','Grafos'),
                  ('Simulaci n','Grafos'),('C lculo 2','Conjuntos')])

nodos1 = {'  lgebra  ','Conjuntos'}
nodos2 = {'C lculo 2','Simulaci n','Grafos'}

shells=[['C lculo 2','Simulaci n','Grafos'],['C lculo 1','  lgebra  ',
        'Conjuntos']]

pos=nx.shell_layout(G,nlist=shells)

nx.draw_networkx_nodes(G, pos, nodelist=nodos1,node_size=400,
                       node_color='r', node_shape='o')
nx.draw_networkx_nodes(G, pos, nodelist=nodos2,node_size=400,
                       node_color='w', node_shape='o',
                       edgecolors="k")

nx.draw_networkx_edges(G, pos)
nx.draw_networkx_labels(G, pos)
plt.axis('off')

plt.savefig('TTercero.eps', format='eps', dpi=1000)
```

Grafos_layout.py

3. Tarea tres original

En este documento se me penalizó las faltas de ortografía, asi como escribir números aislados de manera numérica en lugar de palabras.

Introducción

La importancia de los grafos es que con ellos se puede modelar ciertas procesos; encontrar los caminos mas cortos entre un nodo y otro pueden representar las conexiones sociales que existen entre una persona y otra en el caso que el grafo sea una red social. Para este y otros métodos más, la librería de `Networkx` incluye alguna variedad de algoritmos.

En este documento se implementan 5 algoritmos de `Networkx` y se mide su tiempo de ejecución.

3.1. Especificaciones técnicas

La computadora en la que se corrió los algoritmos es una Macbook Pro(13-inch,2017,Two Thunderbolt 3 ports) cuyo procesador es un Intel Core i5 de 2.3 GHz. Tiene una memoria RAM de 8 GB 2133 MHz LPDDR3

3.2. Algoritmos

Los algoritmos implementados fueron los siguientes.

`dfs_tree()`

Este algoritmo recibe como entrada un un grafo no orientado y te devuelve un árbol orientado hacia el retorno construido a partir de una fuente de búsqueda en profundidad.

`maximum_flow()`

Lo que hace este algoritmo es recibir un grafo dirigido o no dirigido que tenga capacidades, para poder devolver el valor del flujo máximo que va del nodo fuente al nodo sumidero y también devuelve un diccionario en el cual se especifica todos los arcos y el valor del flujo que corre por cada arco.

`all_shortest_pats()`

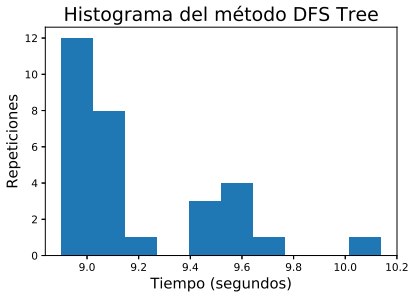
Este algoritmo proporciona el camino más corto de un grafo dirigido o no dirigido, entre un nodo inicial y un nodo final; devuelve todos los caminos con la misma cardinalidad que el del camino más corto.

`strongly_connected_components()`

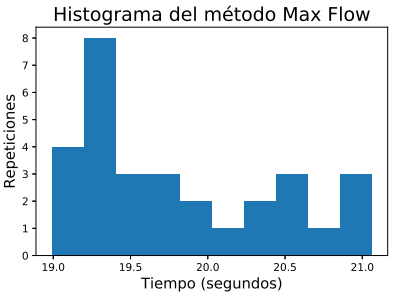
En los grafos dirigidos, los componentes fuertemente conectados son subconjuntos de nodos en los cuales ir de un nodo a otro, es posible a través de cualquier otro nodo del subconjunto. Lo que hace este algoritmo es tomar grafos dirigidos y generar una lista que tiene todos los subconjuntos de componentes fuertemente conectados ordenados de la cardinalidad mas grande a la mas chica.

`treewidth_min_degree()`

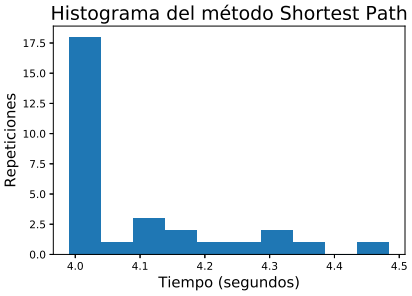
Devuelve una descomposición del ancho de árbol usando la heurística de Grado Mínimo.



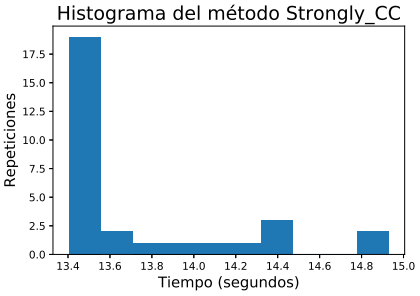
(a) Búsqueda a profundidad



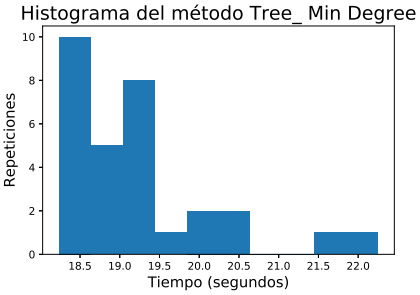
(b) Flujo máximo



(c) Caminos mas cortos



(d) Componentes fuertemente conectados



(e) Descomposición del ancho de árbol

Figura 37: Histogramas de tiempos promedios por algoritmo

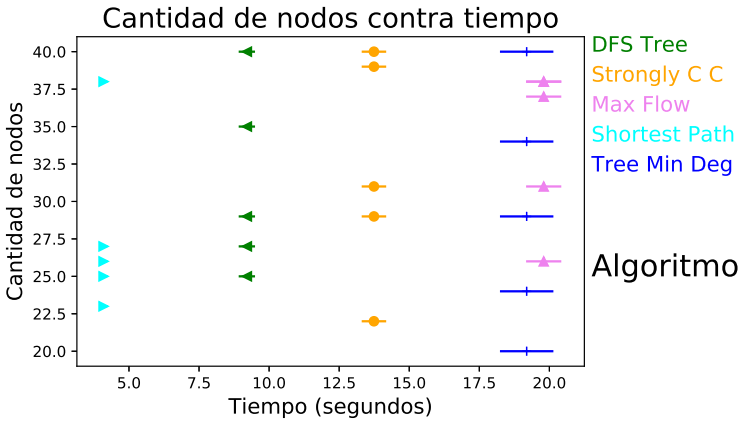


Figura 38: Error bar del tiempo promedio de ejecución vs la cantidad de nodos.

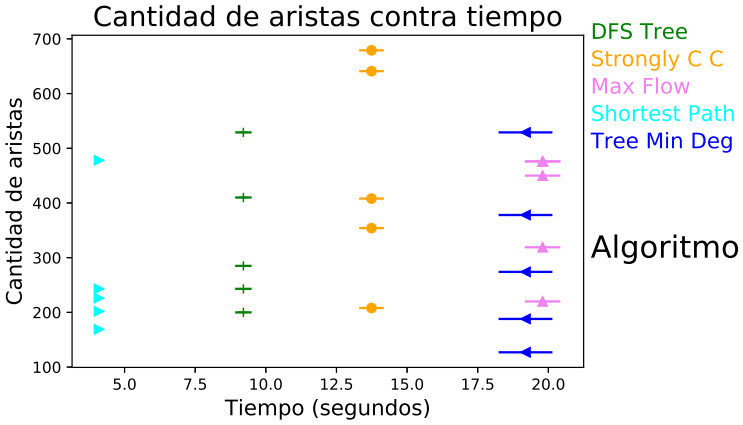


Figura 39: Error bar del tiempo promedio de ejecución vs la cantidad de aristas.

3.3. Metodología

Lo que se hizo para cada algoritmo es correr 5 grafos, cada uno con diferente cantidad de nodos y diferente cantidad de aristas, durante un tiempo de 5 segundos, a estas réplicas se les obtuvo la media, y lo mismo se hizo durante 30 veces, obteniendo así 30 datos de la media.

Con esa información se obtuvo los 5 histogramas de la figura 40

Una vez guardada la información en una variable tipo diccionario, el código que da paso a los histogramas es el siguiente

```
for i in ('DFS Tree','Max Flow','Shortest Path','Strongly_CC','Tree_ Min Degree'):
    plt.xlabel('Tiempo (segundos)', size=14)
    plt.ylabel('Repeticiones', size=14)
    plt.title('Histograma del m todo '+i,size=18)
    plt.hist(dicrio[i]['medias_repeticiones'])
    plt.savefig(i+'.eps', format='eps', dpi=1000)
    plt.show()
```

Algoritmos_grafos.py

de donde se puede apreciar que no se aproximan a una gráfica normal.

Con las medias por algoritmo se graficó el tiempo promedio que se tardó en resolver las cantidades de nodos que se expresan como el eje y, como se puede observar en la figura 47.

Al igual que se graficó el tiempo promedio que se tardó en resolver las cantidades de aristas que se expresan como el eje y, tal como se ilustra en la figura 42.

3.4. Conclusiones

Lo que se puede concluir es que el algoritmo mas rápido es el que encuentra los caminos mas cortos y el algoritmo mas complejo es el de flujo máximo.

Para graficar la error bar de la figura 42 se hace con el siguiente código.

```
y=[]
for i in ('DFS Tree','Max Flow','Shortest Path','Strongly_CC','Tree_ Min Degree'):
    for j in (1,2,3,4,5):
        y.append(dicrio[i][j]['aristas'])
for i in (1,2,3,4,5):
    if i==1:
        x=[dicrio['DFS Tree']['media_algoritmo']]
        xe=[dicrio['DFS Tree']['std_algoritmo']]
        plt.errorbar(5*x,y[0:5], xerr=5*xe, fmt='+',color='g',alpha=0.5)
    if i==2:
        x=[dicrio['Max Flow']['media_algoritmo']]
        xe=[dicrio['Max Flow']['std_algoritmo']]
        plt.errorbar(5*x,y[5:10], xerr=5*xe, fmt='^',color='violet',alpha=0.5)
    if i==3:
        x=[dicrio['Shortest Path']['media_algoritmo']]
        xe=[dicrio['Shortest Path']['std_algoritmo']]
        plt.errorbar(5*x, y[10:15],xerr=5*xe, fmt='>',color='aqua',alpha=0.5)
    if i==4:
        x=[dicrio['Strongly_CC']['media_algoritmo']]
        xe=[dicrio['Strongly_CC']['std_algoritmo']]
        plt.errorbar(5*x,y[15:20], xerr=5*xe, fmt='o',color='orange',alpha=0.5)
    if i==5:
        x=[dicrio['Tree_ Min Degree']['media_algoritmo']]
        xe=[dicrio['Tree_ Min Degree']['std_algoritmo']]
        plt.errorbar(5*x,y[20:25], xerr=5*xe, fmt='<',color='blue',alpha=0.5)
plt.xlabel('Tiempo (segundos)', size=14)
plt.ylabel('Cantidad de aristas', size=14)
plt.title('Cantidad de aristas contra tiempo',size=18)
plt.text(21.5, 700, r'DFS Tree',color='g',size=14)
plt.text(21.5, 650, r'Strongly C C',color='orange',size=14)
plt.text(21.5, 600, r'Max Flow',color='violet',size=14)
plt.text(21.5, 550, r'Shortest Path',color='aqua',size=14)
plt.text(21.5, 500, r'Tree Min Deg',color='blue',size=14)
plt.text(21.5, 300, r'Algoritmo',color='black',size=20)
plt.savefig('scater2.eps', format='eps', bbox_inches='tight', dpi=1000)
```

Algoritmos_grafos.py

3.5. Tarea tres corregida

Introducción

La importancia de los grafos es que con ellos se puede modelar ciertas procesos; encontrar los caminos más cortos entre un nodo y otro pueden representar las conexiones sociales que existen entre una persona y otra en el caso que el grafo sea una red social. Para este y otros métodos más, la librería de `Networkx` incluye alguna variedad de algoritmos.

En este documento se implementan cinco algoritmos de `Networkx` y se mide su tiempo de ejecución.

3.6. Especificaciones técnicas

La computadora en la que se corrió los algoritmos es una Macbook Pro cuyo procesador es un Intel Core i5 de 2.3 GHz. Tiene una memoria RAM de 8 GB 2133 MHz LPDDR3

3.7. Algoritmos

Los algoritmos a implementar son los siguientes.

`dfs_tree()`

Este algoritmo recibe como entrada un un grafo no orientado y devuelve un árbol orientado hacia el retorno construido a partir de una fuente de búsqueda en profundidad.

`maximum_flow()`

Lo que hace este algoritmo es recibir un grafo dirigido o no dirigido que tenga capacidades, para poder devolver el valor del flujo máximo que va del nodo fuente al nodo sumidero y también devuelve un diccionario en el cual se especifica todos los arcos y el valor del flujo que corre por cada arco.

`all_shortest_pats()`

Este algoritmo proporciona el camino más corto de un grafo dirigido o no dirigido, entre un nodo inicial y un nodo final; devuelve todos los caminos con la misma cardinalidad que el del camino más corto.

`strongly_connected_components()`

En los grafos dirigidos, los componentes fuertemente conectados son subconjuntos de nodos en los cuales ir de un nodo a otro, es posible a través de cualquier otro nodo del subconjunto. Lo que hace este algoritmo es tomar grafos dirigidos y generar una lista que tiene todos los subconjuntos de componentes fuertemente conectados ordenados de la cardinalidad más grande a la más chica.

`treewidth_min_degree()`

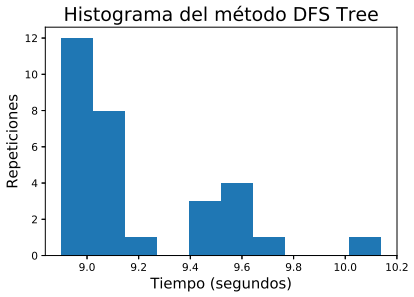
Devuelve una descomposición del ancho de árbol usando la heurística de Grado Mínimo.

3.8. Metodología

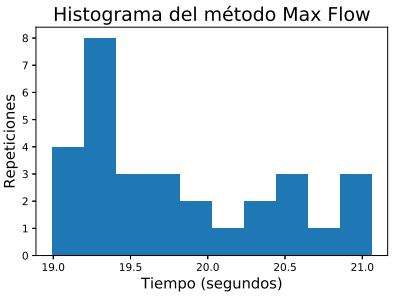
Lo que se hizo para cada algoritmo es correr cinco grafos, cada uno con diferente cantidad de nodos y diferente cantidad de aristas, durante un tiempo de cinco segundos, a estas réplicas se les obtuvo la media, y lo mismo se hizo durante treinta veces, obteniendo así treinta datos de la media.

Con esa información se obtuvo los 5 histogramas de la figura 40.

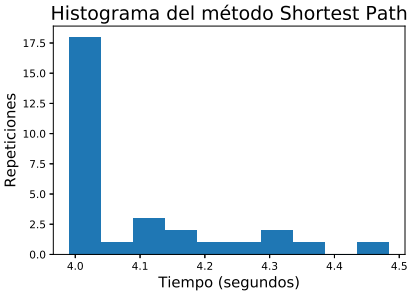
Una vez guardada la información en una variable tipo diccionario, el código que da paso a los histogramas es el siguiente de donde se puede apreciar que no se aproximan a una gráfica normal.



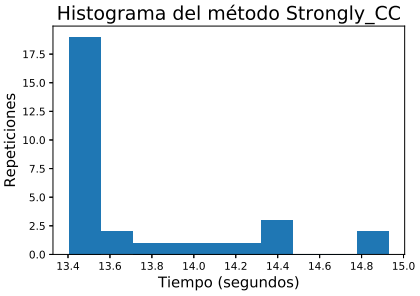
(a) Búsqueda a profundidad



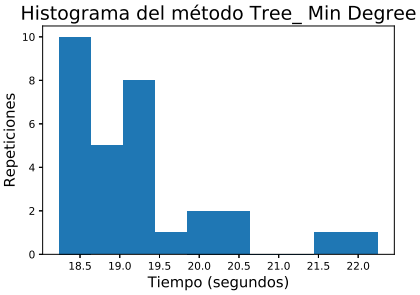
(b) Flujo máximo



(c) Caminos mas cortos



(d) Componentes fuertemente conectados



(e) Descomposición del ancho de árbol

Figura 40: Histogramas de tiempos promedios por algoritmo

