



Universidad Autónoma de Nuevo León

**Facultad de Ingeniería Mecánica y
Eléctrica**

Optimización de flujo en redes

PORTAFOLIO

Héctor Hugo García López
1484166

Enero-Junio 2019

Dra. Elisa Schaeffer

9.5

4/66

Tipos De Grafos

Héctor Hugo García López

Introducción

Un grafo está 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 librerías, como se presenta a continuación.

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

PrimerProyecto.py

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 1 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.

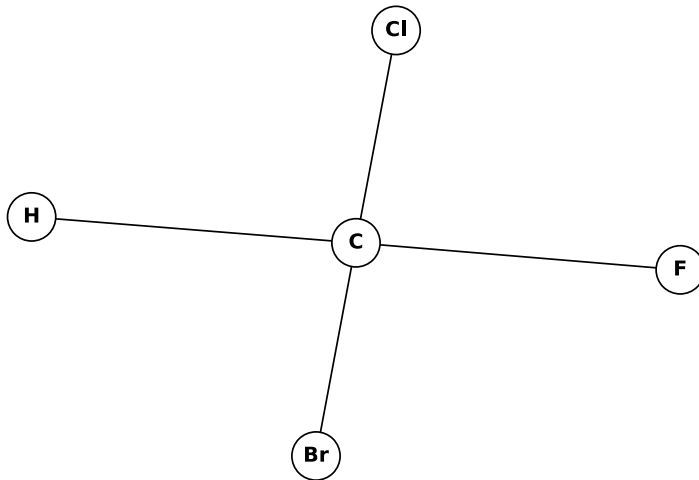


Figura 1: Grafo no dirigido acíclico

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.

ee

jj

2. Grafo simple no dirigido cíclico

En la sección anterior se vió que es un ciclo en un grafo, en la figura 2 se observa un grafo que representa la caja de cambios de un carro, donde el nodo 0 es estar en neutral.

$\$0\$$ $\$n-0\$$

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.

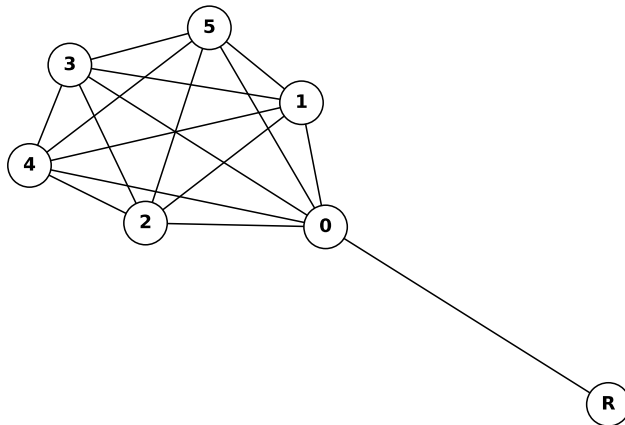


Figura 2: Grafo no dirigido cíclico

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 3 se tiene el horario de cierto profesor, los nodos muestran

detexify

las materias y las aristas las horas que da de clase en una semana específica. 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.

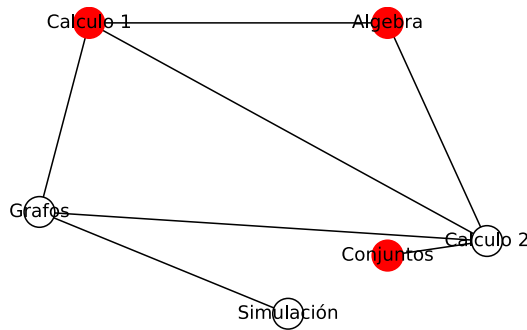


Figura 3: Grafo no dirigido reflexivo

$\setminus \{ \dots \}$

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.

```

1 #Tercero simple no dirigido reflexivo
2 G=nx.Graph()
3 G.add_edges_from([( 'Calculo 1', 'Algebra'), ('Calculo 1', 'Calculo 2')
4                 ,
5                 ( 'Calculo 1', 'Grafos'), ('Algebra', 'Calculo 2'),
6                 ( 'Calculo 2', 'Grafos'), ('Simulaci n', 'Grafos'),
7                 ( 'Calculo 2', 'Conjuntos')])
8
9 nodos1 = { 'Calculo 1', 'Algebra', 'Conjuntos' }
10 nodos2 = { 'Calculo 2', 'Simulaci n', 'Grafos' }
11
12 pos = { 'Calculo 1':(20, 35), 'Algebra':(50,35),
13         'Calculo 2':(60, 20), 'Simulaci n':(40,15),
14         'Grafos':(15,22), 'Conjuntos':(50,19) }
15
16 nx.draw_networkx_nodes(G, pos, nodelist=nodos1, node_size=400,
17                        node_color='r', node_shape='o')
18 nx.draw_networkx_nodes(G, pos, nodelist=nodos2, node_size=400,
19                        node_color='w', node_shape='o',
20                        edgecolors="k")

```

```

21 nx.draw_networkx_edges(G, pos)
22 nx.draw_networkx_labels(G, pos)
23 plt.axis('off')
24
25 plt.savefig('Tercero.eps', format='eps', dpi=1000)

```

PrimerProyecto.py

page

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 4 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.