```
for i in ('DFS Tree','Max Flow','Shortest Path','Strongly_CC','Tree_ Min Degree'):
    plt.xlabel('Tiempo (segundos)', size=14)
    plt.ylabel('Repeticiones', size=14)
    plt.title('Histograma del m todo '+i,size=18)
    plt.hist(dicrio[i]['medias_repeticiones'])
    plt.savefig(i+'.eps', format='eps',  dpi=1000)
    plt.show()
```

Algoritmos_grafos.py

Con las medias por algoritmo se graficó el tiempo promedio que se tardó en resolver las cantidades de nodos que se expresan como el eje y, como se puede observar en la figura 47.

Al igual que se graficó el tiempo promedio que se tardó en resolver las cantidades de aristas que se expresan como el eje y, tal como se ilustra en la figura 42.

3.9. Conclusiones

Lo que se puede concluir es que el algoritmo más rápido es el que encuentra los caminos más cortos y el algoritmo más complejo es el de flujo máximo.

Para graficar la error bar de la figura 42 se hace con el siguiente código.

```
y=[]
for i in ('DFS Tree','Max Flow','Shortest Path','Strongly_CC','Tree_ Min Degree'):
    for j in (1,2,3,4,5):
        y.append(dicrio[i][j]['aristas'])
for i in (1,2,3,4,5):
    if i==1:
        x=[dicrio['DFS Tree']['media_algoritmo']]
        xe=[dicrio['DFS Tree']['std_algoritmo']]
        plt.errorbar(5*x,y[0:5], xerr=5*xe, fmt='+',color='g',alpha=0.5)
    if i==2:
        x=[dicrio['Max Flow']['media_algoritmo']]
        xe=[dicrio['Max Flow']['std_algoritmo']]
        plt.errorbar(5*x,y[5:10], xerr=5*xe, fmt='^',color='violet',alpha=0.5)
    if i==3:
        x=[dicrio['Shortest Path']['media_algoritmo']]
        xe=[dicrio['Shortest Path']['std_algoritmo']]
        plt.errorbar(5*x, y[10:15],xerr=5*xe, fmt='>',color='aqua',alpha=0.5)
    if i==4:
        x=[dicrio['Strongly_CC']['media_algoritmo']]
        xe=[dicrio['Strongly_CC']['std_algoritmo']]
        plt.errorbar(5*x,y[15:20], xerr=5*xe, fmt='o',color='orange',alpha=0.5)
    if i==5:
        x=[dicrio['Tree_ Min Degree']['media_algoritmo']]
        xe=[dicrio['Tree_ Min Degree']['std_algoritmo']]
        plt.errorbar(5*x,y[20:25], xerr=5*xe, fmt='<',color='blue',alpha=0.5)
plt.xlabel('Tiempo (segundos)', size=14)
plt.ylabel('Cantidad de aristas', size=14)
plt.title('Cantidad de aristas contra tiempo',size=18)
plt.text(21.5, 700, r'DFS Tree',color='g',size=14)
plt.text(21.5, 650, r'Strongly C C',color='orange',size=14)
plt.text(21.5, 600, r'Max Flow',color='violet',size=14)
plt.text(21.5, 550, r'Shortest Path',color='aqua',size=14)
plt.text(21.5, 500, r'Tree Min Deg',color='blue',size=14)
plt.text(21.5, 300, r'Algoritmo',color='black',size=20)
plt.savefig('scater2.eps', format='eps', bbox_inches='tight', dpi=1000)
```

Algoritmos_grafos.py

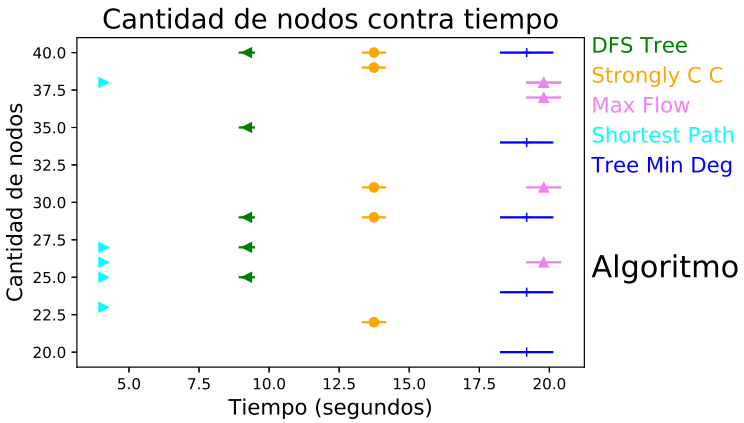


Figura 41: Barras de error del tiempo promedio de ejecución vs la cantidad de nodos.

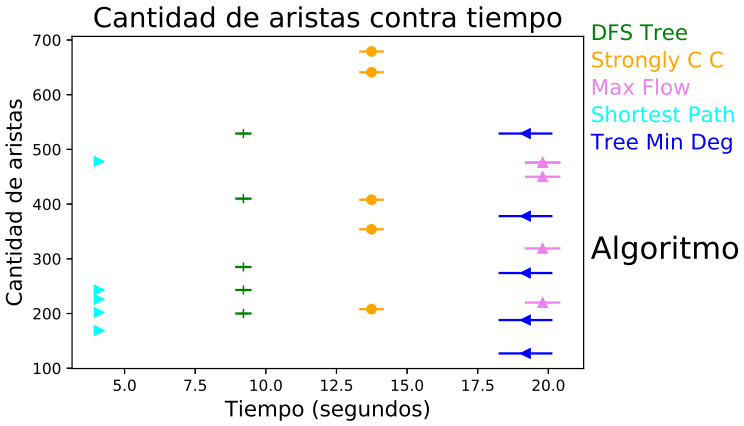


Figura 42: Barras de error del tiempo promedio de ejecución vs la cantidad de aristas.

4. Tarea cuatro original

En este documento se me penalizó por los acentos, aparte de que en el código no se entendieron las variables por el mal acomodo; las gráficas tenían titulo y el ANOVA también presentó algunos detalles.

Introducción

Distintos algoritmos se pueden usar para resolver los mismos problemas; el desempeño de los algoritmos es la manera de evaluar a los mismos en base a su eficiencia en la resolución de dichos problemas. Existen distintos métodos para evaluar del desempeño, esto se debe al tipo de algoritmos y sus problemas. En este documento se verá que afecta directamente el desempeño de ciertos algoritmos de `Networkx`.

4.1. Especificaciones técnicas

La computadora en la que se corrió los algoritmos es una Macbook Pro, cuyo procesador es un Intel Core i5 de 2.3 GHz. Tiene una memoria RAM de 8 GB 2133 MHz.

4.2. Algoritmos

Los algoritmos a implementar son los siguientes.

`maximum_flow_value()`

Este algoritmo recibe como entrada un un grafo no orientado con capacidad en sus aristas junto con un par de nodos, el nodo *fuente* que es de donde parte el flujo y el nodo *sumidero* que es a donde llega el flujo. Devuelve el valor del flujo máximo soportado entre los nodos fuente y sumidero.

Utiliza el método de la etiqueta mas alta, es decir, de tomar el camino por el cual hay mas capacidad de aumento de flujo.

`flow.edmonds_karp()`

Este algoritmo recibe como entrada un grafo con capacidad junto con los nodos fuente y sumidero; lo que devuelve es la *red residual* después de haber calculado el flujo máximo.

Tiene rasgos parecidos al algoritmo de Ford-Fulkerson pero la diferencia más significativa es que el algoritmo de Edmonds Karp busca el camino mas corto para encontrar una ruta de aumento, es decir, explora todos los caminos posibles entre la fuente y el sumidero buscando cuales tienen capacidad disponible.

`flow.boykov_kolmogorov()`

Este algoritmo devuelve la red residual resultante después de calcular el flujo máximo, utilizando el algoritmo Boykov-Kolmogorov.

4.3. Generadores

Los generadores de grafos implementados fueron los siguientes.

`dense_gnm_random_graph()`

Este generador de grafos toma como parámetros la cantidad de nodos y aristas que se desean obtener; y devuele un grafo de entre los posibles grafos que cumplen con esos parámetros.

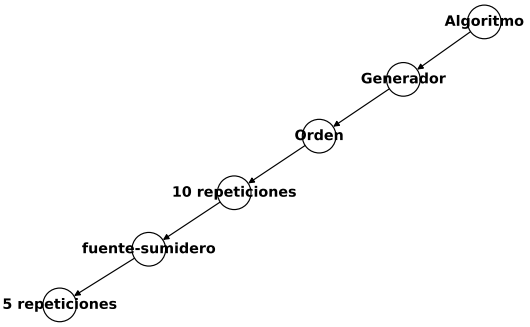


Figura 43: Diagrama de flujo aplicada en la metodología.

`random_graphs.barabasi_albert_graph()`

Este comando genera grafos aleatorios agregando las aristas de manera preferencial. Al generador se le dan los parámetros n y m que representan los nodos totales y el número de aristas a unir desde un nuevo nodo a los nodos existentes de acuerdo al algoritmo de Barabási-Albert.

`duplication.duplication_divergence_graph()`

A este generador se le dan los parámetros n y p que representan el total de nodos y la probabilidad de retención de aristas incidentes a los nodos originales.

4.4. Metodología

El interés de este estudio es ver que variables influyen en el tiempo del algoritmo y de que manera influyen estas variables.

Cada algoritmo usó los tres generadores; cada generador hizo diez grafos con cuatro cantidades de nodos diferentes, cada grafo de orden logarítmico base cuatro; y se corrió el algoritmo cinco veces por cada par de nodo fuente-sumidero.

Como se muestra en la figura 51 son tres algoritmos, con tres generadores, cuatro ordenes y cinco diferentes pares de nodos fuente y sumidero.

La implementación en python se lleva a cabo con el código siguiente.

```
def correr(generator, algorithm, crdl, l):
    if generator==0:
        G=AddEdges(nx.dense_gnm_random_graph(crdl,
                   int((0.5*sz*(sz-1))-(1-((dnsd+1)/10))*(0.5*sz*(sz-1)))))
        nds=RandNodes(crdl-1,G)
        data[contador,3]=nx.density(G)
        if algorithm==0:
            nx.maximum_flow_value(G,nds[l][0],nds[l][1])
        elif algorithm==1:
            nx.algorithms.flow.edmonds_karp(G,nds[l][0],nds[l][1])
        elif algorithm==2:
            nx.algorithms.flow.boykov_kolmogorov(G,nds[l][0],nds[l][1])
    elif generator==1:
        F=AddEdges(nx.generators.random_graphs.barabasi_albert_graph(crdl,
                   random.randint(1,crdl-1)))
        nds=RandNodes(crdl-1,F)
        data[contador,3]=nx.density(F)
        if algorithm==0:
            nx.maximum_flow_value(F,nds[l][0],nds[l][1])
        elif algorithm==1:
            nx.algorithms.flow.edmonds_karp(F,nds[l][0],nds[l][1])
        elif algorithm==2:
            nx.algorithms.flow.boykov_kolmogorov(F,nds[l][0],nds[l][1])
    elif generator==2:
        J=AddEdges(nx.generators.duplication.duplication_divergence_graph(crdl,
                   random.random(), seed=None))
        nds=RandNodes(crdl-1,J)
        data[contador,3]=nx.density(J)
        if algorithm==0:
            nx.maximum_flow_value(J,nds[l][0],nds[l][1])
        elif algorithm==1:
            nx.algorithms.flow.edmonds_karp(J,nds[l][0],nds[l][1])
        elif algorithm==2:
            nx.algorithms.flow.boykov_kolmogorov(J,nds[l][0],nds[l][1])

data=np.arange(1800*5, dtype=float).reshape(1800,5)
contador=0
for generador in range(3):
    for algoritmo in range(3):
        for orden in (16,64,256,1024):
            sz=orden
            for dnsd in range(10):
                for repeticion in range(5):
                    tiempo_inicial=time()
                    correr(generador, algoritmo, sz, repeticion)
                    tiempo_final=time()
                    tiempo_ejecucion=tiempo_final-tiempo_inicial
                    data[contador,0]=generador
                    data[contador,1]=algoritmo
                    data[contador,2]=orden
                    data[contador,4]=tiempo_ejecucion
                    contador+=1
```

Complejidad_experimental.py

4.5. Resultados

Con esa información se grafica la figura 47 cuya variable independiente son las observaciones, la variable dependiente es el tiempo que hizo en esa observación y cada punto representa una combinación de algoritmo y generador.

Se hizo una prueba de correlación entre las variables, para ver si las variables se relacionan entre ellas y con el tiempo de ejecución. Lo que muestra la figura 81 es que el orden y la densidad se relaciona directamente proporcional con el tiempo de ejecución.

Luego, se hizo un ANOVA para identificar si las medias de los conjuntos eran iguales o hay diferencias significativas en ellas. Se obtuvo los resultados que se muestra en la tabla llamada

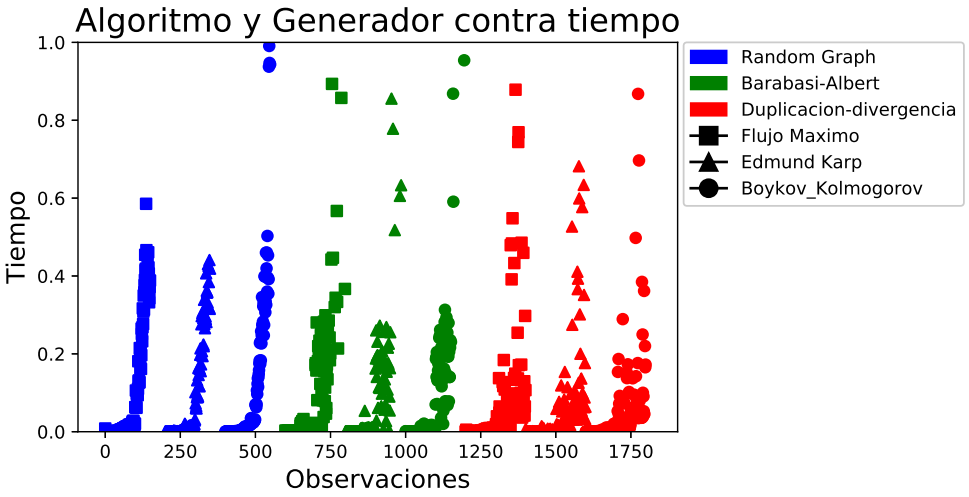


Figura 44: Los colores representan a los generadores y las figuras representan algoritmos.

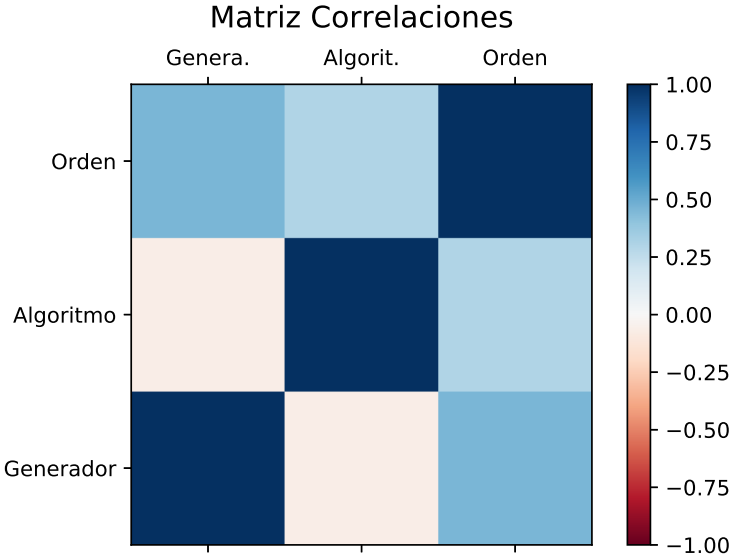


Figura 45: Correlaciones entre los datos cuantitativos.

`ANOVA.txt` en el cual los valores de la tercer columna que son mayores a 0.1, sus medias son iguales y los valores menores 0.1 se rechaza la hipótesis de que las medias sean iguales.

ANOVA.txt

```
                                    sum_sq  ...         PR>F
Generador                        19.896559  ...   4.091688e-02
Algoritmo                        30.651822  ...   1.119168e-02
Orden                          3983.251412  ...  2.274746e-151
Densidad                        935.908570  ...   1.666072e-42
Generador:Algoritmo              51.350711  ...   1.033386e-03
Algoritmo:Orden                  95.235651  ...   8.087981e-06
Orden:Densidad                 2475.624075  ...  2.379699e-101
Generador:Orden                  30.155743  ...   1.186676e-02
Generador:Densidad               25.595379  ...   2.043037e-02
Algoritmo:Densidad              216.461211  ...   2.019525e-11
Generador:Algoritmo:Orden       131.772632  ...   1.571421e-07
Generador:Algoritmo:Densidad     39.192928  ...   4.135139e-03
Generador:Densidad:Orden         90.019381  ...   1.427670e-05
Algoritmo:Densidad:Orden        615.849394  ...   5.013261e-29
Generador:Algoritmo:Densidad:Orden  115.887933  ...   8.652402e-07
Residual                       8480.522655  ...            NaN

[16 rows x 4 columns]
```

4.6. Tarea cuatro corregida

Introducción

Distintos algoritmos se pueden usar para resolver los mismos problemas; el desempeño de los algoritmos es la manera de evaluar a los mismos en base a su eficiencia en la resolución de dichos problemas. Existen distintos métodos para evaluar del desempeño, esto se debe al tipo de algoritmos y sus problemas. En este documento se verá que afecta directamente el desempeño de ciertos algoritmos de `Networkx`.

4.7. Especificaciones técnicas

La computadora en la que se corrió los algoritmos es una Macbook Pro, cuyo procesador es un Intel Core i5 de 2.3 GHz. Tiene una memoria RAM de 8 GB 2133 MHz.

4.8. Algoritmos

Los algoritmos a implementar son los siguientes.

`maximum_flow_value()`

Este algoritmo recibe como entrada un un grafo no orientado con capacidad en sus aristas junto con un par de nodos, el nodo *fuente* que es de donde parte el flujo y el nodo *sumidero* que es a donde llega el flujo. Devuelve el valor del flujo máximo soportado entre los nodos fuente y sumidero.

Utiliza el método de la etiqueta mas alta, es decir, de tomar el camino por el cual hay mas capacidad de aumento de flujo.

`flow.edmonds_karp()`

Este algoritmo recibe como entrada un grafo con capacidad junto con los nodos fuente y sumidero; lo que devuelve es la *red residual* después de haber calculado el flujo máximo.

Tiene rasgos parecidos al algoritmo de Ford-Fulkerson pero la diferencia más significativa es que el algoritmo de Edmonds Karp busca el camino mas corto para encontrar una ruta de aumento, es decir, explora todos los caminos posibles entre la fuente y el sumidero buscando cuales tienen capacidad disponible.

`flow.boykov_kolmogorov()`

Este algoritmo devuelve la red residual resultante después de calcular el flujo máximo, utilizando el algoritmo Boykov-Kolmogorov.

4.9. Generadores

Los generadores de grafos implementados fueron los siguientes.

`dense_gnm_random_graph()`

Este generador de grafos toma como parámetros la cantidad de nodos y aristas que se desean obtener; y devuele un grafo de entre los posibles grafos que cumplen con esos parámetros.

`random_graphs.barabasi_albert_graph()`

Este comando genera grafos aleatorios agregando las aristas de manera preferencial. Al generador se le dan los parámetros n y m que representan los nodos totales y el número de aristas a unir desde un nuevo nodo a los nodos existentes de acuerdo al algoritmo de Barabási-Albert.

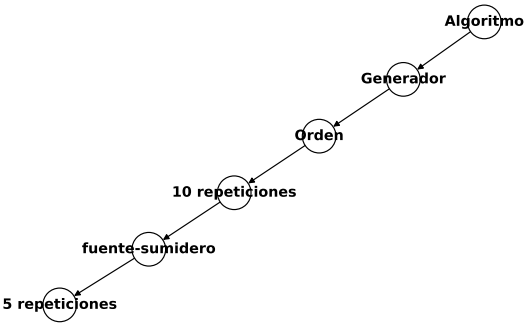


Figura 46: Diagrama de flujo aplicada en la metodología.

`duplication.duplication_divergence_graph()`

A este generador se le dan los parámetros n y p que representan el total de nodos y la probabilidad de retención de aristas incidentes a los nodos originales.

4.10. Metodología

El interés de este estudio es ver que variables influyen en el tiempo del algoritmo y de que manera influyen estas variables.

Cada algoritmo usó los tres generadores; cada generador hizo diez grafos con cuatro cantidades de nodos diferentes, cada grafo de orden logarítmico base cuatro; y se corrió el algoritmo cinco veces por cada par de nodo fuente-sumidero.

Como se muestra en la figura 51 son tres algoritmos, con tres generadores, cuatro ordenes y cinco diferentes pares de nodos fuente y sumidero.

La implementación en python se lleva a cabo con el código siguiente.

```
def correr(generator, algorithm, crdl, l):
    if generator==0:
        G=AddEdges(nx.dense_gnm_random_graph(crdl,
                    int((0.5*sz*(sz-1))-(1-((dnsd+1)/10))*(0.5*sz*(sz-1)))))
        nds=RandNodes(crdl-1,G)
        data[contador,3]=nx.density(G)
        if algorithm==0:
            nx.maximum_flow_value(G,nds[l][0],nds[l][1])
        elif algorithm==1:
            nx.algorithms.flow.edmonds_karp(G,nds[l][0],nds[l][1])
        elif algorithm==2:
            nx.algorithms.flow.boykov_kolmogorov(G,nds[l][0],nds[l][1])
    elif generator==1:
        F=AddEdges(nx.generators.random_graphs.barabasi_albert_graph(crdl,
                    random.randint(1,crdl-1)))
        nds=RandNodes(crdl-1,F)
        data[contador,3]=nx.density(F)
        if algorithm==0:
            nx.maximum_flow_value(F,nds[l][0],nds[l][1])
        elif algorithm==1:
            nx.algorithms.flow.edmonds_karp(F,nds[l][0],nds[l][1])
        elif algorithm==2:
            nx.algorithms.flow.boykov_kolmogorov(F,nds[l][0],nds[l][1])
    elif generator==2:
        J=AddEdges(nx.generators.duplication.duplication_divergence_graph(crdl,
                    random.random(), seed=None))
        nds=RandNodes(crdl-1,J)
        data[contador,3]=nx.density(J)
        if algorithm==0:
            nx.maximum_flow_value(J,nds[l][0],nds[l][1])
        elif algorithm==1:
            nx.algorithms.flow.edmonds_karp(J,nds[l][0],nds[l][1])
        elif algorithm==2:
            nx.algorithms.flow.boykov_kolmogorov(J,nds[l][0],nds[l][1])