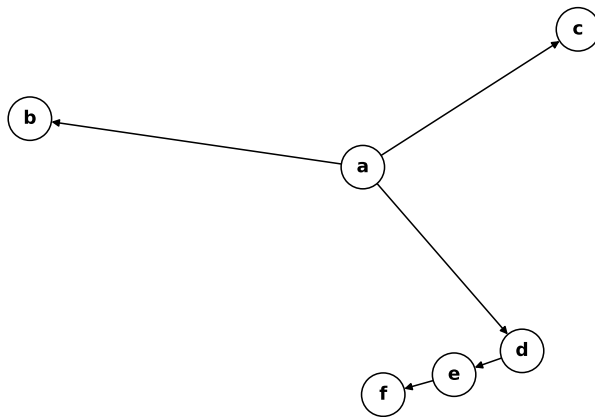


Figura 4: Grafo dirigido acíclico

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

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

PrimerProyecto.py

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 5 se aprecia el ciclo y las flechas que nos dice que es un grafo dirigido.

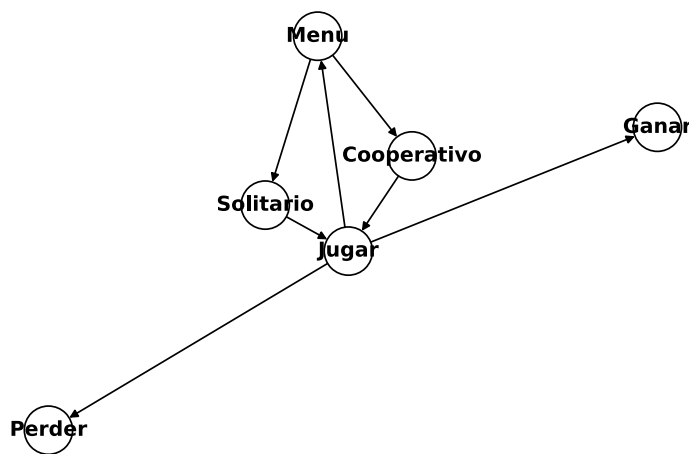


Figura 5: Grafo dirigido cíclico

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 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 6 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.

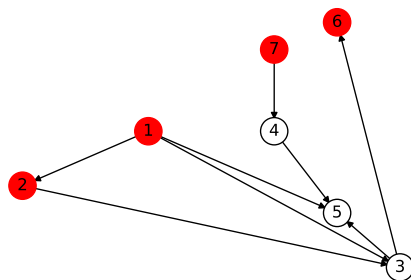


Figura 6: Grafo dirigido reflexivo

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 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 7 donde los nodos son los países y las aristas son la cantidad de vuelos.

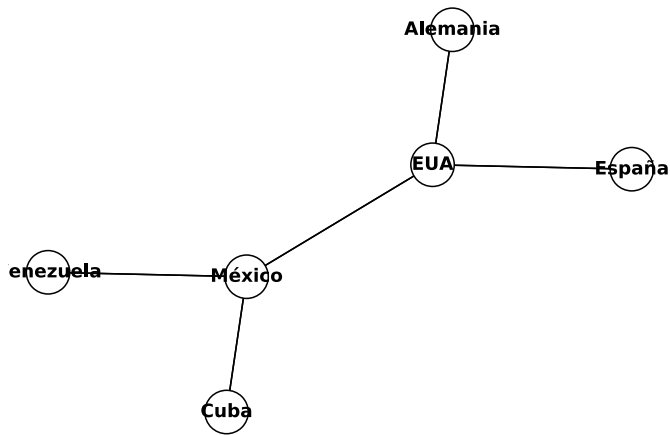


Figura 7: Multigrafo no dirigido acíclico

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

```

1 #Septimo multigrafo no dirigido aciclico
2 H=nx.MultiGraph()
3 H.add_edges_from([( 'Mexico', 'EUA'), ( 'EUA', 'Mexico'),
4                   ( 'EUA', 'España'), ( 'España', 'EUA'),
5                   ( 'Mexico', 'Venezuela'), ( 'Venezuela', 'Mexico'),
6                   ( 'Cuba', 'Mexico'), ( 'Mexico', 'Cuba'),
7                   ( 'EUA', 'Alemania'), ( 'Alemania', 'EUA')])
8 nx.draw(H, node_color="white", node_size=800, with_labels=True,
9         font_weight="bold", edgecolors="black")
10 plt.savefig( 'Septimo.eps', format='eps', dpi=1000)