data=np.arange(1800*5, dtype=float).reshape(1800,5)
contador=0
for generador in range(3):
    for algoritmo in range(3):
        for orden in (16,64,256,1024):
            sz=orden
            for dnsd in range(10):
                for repeticion in range(5):
                    tiempo_inicial=time()
                    correr(generador, algoritmo, sz, repeticion)
                    tiempo_final=time()
                    tiempo_ejecucion=tiempo_final-tiempo_inicial
                    data[contador,0]=generador
                    data[contador,1]=algoritmo
                    data[contador,2]=orden
                    data[contador,4]=tiempo_ejecucion
                    contador+=1
```

Complejidad_experimental.py

4.11. Resultados

Con esa información se grafica la figura 47 cuya variable independiente son las observaciones, la variable dependiente es el tiempo que hizo en esa observación y cada punto representa una combinación de algoritmo y generador.

Se hizo una prueba de correlación entre las variables, para ver si las variables se relacionan entre ellas y con el tiempo de ejecución. Lo que muestra la figura 81 es que el orden y la densidad se relaciona directamente proporcional con el tiempo de ejecución.

Luego, se hizo un ANOVA para identificar si las medias de los conjuntos eran iguales o hay diferencias significativas en ellas. Se obtuvo los resultados que se muestra en la tabla llamada

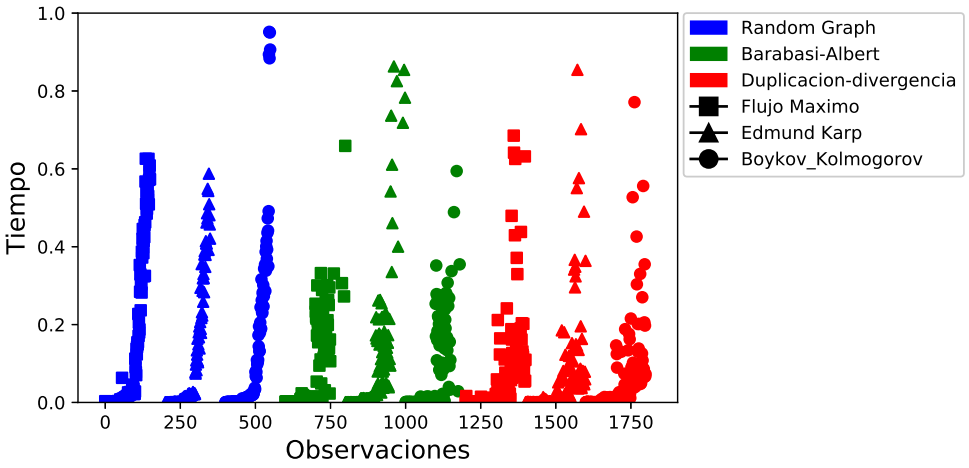


Figura 47: Los colores representan a los generadores y las figuras representan algoritmos.

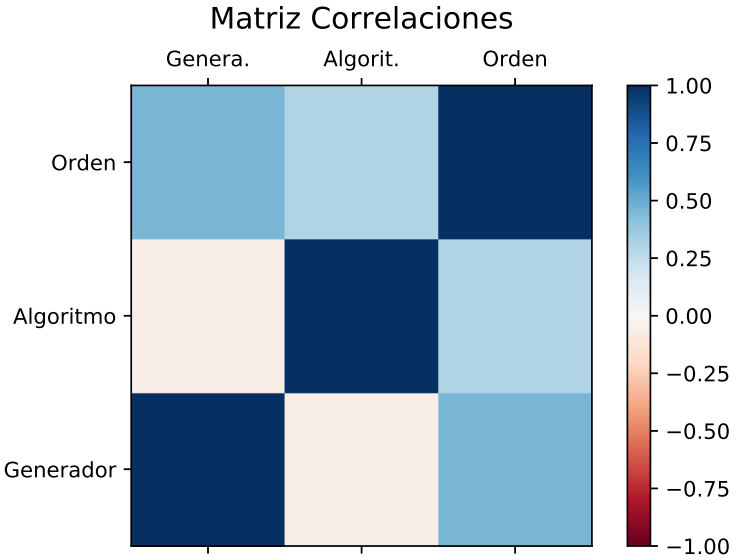


Figura 48: Correlaciones entre los datos cuantitativos.

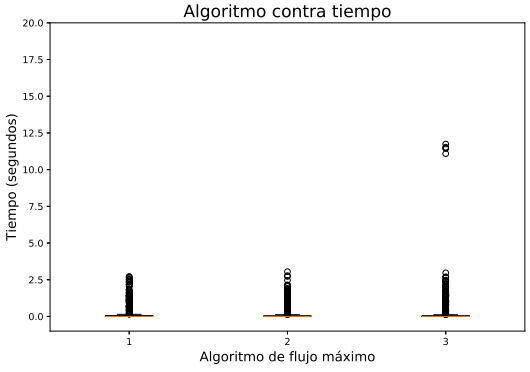


Figura 49: Diagrama de caja de algoritmo.

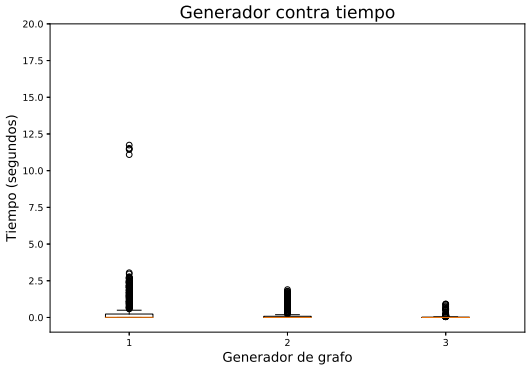


Figura 50: Diagrama de caja del generador.

Figura 51: Diagrama de caja del orden.

`ANOVA.txt` en el cual los valores de la tercer columna que son mayores a 0.1, sus medias son iguales y los valores menores 0.1 se rechaza la hipótesis de que las medias sean iguales.

ANOVA.txt

	PR>F	
Generador	0.1081764	
Algoritmo	0.0000661265	
Orden		0
Densidad	0	
Generador:Algoritmo	0.003500409	
Algoritmo:Orden	0	
Orden:Densidad		0
Generador:Orden	0.09022655	
Generador:Densidad	0.0004868091	
Algoritmo:Densidad	0	
Generador:Algoritmo:Orden	0.0000142996	
Generador:Algoritmo:Densidad	0.0000049859	
Generador:Densidad:Orden	0	
Algoritmo:Densidad:Orden	0	
Generador:Algoritmo:Densidad:Orden	0	
Residual		

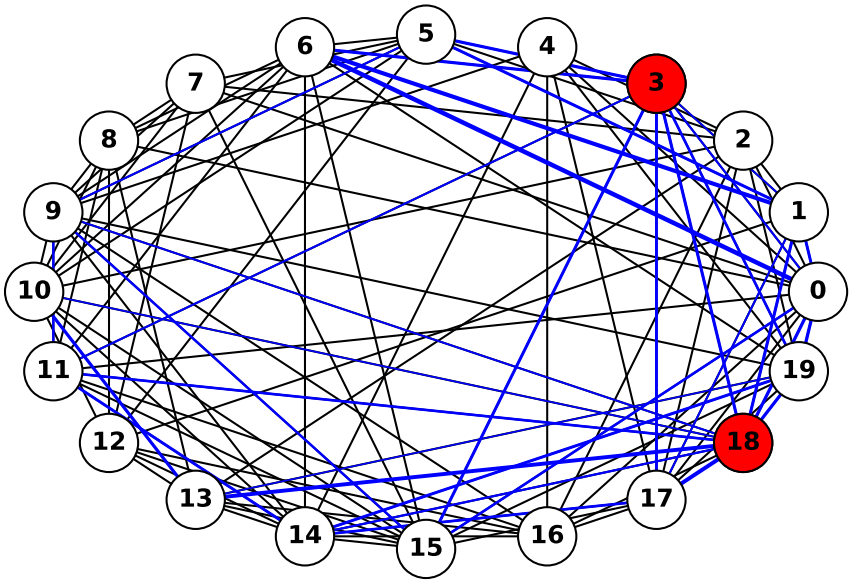


Figura 52: Grafo con rutas de flujo máimo.

5. Tarea cinco original

Introducción

En un grafo formado por aristas y nodos, donde cada arista tiene un valor que se interpreta como la capacidad, algo de interés es, dado dos nodos identificados como **fuente** y **sumidero**, es saber cual es el valor del flujo máximo que puede transportar desde el fuente al sumidero. Otros aspectos importantes es ver como las caracteristicas del nodo fuente y el nodo sumidero afectan al valor del flujo máximo en el mismo grafo.

En este documento se habla acerca de las caracteristicas que presentan los nodos en un grafo y de que manera afectan estos en el valor del flujo máximo mediante un algoritmo de `Networkx`.

5.1. Especificaciones técnicas

La computadora en la que se corrió los algoritmos es una Macbook Pro, cuyo procesador es un Intel Core i5 de 2.3 GHz. Tiene una memoria RAM de 8 GB 2133 MHz.

5.2. Generador

`watts_strogatz_graph()`

Este generador de grafos toma como parámetros la cantidad de nodos deseados, el numero de nodos vecinos con los que se puede conectar y la probabilidad de conexión desde ese nodo con los nodos que puede conectar.

Usualmente, este tipo de grafos, se usa para hacer las redes de mundo pequeño estas redes hacen referencia a que por mas alejado que esté un nodo del otro, todos los nodos son accesibles desde otro nodo.

En las figuras 67, 83, 84, 85, 86 se muestran grafos que son hechos con este generador.

5.3. Algoritmos

En esta sección se habla de los algoritmos que se usaron junto con una breve descripción.

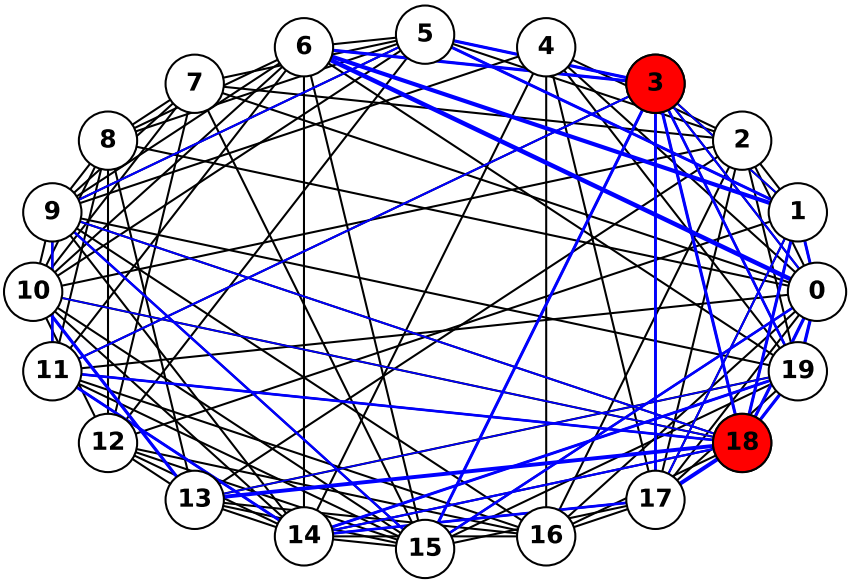


Figura 53: Grafo con rutas de flujo máimo.

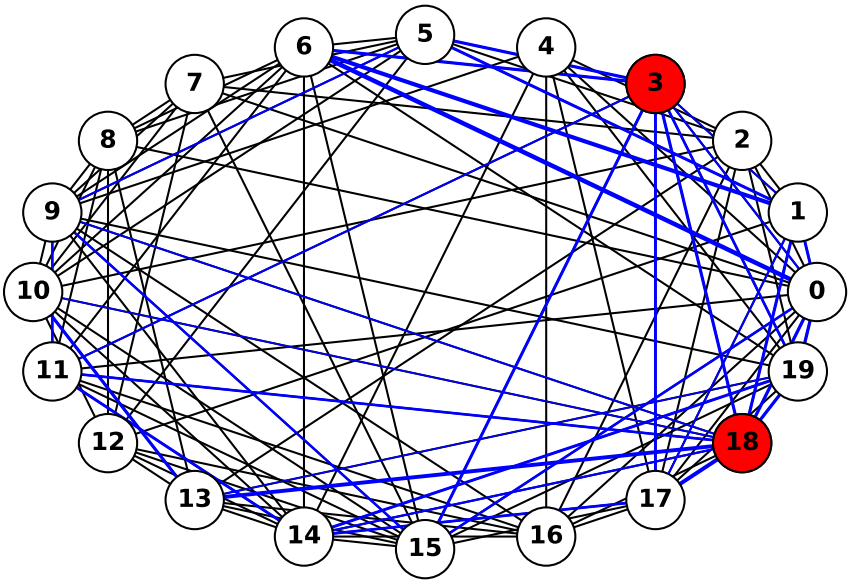


Figura 54: Grafo con rutas de flujo máimo.

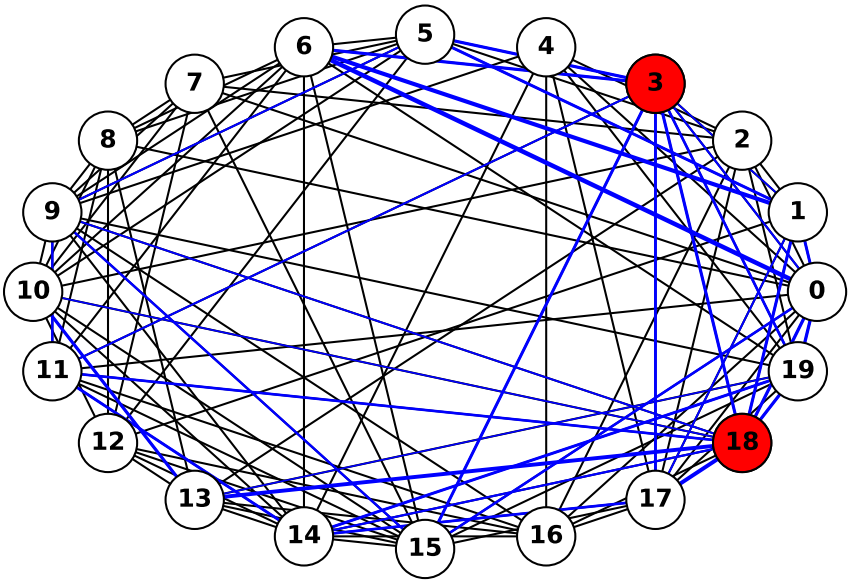


Figura 55: Grafo con rutas de flujo máimo.

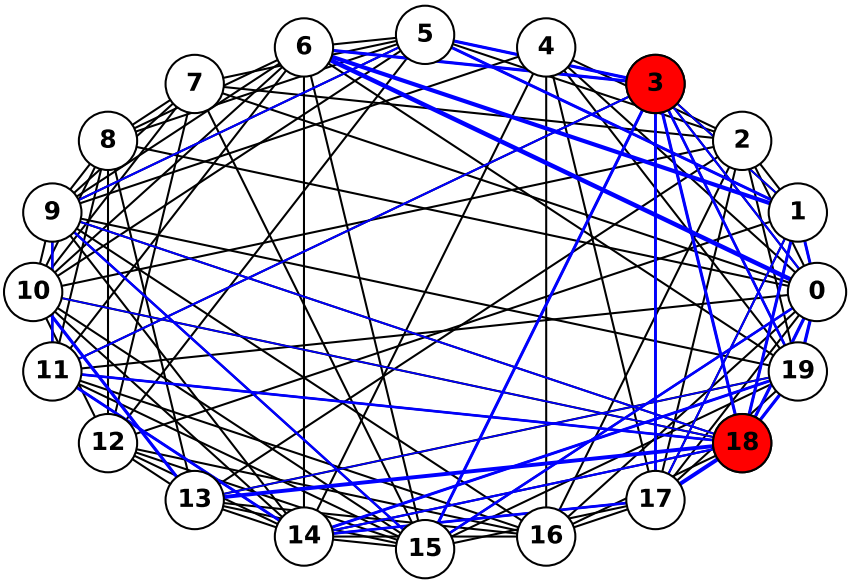


Figura 56: Grafo con rutas de flujo máimo.

Algoritmo de flujo máximo

El algoritmo implementado para encontrar el flujo máximo es `maximum_flow()` el cual toma como parámetros un grafo el cual debe tener como atributo la capacidad de cada arista, junto con el par de nodos fuente y sumidero. Al encontrar el flujo máximo devuelve el valor de este, asi como el flujo que pasa por cada arista para obtener ese valor del flujo máximo.

`degree_centrality()`

`clustering()`

Este algoritmo que recibe como parámetro un grafo que puede ser con pesos en sus aristas o sin pesos; el caso que se tiene aristas con pesos, devuelve el promedio geométrico de los pesos del subgrafo.

`closeness_centrality()`

Mediante este algoritmo se obtiene una medida de centralidad de la red, es decir, para cada nodo se calcula que tan central es; entre más al centro esté el nodo está mas cerca de todos los demás nodos.

`load_centrality()`

Una vez que se obtienen todas las rutas mas cortas, el dato que ofrece este algoritmo es la fracción de todas las rutas mas cortas que pasan a través de ese nodo.

`eccentricity()`

La información que aporta este algoritmo es para determinar que tan cerca o tan lejos está un nodo con respecto a los demás.

`pagerank()`

Este algoritmo calcula una clasificación de los nodos en el gráfico de acuerdo a las características de los nodos con quien esta conectado.

5.4. Metodología

El interés de este estudio es determinar qué características de los nodos influyen en un mayor valor de flujo máximo. Lo que se hace es generar un grafo de diferentes tamaños, desde veinte nodos hasta sesenta nodos con incrementos de diez nodos, una vez que el grafo ya tiene capacidad se eligen los pares de nodos fuente y sumidero, se calcula el valor del flujo máximo y se guarda las características que tienen éstos nodos.

La implementación en python se lleva a cabo con el código siguiente.

```
#--------------------Experimentacion-------------------------------
datos=np.arange(sum(orden)*15, dtype=float).reshape(sum(orden),15)
fila=0
for i in range(len(orden)):
    H=nx.watts_strogatz_graph(orden[i], int(orden[i]/2) , 0.33 , seed=None)
    initial=0
    final=0
    lista=[]
    lista[:]=H.edges
    width=np.arange(len(lista)*1,dtype=float).reshape(len(lista),1)
    for r in range(len(lista)):
        R=np.random.normal(loc=20, scale=5.0, size=None)
        width[r]=R
        H.add_edge(lista[r][0],lista[r][1],capacity=R)
    for w in range(orden[i]):
        initial=random.randint(0,round(len(H.nodes)/2))
        final=random.randint(initial,len(H.nodes)-2)
        while initial==final:
            initial=random.randint(0,round(len(H.nodes)/2))
            final=random.randint(initial,len(H.nodes)-2)
        tiempo_inicial=time()
        T=nx.maximum_flow(H, initial, final)
        tiempo_final=time()
        datos[fila,0]=T[0]
        datos[fila,1]=tiempo_final-tiempo_inicial
#--------------------Info Fuente-----------------------------------
        datos[fila,2]=nx.degree_centrality(H)[initial]
        datos[fila,3]=nx.clustering(H, nodes=initial)
        datos[fila,4]=nx.closeness_centrality(H, u=initial)
        datos[fila,5]=nx.load_centrality(H, v=initial)
        datos[fila,6]=nx.eccentricity(H, v=initial)
        datos[fila,7]=nx.pagerank(H, alpha=0.9, weight='weight')[initial]
#--------------------Info Sumidero---------------------------------
        datos[fila,8]=nx.degree_centrality(H)[final]
        datos[fila,9]=nx.clustering(H, nodes=final)
        datos[fila,10]=nx.closeness_centrality(H, u=final)
        datos[fila,11]=nx.load_centrality(H, v=final)
        datos[fila,12]=nx.eccentricity(H, v=final)
        datos[fila,13]=nx.pagerank(H, alpha=0.9, weight='weight')[final]
        datos[fila,14]=orden[i]
        fila+=1
```

caracterizacion.py

5.5. Resultados

Se elaboró unos diagramas de caja para ver el comportamiento de las características de los nodos fuente y sumidero 72, 73, 74, 75, 76, 77, 78, 79, 80. Los valores del eje de las abscisas representan el orden de los grafos.

Con esa información se hace una matriz de correlación 81 donde el cero es el valor del flujo máximo, el uno es el valor del tiempo, la fila cinco es el valor de load centrality del nodo fuente, el siete representa el pagerank del nodo fuente, las filas once y trece es el valor de load centrality y pagerank del nodo sumidero respectivamente.

Las filas y columnas en negro, es debido a que representan la excentricidad y su valor permanece constante para todos los grafos.

Lo que nos muestra 81 es que el valor de load centrality y el pagerank, tanto del nodo fuente como del sumidero afectan inversamente al valor del flujo máximo.

Luego, se hizo un ANOVA para identificar si las medias de los conjuntos eran iguales o hay diferencias significativas en ellas. Se obtuvo los resultados que se muestra en la tabla llamada `ANOVA.txt` en el cual los valores de la tercer columna que son mayores a 0.1, sus medias son iguales y los valores menores 0.1 se rechaza la hipótesis de que las medias sean iguales.

`ANOVA.txt`

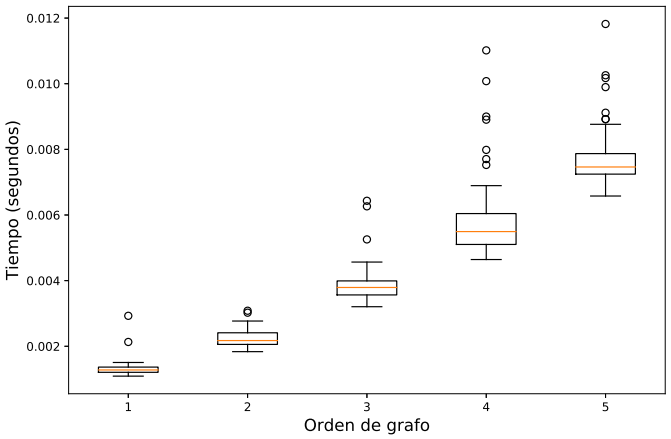


Figura 57: Diagrama de caja de tiempo.

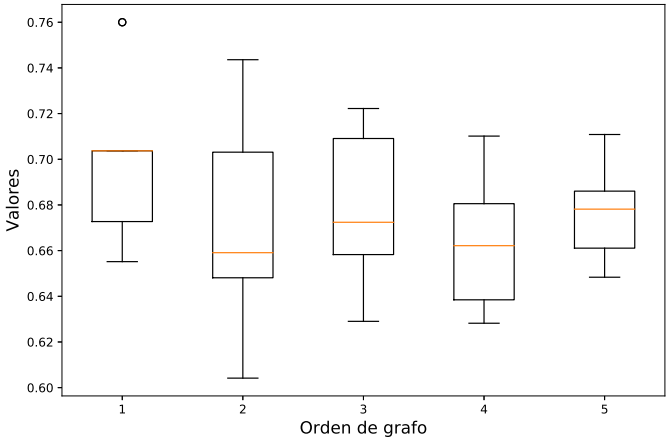


Figura 58: Diagrama de caja de centralidad de nodos fuentes.

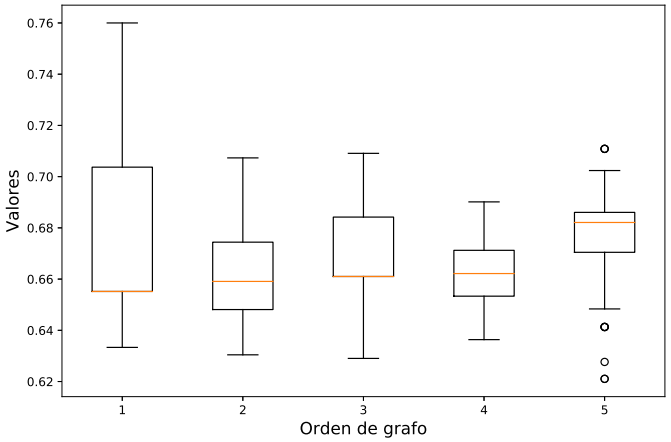


Figura 59: Diagrama de caja de centralidad de nodos sumidero.

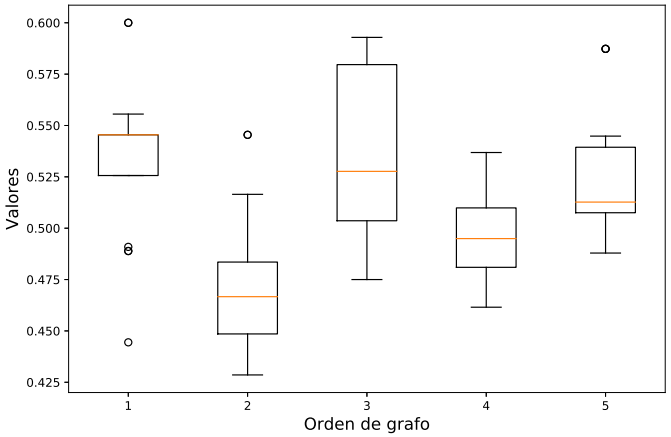


Figura 60: Diagrama de caja de clustering de nodos fuentes.

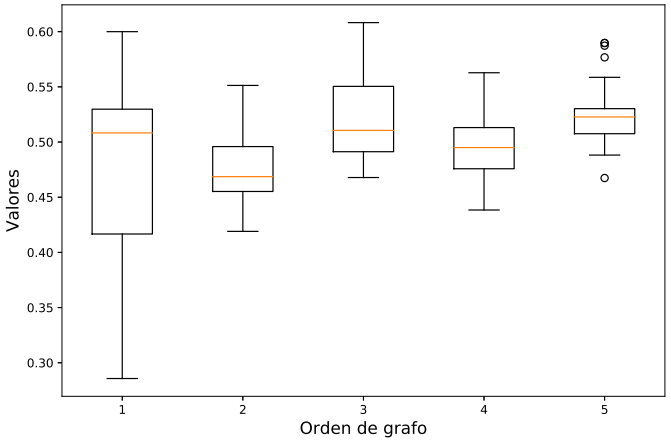


Figura 61: Diagrama de caja de clustering de nodos sumidero.

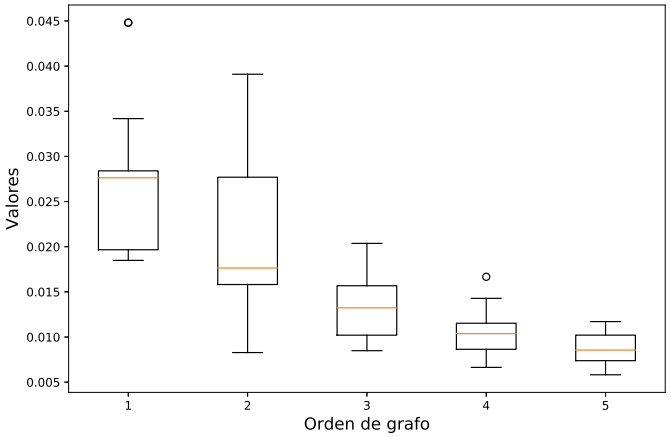


Figura 62: Diagrama de caja de load centrality de nodos fuentes.

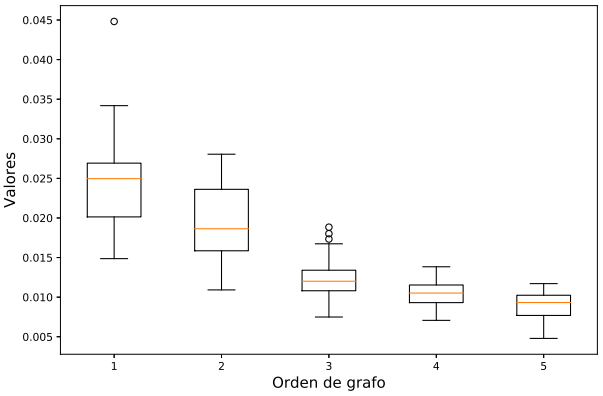


Figura 63: Diagrama de caja de load centrality de nodos sumidero.

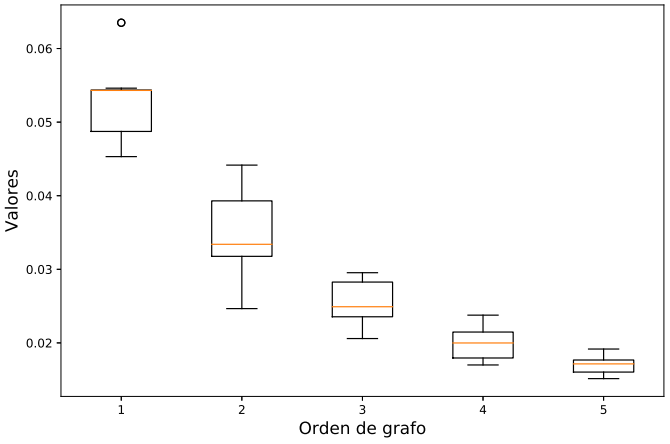


Figura 64: Diagrama de caja de pagerank de nodos fuentes.

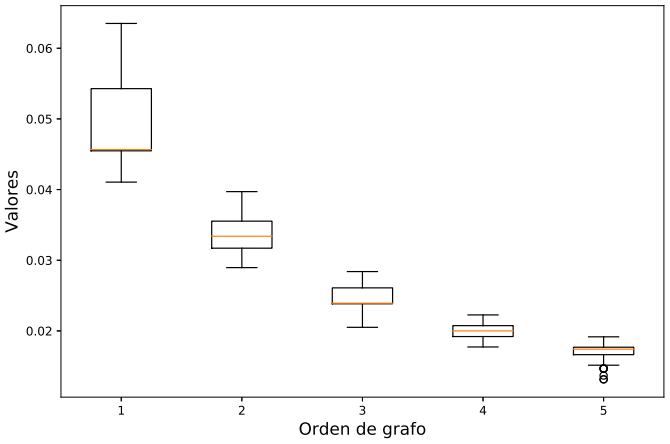


Figura 65: Diagrama de caja de load pagerank de nodos sumidero.

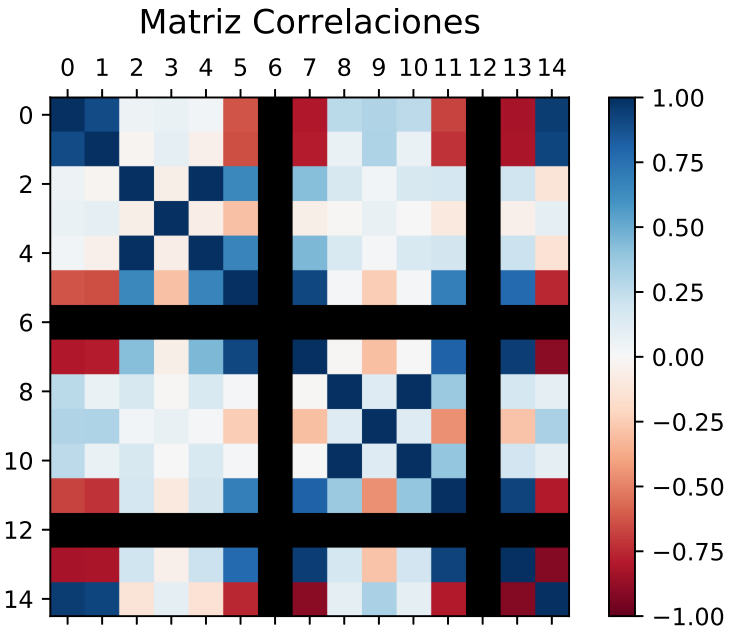


Figura 66: Correlaciones de las características de los nodos fuente y sumidero.

	sum_sq	df	F	PR>F
Deg_fuente	5846.313375	2.0	1.567752	2.112127e-01
Clstr_fuente	19842.361020	1.0	10.641886	1.311800e-03