```

PrimerProyecto.py

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 8 vemos que ahora, Venezuela y Cuba tienen vuelos entre sí, lo mismo que Venezuela y España, formándose dos ciclos

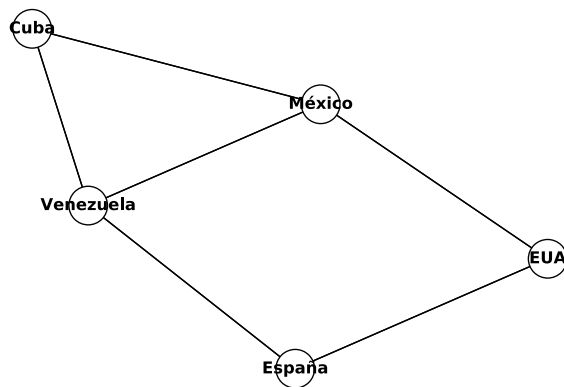


Figura 8: Multigrafo no dirigido cíclico

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

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.

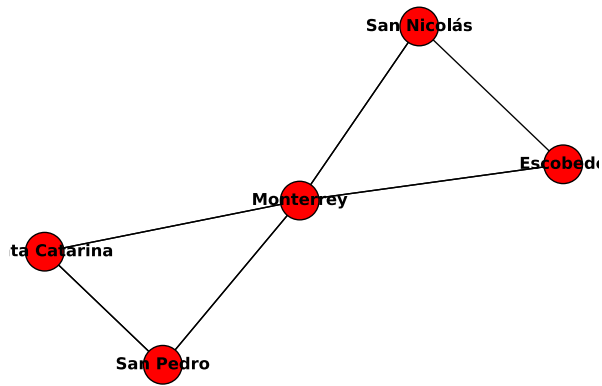


Figura 9: Multigrafo no dirigido reflexivo

10. Multigrafo dirigido acíclico

En la figura 10 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.

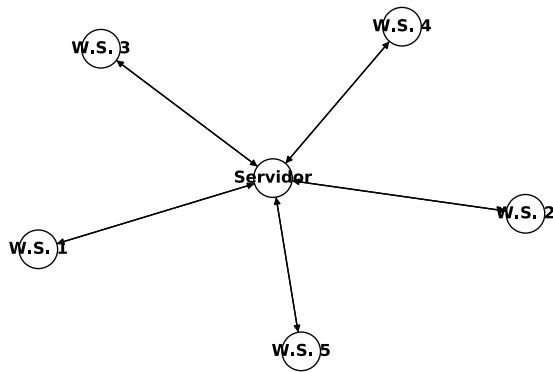


Figura 10: Multigrafo dirigido acíclico

El código necesario para la gráfica es el siguiente

```
1 #Decimo multigrafo dirigido aciclico
2 H=nx.MultiDiGraph()
3 H.add_edges_from([( 'Jose', 'Claudia'), ( 'Claudia', 'Jose'),
4                   ( 'Claudia', 'Agustin'), ( 'Agustin', 'Claudia'),
5                   ( 'Agustin', 'Priscila'), ( 'Agustin', 'Felipe'),
6                   ( 'Agustin', 'Cristina'), ( 'Cristina', 'Carlos'),
7                   ( 'Carlos', 'Pedro'), ( 'Pedro', 'Carlos'),
8                   ( 'Priscila', 'Carlos'), ( 'Carlos', 'Priscila')])
9 nx.draw(H, node_size=800, with_labels=True,
10         font_weight="bold", edgecolors="black", node_color='white')
11 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.

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

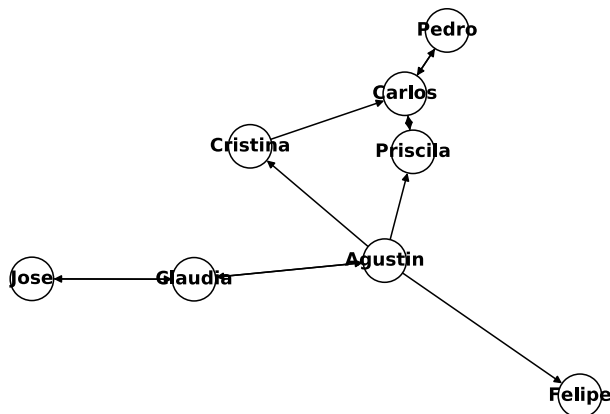


Figura 11: Multigrafo dirigido cíclico

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 12 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

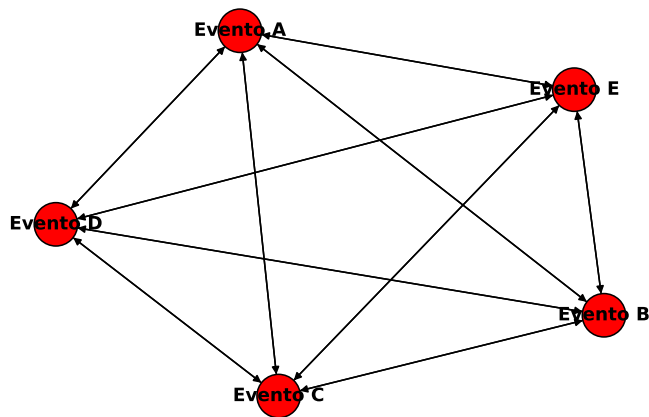


Figura 12: Multigrafo dirigido reflexivo

Referencias

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

1. 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 librerías, 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 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 1 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 #Primer 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('Primer.eps', format='eps', dpi=1000)
```

PrimerProyecto.py

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

1.2. Grafo simple no dirigido cíclico

Un ejemplo de un grafo cíclico es el de la figura 2, 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.