Clsns_fuente	3926.661470	1.0	2.105953	1.483847e-01
Load_fuente	33331.798014	1.0	17.876562	3.667508e-05
Ex_fuente	33.471134	1.0	0.017951	8.935590e-01
Prank_fuente	29028.620403	1.0	15.568675	1.120869e-04
Clstr_sumidero	1220.063324	1.0	0.654346	4.195802e-01
Clsns_sumidero	2692.447162	1.0	1.444017	2.309939e-01
Load_sumidero	3810.709229	1.0	2.043766	1.544818e-01
Ex_sumidero	33.471134	1.0	0.017951	8.935590e-01
Prank_sumidero	54301.746662	1.0	29.123197	2.018353e-07
Residual	352400.527109	189.0	NaN	NaN

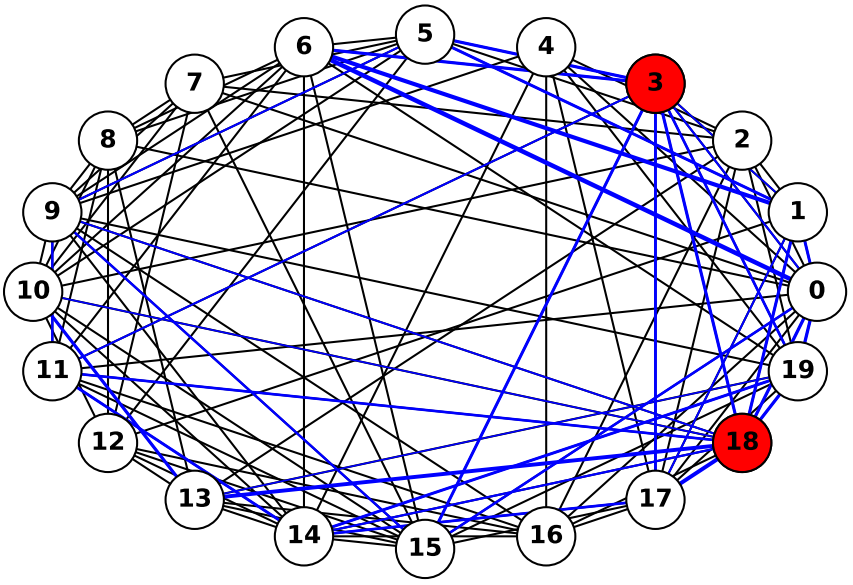


Figura 67: Grafo con rutas de flujo máximo.

5.6. Tarea cinco corregida

Introducción

En un grafo formado por aristas y nodos, donde cada arista tiene un valor que se interpreta como la capacidad, algo de interés es, dado dos nodos identificados como **fuente** y **sumidero**, es saber cual es el valor del flujo máximo que puede transportar desde el fuente al sumidero. Otros aspectos importantes es ver como las características del nodo fuente y el nodo sumidero afectan al valor del flujo máximo en el mismo grafo.

En este documento se habla acerca de las características que presentan los nodos en un grafo y de qué manera afectan estos en el valor del flujo máximo mediante un algoritmo de `Networkx`.

5.7. Especificaciones técnicas

La computadora en la que se implementaron los algoritmos es una Macbook Pro, cuyo procesador es un Intel Core i5 de 2.3 GHz. Tiene una memoria RAM de 8 GB 2133 MHz.

5.8. Generador

`watts_strogatz_graph()`

Este generador de grafos toma como parámetros la cantidad de nodos deseados, el número de nodos vecinos con los que se puede conectar y la probabilidad de conexión desde ese nodo con los nodos que puede conectar.

Usualmente, este tipo de grafos se usa para hacer las redes de mundo pequeño. Estas redes hacen referencia a que por más alejado que esté un nodo del otro, todos los nodos son accesibles desde otro nodo.

En las figuras 67, 83, 84, 85, 86 se muestran grafos que son hechos con este generador.

5.9. Algoritmos

En esta sección se habla de los algoritmos que se usaron junto con una breve descripción.

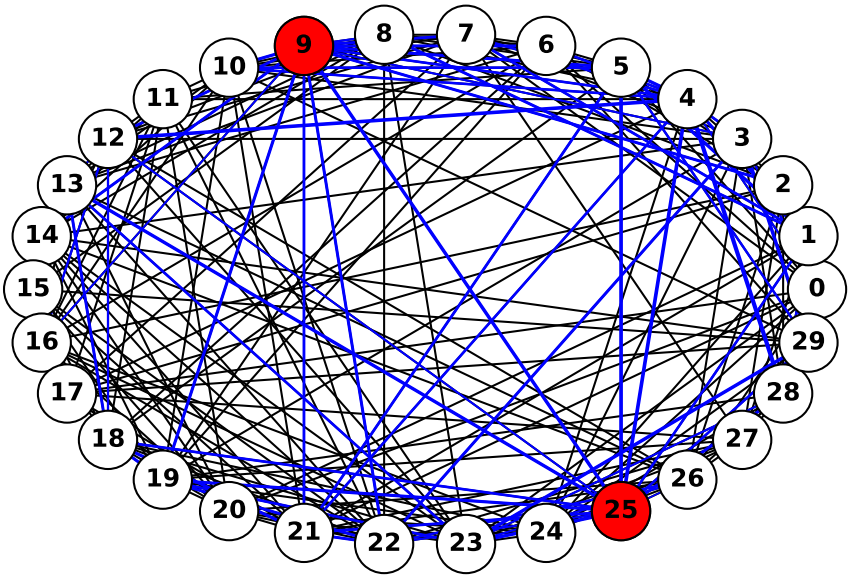


Figura 68: Grafo con rutas de flujo máximo.

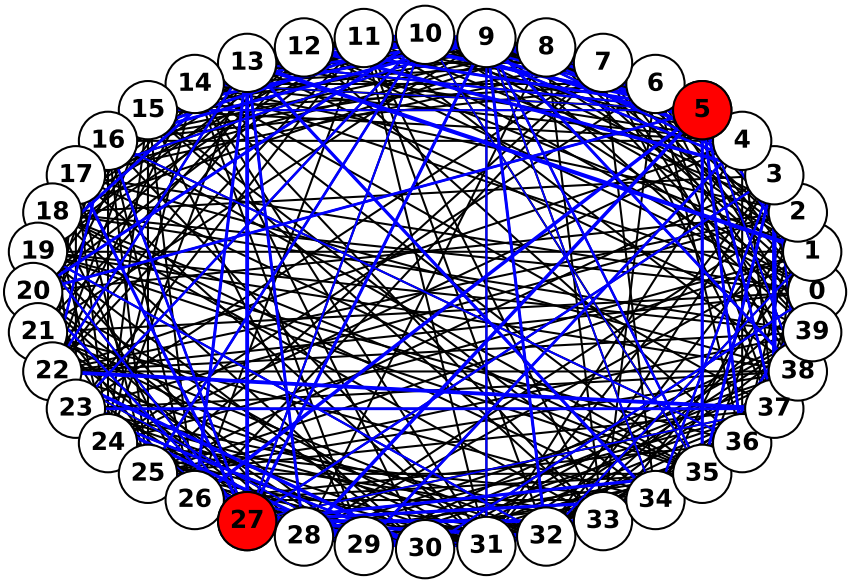


Figura 69: Grafo con rutas de flujo máximo.

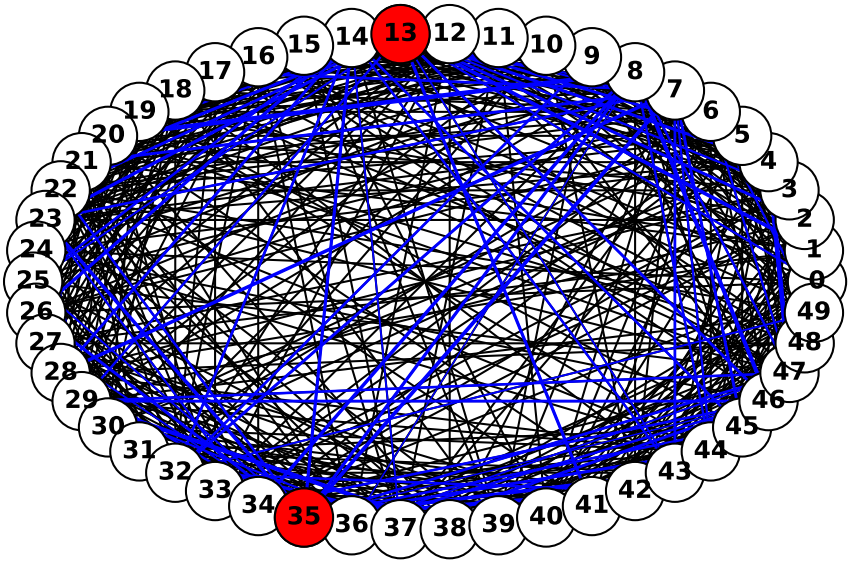


Figura 70: Grafo con rutas de flujo máximo.

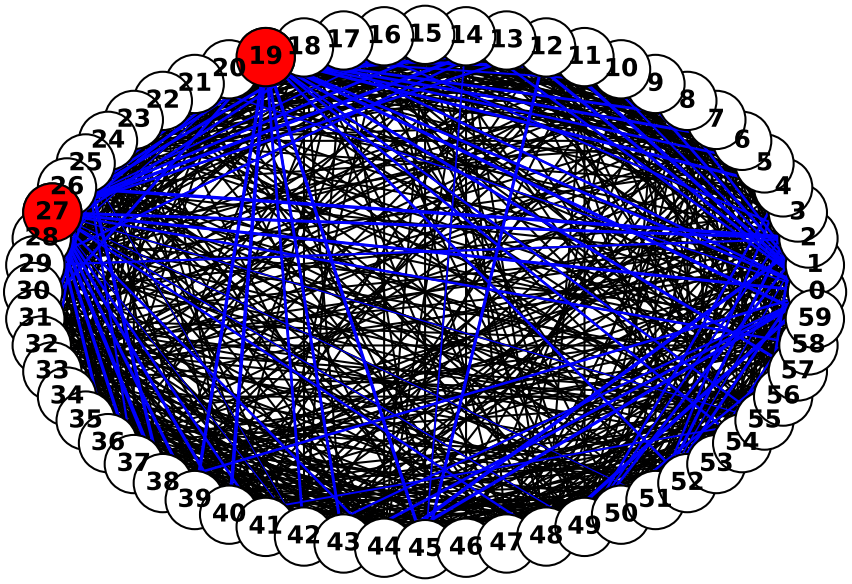


Figura 71: Grafo con rutas de flujo máximo.

Algoritmo de flujo máximo

El algoritmo implementado para encontrar el flujo máximo es `maximum_flow()` el cual toma como parámetros un grafo el cual debe tener como atributo la capacidad de cada arista, junto con el par de nodos fuente y sumidero. Al encontrar el flujo máximo devuelve el valor de este, así como el flujo que pasa por cada arista para obtener ese valor del flujo máximo.

`degree_centrality()`

`clustering()`

Este algoritmo que recibe como parámetro un grafo que puede ser con pesos en sus aristas o sin pesos; el caso que se tiene aristas con pesos, devuelve el promedio geométrico de los pesos del subgrafo.

`closeness_centrality()`

Mediante este algoritmo se obtiene una medida de centralidad de la red, es decir, para cada nodo se calcula que tan central es; entre más al centro esté el nodo está más cerca de todos los demás nodos.

`load_centrality()`

Una vez que se obtienen todas las rutas más cortas, el dato que ofrece este algoritmo es la fracción de todas las rutas más cortas que pasan a través de ese nodo.

`eccentricity()`

La información que aporta este algoritmo es para determinar qué tan cerca o tan lejos está un nodo con respecto a los demás.

`pagerank()`

Este algoritmo calcula una clasificación de los nodos en el gráfico de acuerdo a las características de los nodos con quien está conectado.

5.10. Metodología

El interés de este estudio es determinar qué características de los nodos influyen en un mayor valor de flujo máximo. Lo que se hace es generar un grafo de diferentes tamaños, desde veinte nodos hasta sesenta nodos con incrementos de diez nodos, una vez que el grafo ya tiene capacidad se eligen los pares de nodos fuente y sumidero, se calcula el valor del flujo máximo y se guarda las características que tienen éstos nodos.

La implementación en python se lleva a cabo con el código siguiente.