1.3. 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 ejemplo 3 se tiene el horario de cierto profesor, los nodos muestran las materias y las aristas las horas que da de clase en una semana específica. 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`.

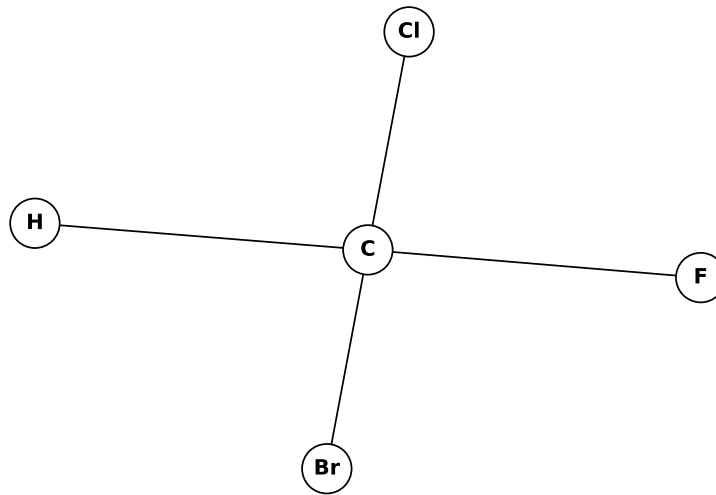


Figura 1: Grafo no dirigido acíclico

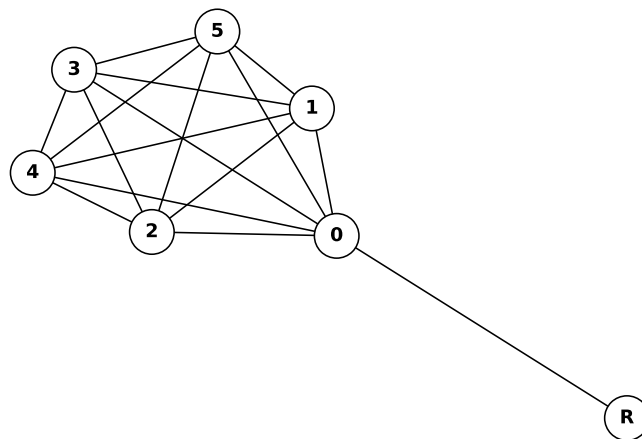


Figura 2: Grafo no dirigido cíclico

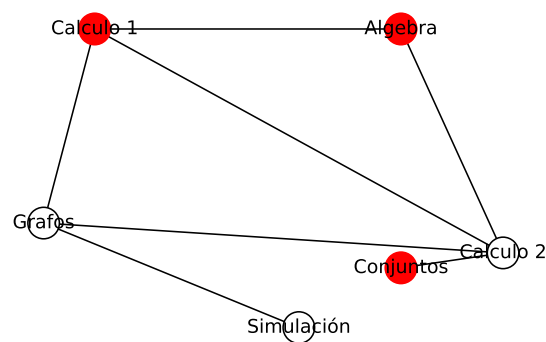


Figura 3: Grafo no dirigido reflexivo

```

1 #Tercero simple no dirigido reflexivo
2 G=nx.Graph()
3 G.add_edges_from([( 'Calculo 1', 'Algebra'), ('Calculo 1', 'Calculo 2'),
4                  ('Calculo 1', 'Grafos'), ('Algebra', 'Calculo 2'),
5                  ('Calculo 2', 'Grafos'), ('Simulaci n', 'Grafos'),
6                  ('Calculo 2', 'Conjuntos')])
7
8 nodos1 = { 'Calculo 1', 'Algebra', 'Conjuntos' }
9 nodos2 = { 'Calculo 2', 'Simulaci n', 'Grafos' }
10
11 pos = { 'Calculo 1':(20, 35), 'Algebra':(50,35),
12         'Calculo 2':(60, 20), 'Simulaci n':(40,15),
13         'Grafos':(15,22), 'Conjuntos':(50,19) }
14
15 nx.draw_networkx_nodes(G, pos, nodelist=nodos1, node_size=400,
16                        node_color='r', node_shape='o')
17 nx.draw_networkx_nodes(G, pos, nodelist=nodos2, node_size=400,
18                        node_color='w', node_shape='o',
19                        edgecolors="k")
20
21 nx.draw_networkx_edges(G, pos)
22 nx.draw_networkx_labels(G, pos)
23 plt.axis('off')
24
25 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 4 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.

```

1 #Cuarto simple dirigido aciclico
2 H=nx.DiGraph()
3 H.add_edges_from([( 'a', 'b'), ('a', 'c'), ('a', 'd'), ('d', 'e'),
4                  ('e', 'f'), ('c', 'a')])
5 nx.draw(H, node_color="white", node_size=800, with_labels=True,
6         font_weight="bold", edgecolors="black")
7 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 5 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 entran y se cuentan los mismos con el fin de ver si lo que llegó es lo que se pidió. En caso de no

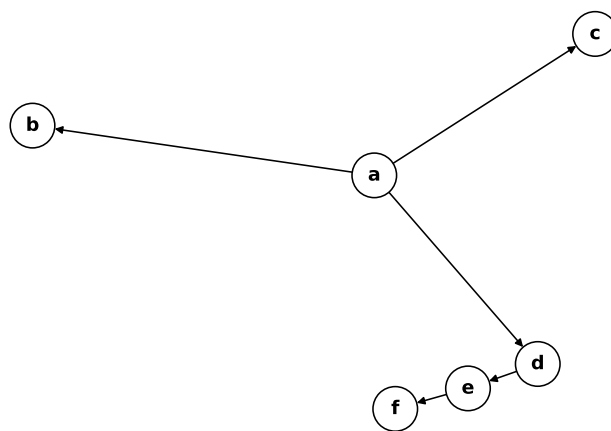


Figura 4: Grafo dirigido acíclico

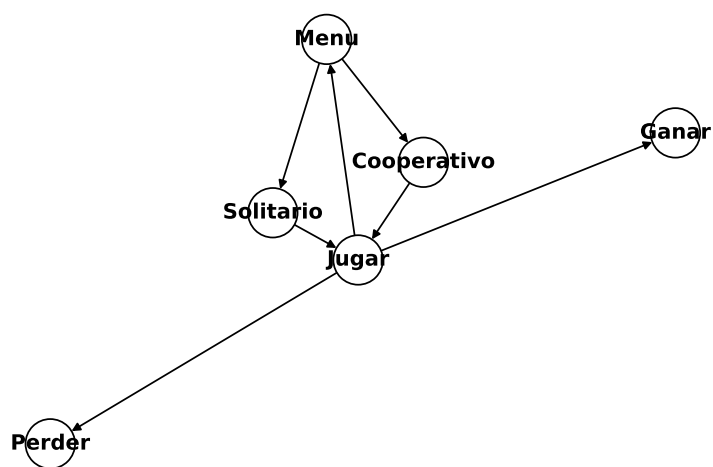


Figura 5: Grafo dirigido cíclico

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 6 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 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 7 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 agregar aristas que, aunque vistos como conjuntos son iguales, importan porque representan flujos (cantidades de vuelos en nuestro ejemplo) diferentes

```
1 #Septimo multigrafo no dirigido aciclico
2 H=nx.MultiGraph()
3 H.add_edges_from([( 'Mexico', 'EUA'), ( 'EUA', 'Mexico'),