```
#--------------------Experimentacion-------------------------------
datos=np.arange(sum(orden)*15, dtype=float).reshape(sum(orden),15)
fila=0
for i in range(len(orden)):
    H=nx.watts_strogatz_graph(orden[i], int(orden[i]/2) , 0.33 , seed=None)
    initial=0
    final=0
    lista=[]
    lista[:]=H.edges
    width=np.arange(len(lista)*1,dtype=float).reshape(len(lista),1)
    for r in range(len(lista)):
        R=np.random.normal(loc=20, scale=5.0, size=None)
        width[r]=R
        H.add_edge(lista[r][0],lista[r][1],capacity=R)
    for w in range(orden[i]):
        initial=random.randint(0,round(len(H.nodes)/2))
        final=random.randint(initial,len(H.nodes)-2)
        while initial==final:
            initial=random.randint(0,round(len(H.nodes)/2))
            final=random.randint(initial,len(H.nodes)-2)
        tiempo_inicial=time()
        T=nx.maximum_flow(H, initial, final)
        tiempo_final=time()
        datos[fila,0]=T[0]
        datos[fila,1]=tiempo_final-tiempo_inicial
#--------------------Info Fuente-----------------------------------
        datos[fila,2]=nx.degree_centrality(H)[initial]
        datos[fila,3]=nx.clustering(H, nodes=initial)
        datos[fila,4]=nx.closeness_centrality(H, u=initial)
        datos[fila,5]=nx.load_centrality(H, v=initial)
        datos[fila,6]=nx.eccentricity(H, v=initial)
        datos[fila,7]=nx.pagerank(H, alpha=0.9, weight='weight')[initial]
#--------------------Info Sumidero---------------------------------
        datos[fila,8]=nx.degree_centrality(H)[final]
        datos[fila,9]=nx.clustering(H, nodes=final)
        datos[fila,10]=nx.closeness_centrality(H, u=final)
        datos[fila,11]=nx.load_centrality(H, v=final)
        datos[fila,12]=nx.eccentricity(H, v=final)
        datos[fila,13]=nx.pagerank(H, alpha=0.9, weight='weight')[final]
        datos[fila,14]=orden[i]
        fila+=1
```

caracterizacion.py

5.11. Resultados

Se elaboró unos diagramas de caja para ver el comportamiento de las características de los nodos fuente y sumidero 72, 73, 74, 75, 76, 77, 78, 79, 80. Los valores del eje de las abscisas representan el orden de los grafos.

Con esa información se hace una matriz de correlación 81 donde el cero es el valor del flujo máximo, el uno es el valor del tiempo, la fila cinco es el valor de load centrality del nodo fuente, el siete representa el pagerank del nodo fuente, las filas once y trece es el valor de load centrality y pagerank del nodo sumidero respectivamente.

Las filas y columnas en negro, es debido a que representan la excentricidad y su valor permanece constante para todos los grafos.

Lo que nos muestra 81 es que el valor de load centrality y el pagerank, tanto del nodo fuente como del sumidero afectan inversamente al valor del flujo máximo.

Luego, se hizo un ANOVA para identificar si las medias de los conjuntos eran iguales o hay diferencias significativas en ellas. Se obtuvo los resultados que se muestra en la tabla llamada `ANOVA.txt` en el cual los valores de la tercer columna que son mayores a 0.1, sus medias son iguales y los valores menores 0.1 se rechaza la hipótesis de que las medias sean iguales.

`ANOVA.txt`

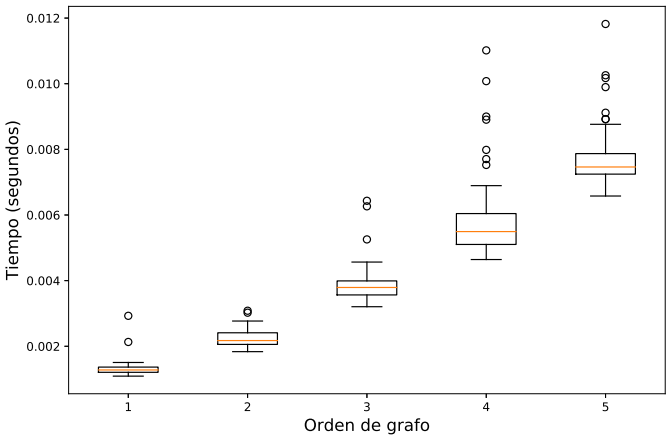


Figura 72: Diagrama de caja de tiempo.

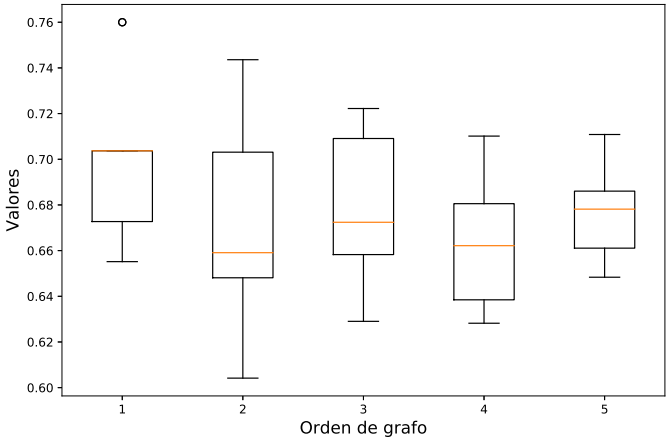


Figura 73: Diagrama de caja de centralidad de nodos fuentes.

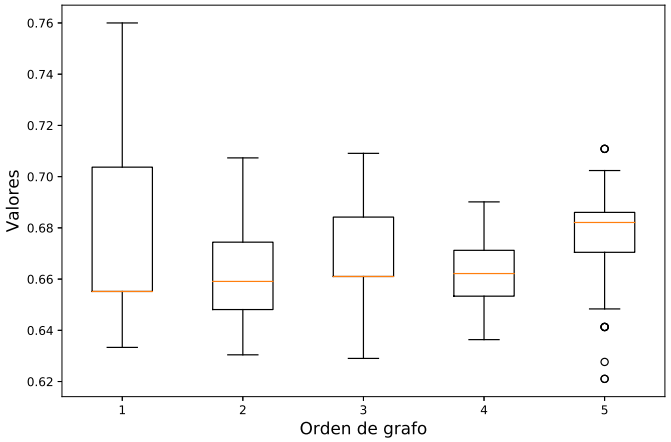


Figura 74: Diagrama de caja de centralidad de nodos sumidero.

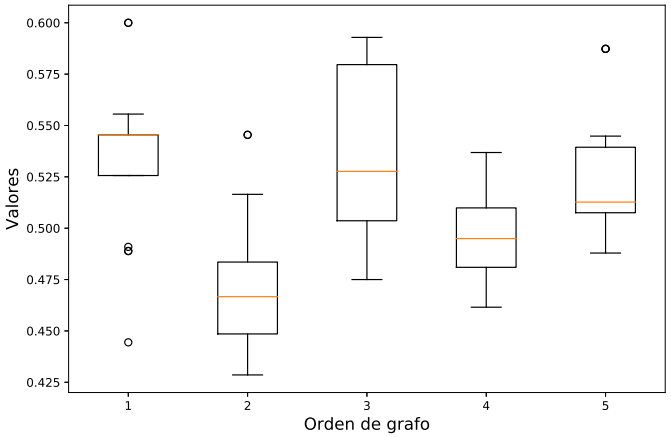


Figura 75: Diagrama de caja de clustering de nodos fuentes.

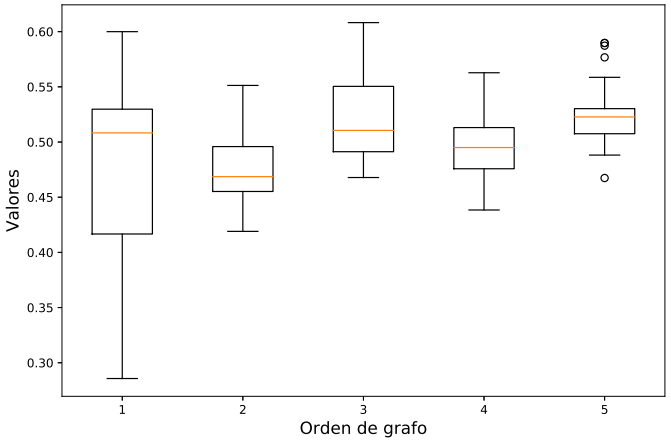


Figura 76: Diagrama de caja de clustering de nodos sumidero.

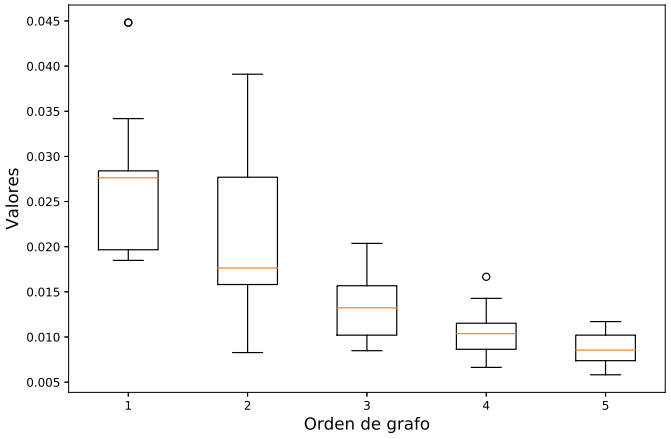


Figura 77: Diagrama de caja de load centrality de nodos fuentes.

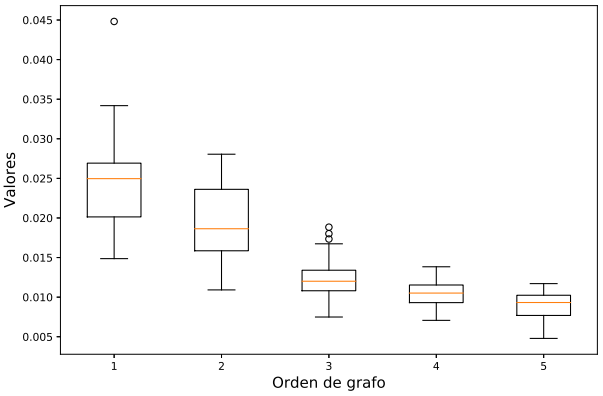


Figura 78: Diagrama de caja de load centrality de nodos sumidero.

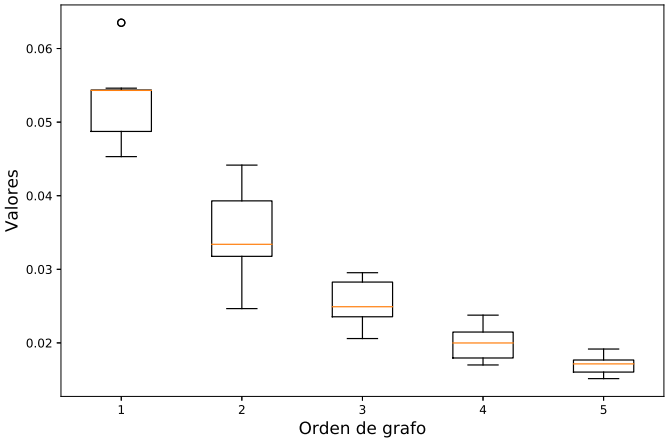


Figura 79: Diagrama de caja de pagerank de nodos fuentes.

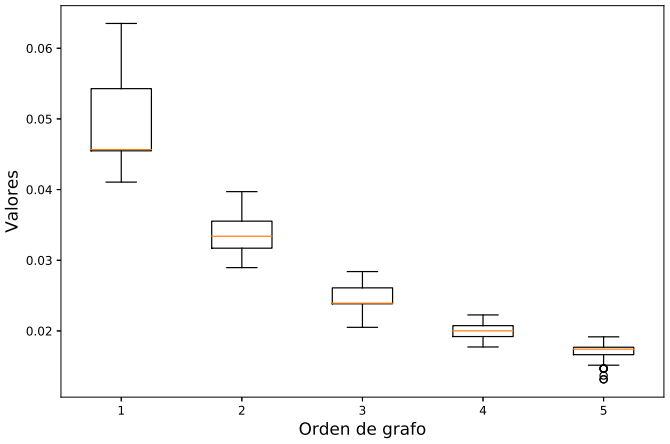


Figura 80: Diagrama de caja de load pagerank de nodos sumidero.

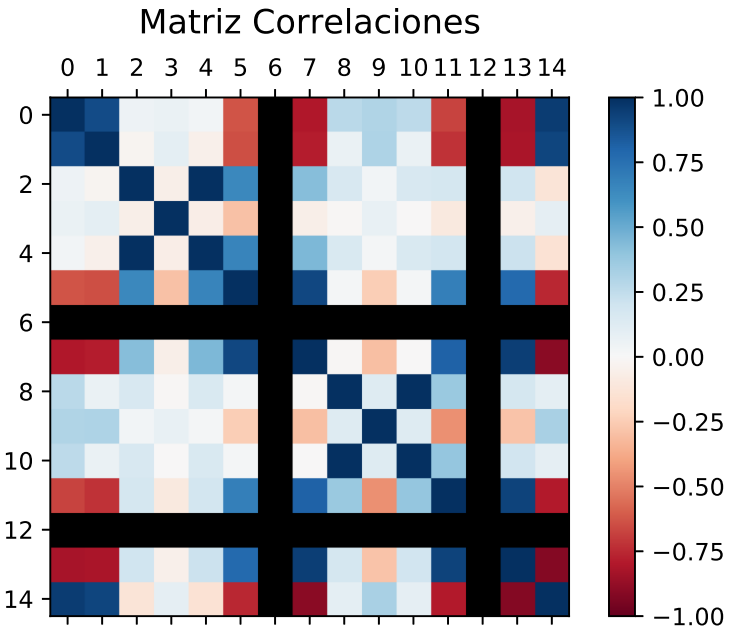


Figura 81: Correlaciones de las características de los nodos fuente y sumidero.

	sum_sq	df	F	PR>F
Deg_fuente	5846.313375	2.0	1.567752	2.112127e-01
Clstr_fuente	19842.361020	1.0	10.641886	1.311800e-03
Clsns_fuente	3926.661470	1.0	2.105953	1.483847e-01
Load_fuente	33331.798014	1.0	17.876562	3.667508e-05
Ex_fuente	33.471134	1.0	0.017951	8.935590e-01
Prank_fuente	29028.620403	1.0	15.568675	1.120869e-04
Clstr_sumidero	1220.063324	1.0	0.654346	4.195802e-01
Clsns_sumidero	2692.447162	1.0	1.444017	2.309939e-01
Load_sumidero	3810.709229	1.0	2.043766	1.544818e-01
Ex_sumidero	33.471134	1.0	0.017951	8.935590e-01
Prank_sumidero	54301.746662	1.0	29.123197	2.018353e-07
Residual	352400.527109	189.0	NaN	NaN

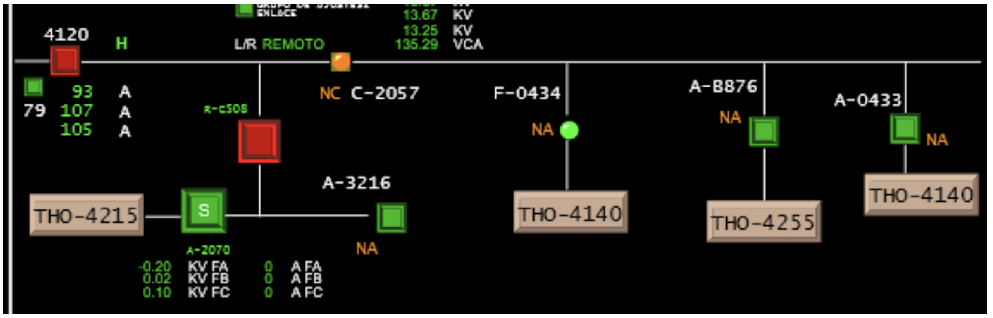


Figura 82: Ejemplo de circuito.

6. Tarea seis

Introducción

En este trabajo se realiza una visualización de cómo se encuentra configurado cierto circuito de la red de distribución energética, se usa el algoritmo de flujo máximo para dar apoyo a la visualización.

6.1. Contexto

El enfoque de la tesis en la que trabajo es el encontrar el orden de las **maniobras** a realizar en los circuitos de distribución de electricidad, de manera que la carga esté balanceada y la ejecución de estas maniobras sean al mejor costo posible. Entiéndase por maniobra el tipo de operación realizada por una persona o un grupo de ellas con el motivo de modificar la manera en que la red está conectada. Una particularidad de ésta red es que los clientes se encuentran en las aristas, por lo que la demanda se presenta en las mismas, mientras que los nodos sirven de conexión entre las lineas y para cambiar su topología.

Como se aprecia en la figura 82 un circuito está conectado por un alimentador que es el primer cuadro cuyo nombre es `4120` y los demás componentes. Los rectángulos cafés representan otros circuitos que están conectados a ese mismo circuito, se hace de ésta manera ya que si el alimentador 4120 tiene que ser apagado para realizar un mantenimiento, el circuito se modifique para que no tenga que estar desabastecido de energía.

Los nodos en color rojo de la figura 82 representan componentes cerrados, es decir por donde fluye la energía. Y los nodos en color verde son aquellos donde el circuito está abierto.

6.2. Metodología

Primero se ingresaron los datos para poder representar la red, lo que se obtiene es un grafo como 83. Lo que se quiere representar en ese grafo es que el alimentador 4120 está **cargando** toda la red, entonces ya que las demandas están en las aristas, se cambia el modelo de red por el que se muestra en la figura 84 donde los nodos simbolizan las lineas y las aristas son los componentes que las unen. Las etiquetas de las lineas fueron tomadas por una búsqueda en profundidad del circuito, se hizo por el motivo de tener un orden al momento de ingresar la información. También hay nodos auxiliares, éstos significan las intersecciones de las lineas de distribución en las cuales no puede haber maniobras, lo que quiere decir es que siempre esta el circuito cerrado. Para visualizar la carga que presenta el alimentador 4120 del ejemplo de circuito, se puede encontrar el flujo máximo desde la fuente al sumidero.