4                   ( 'EUA', 'Espa a'), ( 'Espa a', 'EUA'),
5                   ( 'Mexico', 'Venezuela'), ( 'Venezuela', 'Mexico'),
6                   ( 'Cuba', 'Mexico'), ( 'Mexico', 'Cuba'),
7                   ( 'EUA', 'Alemania'), ( 'Alemania', 'EUA')])
8 nx.draw(H, node_color="white", node_size=800, with_labels=True,
9         font_weight="bold", edgecolors="black")
10 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 8 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

1.9. 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 10 se representa a cinco estaciones de trabajo (work station) las cuales están conectadas a un servidor. Las 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.

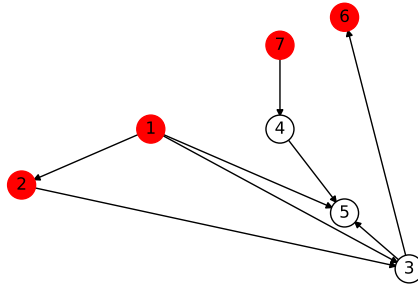


Figura 6: Grafo dirigido reflexivo

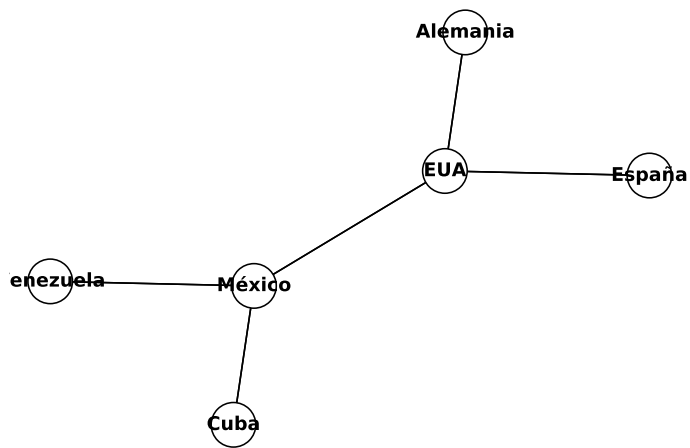


Figura 7: Multigrafo no dirigido acíclico

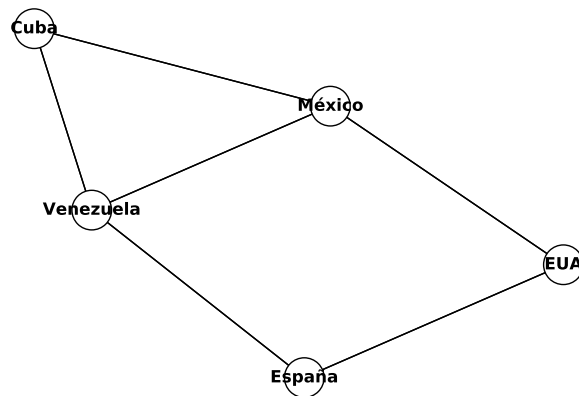


Figura 8: Multigrafo no dirigido cíclico

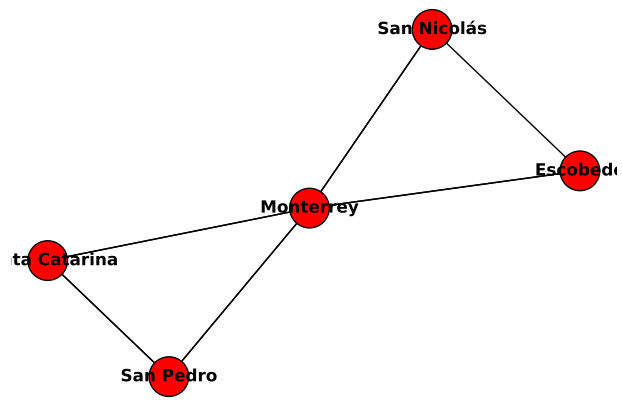


Figura 9: Multigrafo no dirigido reflexivo

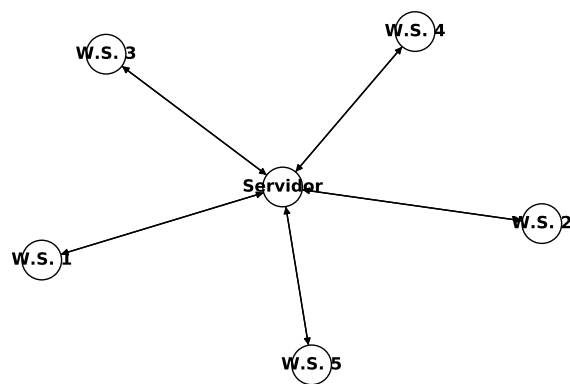


Figura 10: Multigrafo dirigido acíclico

```

1 #Decimo multigrafo dirigido aciclico
2 H=nx.MultiDiGraph()
3 H.add_edges_from([( 'Jose ', 'Claudia '), ( 'Claudia ', 'Jose '),
4                   ( 'Claudia ', 'Agustin '), ( 'Agustin ', 'Claudia '),
5                   ( 'Agustin ', 'Priscila '), ( 'Agustin ', 'Felipe '),
6                   ( 'Agustin ', 'Cristina '), ( 'Cristina ', 'Carlos '),
7                   ( 'Carlos ', 'Pedro '), ( 'Pedro ', 'Carlos '),
8                   ( 'Priscila ', 'Carlos '), ( 'Carlos ', 'Priscila ')]])
9 nx.draw(H, node_size=800, with_labels=True,
10         font_weight="bold", edgecolors="black", node_color='white')
11 plt.savefig('Decimo.eps', format='eps', dpi=1000)

```

PrimerProyecto.py

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 12 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

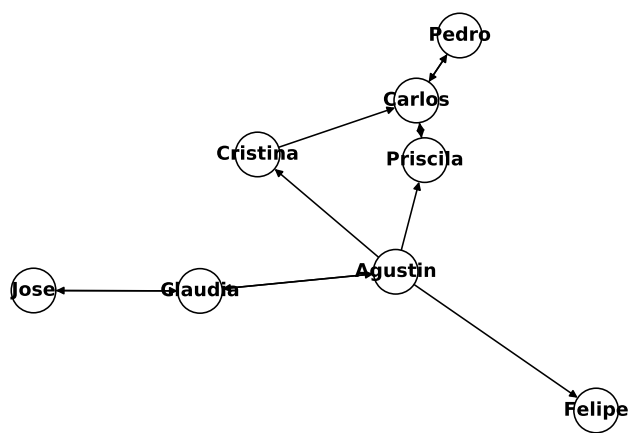


Figura 11: Multigrafo dirigido cíclico

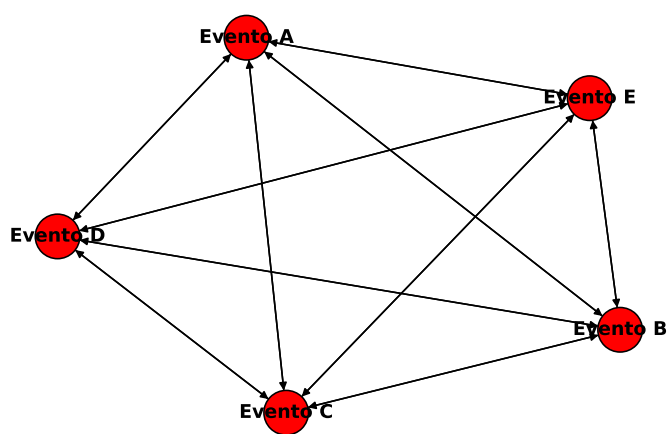


Figura 12: Multigrafo dirigido reflexivo

Diseño de los grafos

1484166



Introducción

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

1. Diseño bipartito

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

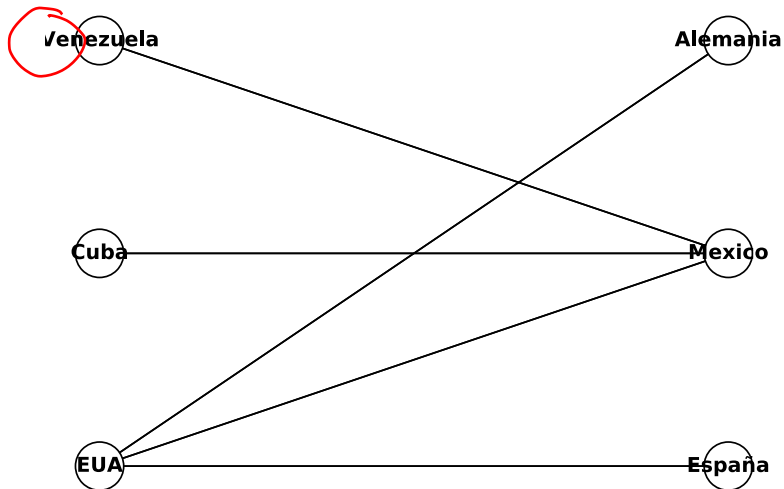


Figura 1: Grafo bipartito

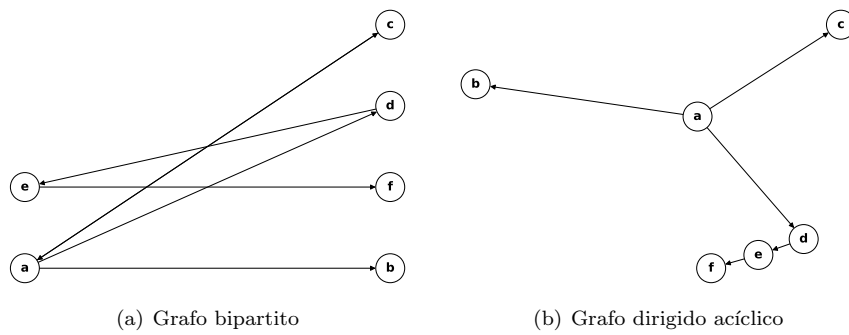


Figura 2: Comparativa entre grafo con acomodo bipartito vs grafo con acomodo por default

¿ser...?

El código que da paso a la figura 1 es el siguiente.