```
cerrados=['4120', 'R-c508', 'C-2057']
abiertos=['A-2070', 'A-B876', 'F-0434', 'A-0433']
aux=['Aux','Aux1', 'Aux2', 'Aux3', 'Aux4']


Grafo_disp=nx.Graph()
Grafo_disp.add_edges_from([('4120','Aux'),('Aux','R-c508'),('R-c508','Aux1'),
                           ('Aux1','A-2070'),('Aux','C-2057'),
                           ('C-2057','Aux2'),('Aux2','F-0434'),
                           ('Aux2','Aux3'),('Aux3','A-B876'),
                           ('Aux3','Aux4'),('Aux4','A-0433')])

labels_disp={('4120','Aux'): 'H1', ('Aux','R-c508'): 'V1',
             ('R-c508','Aux1'):'V2',('Aux1','A-2070'):'H3',
             ('Aux','C-2057'): 'H2', ('C-2057','Aux2'): 'H4',
             ('Aux2','F-0434'): 'V3',('Aux2','Aux3'): 'H5',
             ('Aux3','A-B876'): 'V4', ('Aux3','Aux4'): 'H6',
             ('Aux4','A-0433'): 'V5'}

labels={('H1','H2'): 'Aux', ('H1','V1'): 'Aux', ('H2','H4'):'C-2057',
        ('V1','V2'):'R-C508', ('V2','H3'): 'Aux1', ('H4','H5'): 'Aux2',
        ('H4','V3'): 'Aux2',('H5','H6'): 'Aux3', ('H5','V4'): 'Aux3',
        ('H6','V5'): 'Aux4',('H3','Sumidero'): 'A-2070',
        ('V3','Sumidero'): 'F-0434', ('V4','Sumidero'): 'A-B876',
        ('V5','Sumidero'): 'A-0433',('Fuente', 'H1'): '4120',
        ('Fuente', 'H3'): 'A-2070', ('Fuente', 'V3'): 'F-0434',
        ('Fuente','V4'): 'A-B876', ('Fuente','V5'): 'A-0433'}

Grafo_lineas=nx.Graph()
Grafo_lineas.add_edges_from(list(labels.keys()))

                                  #----------Graficar----------
nx.draw_networkx_nodes(Grafo_disp, pos_disp, nodelist=abiertos, node_color='green',
    node_shape='s')
nx.draw_networkx_nodes(Grafo_disp, pos_disp, nodelist=cerrados, node_color='red',
    node_shape='s')
nx.draw_networkx_nodes(Grafo_disp, pos_disp, nodelist=aux, alpha=0)
nx.draw_networkx_edges(Grafo_disp, pos_disp, edge_color='white')
plt.axis('off')
plt.style.use('dark_background')
plt.show()
```

proyecto.py

Para obtener el flujo presente en la red se asigna una demanda que sale del nodo `Fuente`, se tienen varios componentes por los cuales al estar cerrados puede haber energía no solo del alimentador 4120, por lo que si éstos componentes se encuentran abiertos se asigna una capacidad de cero en la arista que conecta esa componente y la Fuente. Luego a los componentes que estén abiertos se conectan con el nodo `Sumidero` para después ajustar la capacidad de las aristas que llegan al Sumidero como un promedio entre la demanda y el numero de aristas. El flujo que se obtiene se grafica en la representación del circuito quedando como se presenta en la figura 85, donde ya es posible visualizar la carga del circuito.

Al cambiar la configuración actual del circuito a otra donde el componente `C-2057` sea abierto, la red queda como en la figura 86 donde se ve que hay un segmento de aristas que no están conectadas. Con ayuda de esta visualización y al continuar trabajando se pretende hacer un simulador de la red.

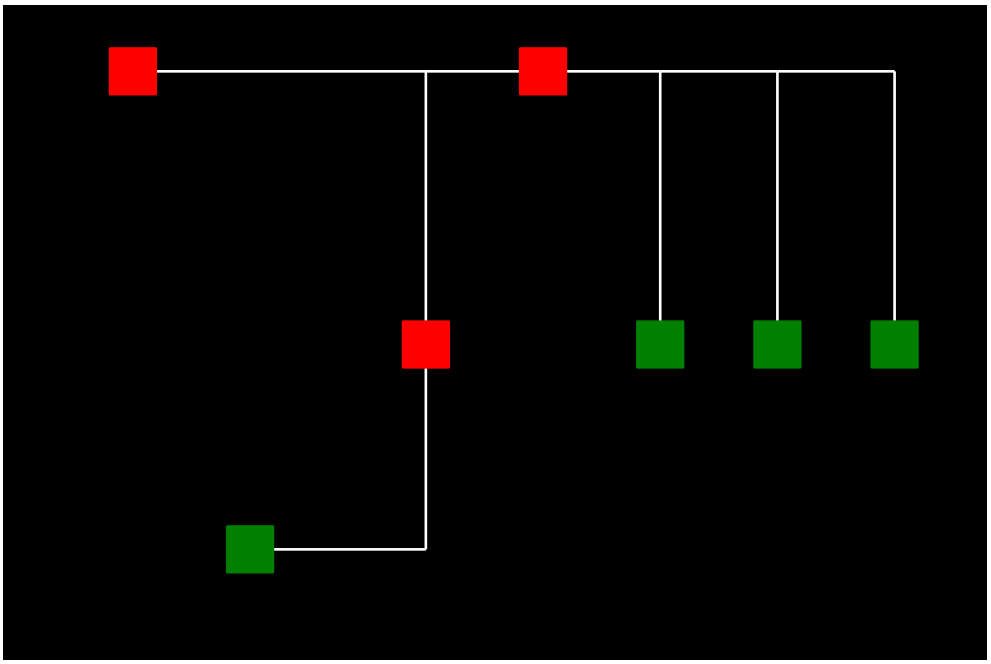


Figura 83: Representación de circuito.

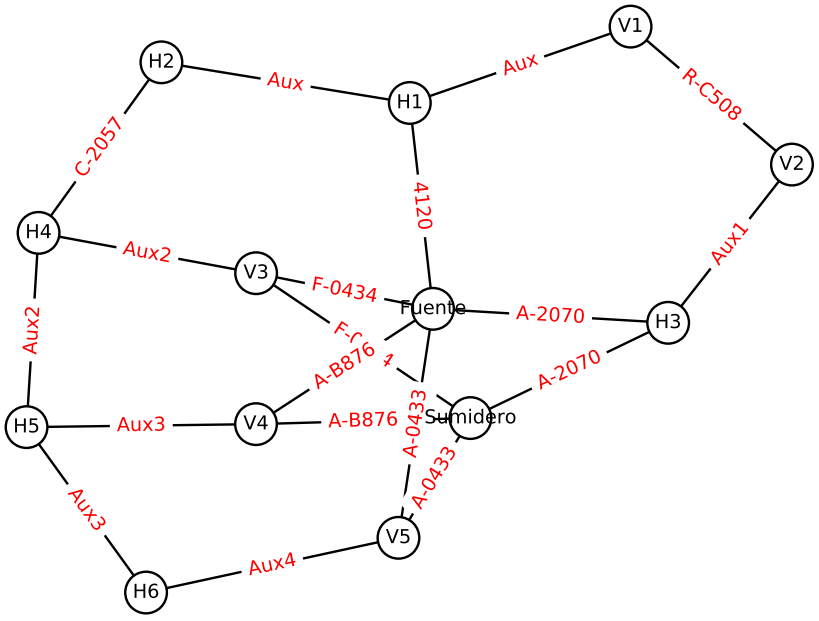


Figura 84: Grafo auxiliar.

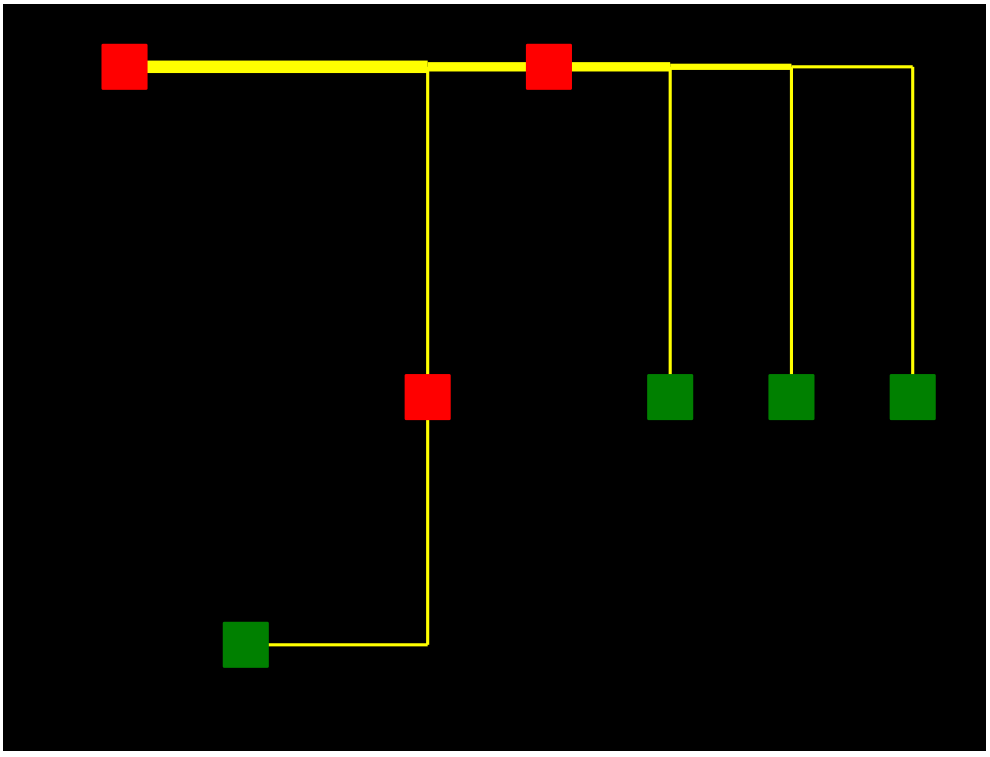


Figura 85: Visualización de configuración de la red.

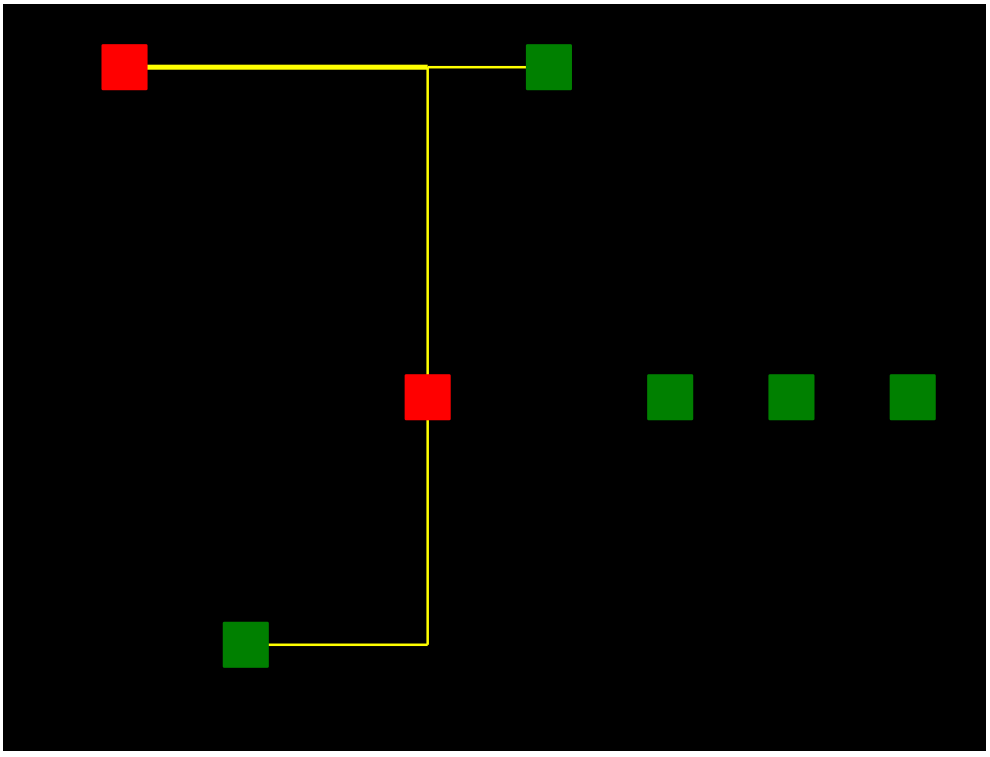


Figura 86: Visualización de configuración de la red.

Referencias

[1] A. C. Barrero, G. W. de García, and R. M. M. Parra, *Introducción a la Teoría de Grafos*. ELIZCOM SAS, 2010.

[2] NetworkX Developers, "Networkx documentation," 2012.

[3] J. M. Six and I. G. Tollis, "A framework for circular drawings of networks," in *International Symposium on Graph Drawing*. Springer, 1999, pp. 107–116.

[4] J. L. Molina, "La ciencia de las redes," *Apuntes de Ciencia y Tecnología*, vol. 11, no. 1, pp. 36–42, 2004.

[5] U. Brandes, P. Kenis, and J. Raab, "La explicación a través de la visualización de redes." *REDES. Revista hispana para el análisis de redes sociales*, vol. 9, no. 2, 2005.

[6] E. Coto, "Algoritmos básicos de grafos," *Lecturas en Ciencias de computación. ISSN 1316*, vol. 6239, 2003.

[7] A. R. Villalobos, "Grafos: herramienta informática para el aprendizaje y resolución de problemas reales de teoría de grafos." in *X Congreso de Ingeniería de Organización*, 2006.