```

1 H=nx.MultiGraph()
2 H.add_nodes_from(['Venezuela', 'Mexico', 'Cuba'], bipartite=0)
3 H.add_nodes_from(['Español', 'EUA', 'Alemania'], bipartite=1)
4 H.add_edges_from([('Mexico', 'EUA'), ('EUA', 'Mexico'),
5                  ('EUA', 'Español'), ('Español', 'EUA'),
6                  ('Mexico', 'Venezuela'), ('Venezuela', 'Mexico'),
7                  ('Cuba', 'Mexico'), ('Mexico', 'Cuba'),
8                  ('EUA', 'Alemania'), ('Alemania', 'EUA')])
9 X,Y=bipartite.sets(H)
10 pos=dict()
11 pos.update((n, (1, i)) for i, n in enumerate(X))
12 pos.update((n, (2, i)) for i, n in enumerate(Y))
13 nx.draw(H, pos, node_color="white", node_size=800, with_labels=True
14         , font_weight="bold", edgecolors="black")
15 plt.savefig('SSeptimo.eps', format='eps', dpi=1000)

```

Grafos_layout.py

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()`.

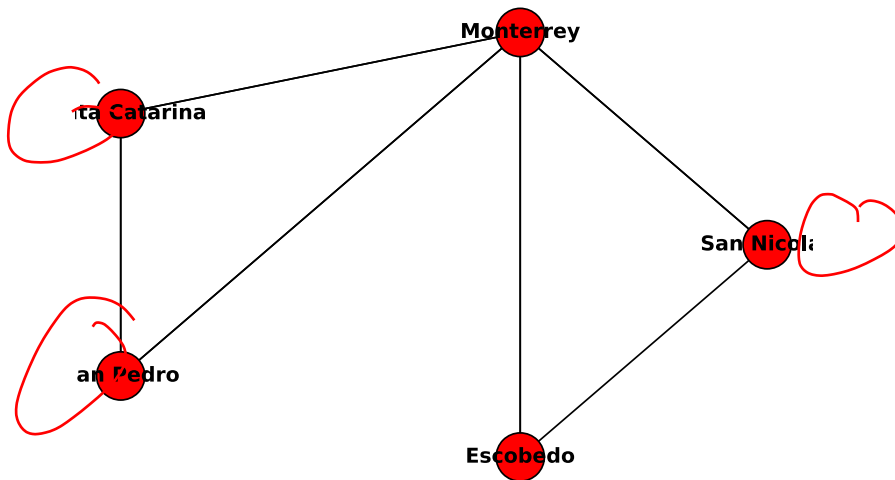


Figura 3: Grafo circular

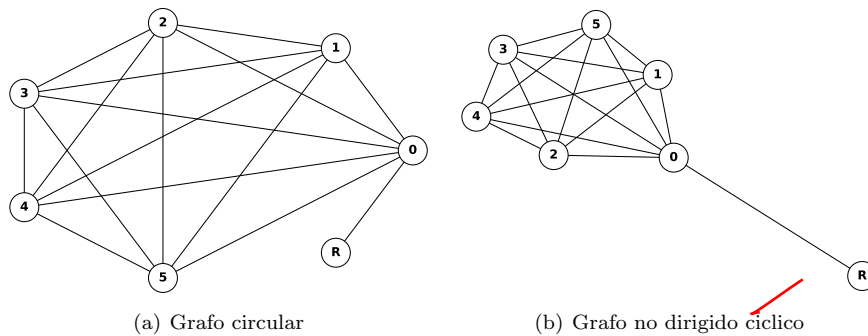


Figura 4: Comparativa entre grafo con acomodo circular vs grafo acomodo por default

Para graficar la red de la figura 4, grafo con red circular se utilizó el siguiente código.

```

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

```

Grafos_layout.py

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()`.

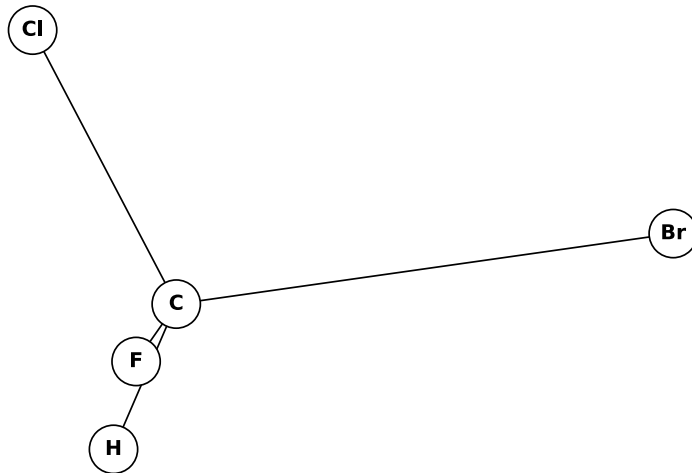


Figura 5: Grafo spectral

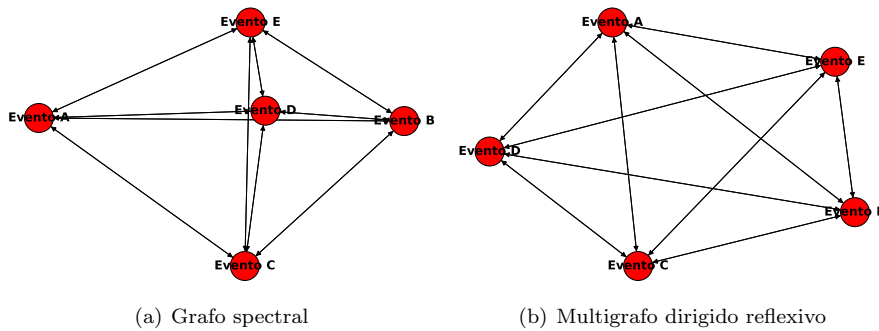


Figura 6: Comparativa entre grafo con acomodo espectral vs grafo con acomodo por default

El código con el que se hizo el grafo de la figura 6 con acomodo espectral es.

```
1 G=nx.MultiDiGraph()
2 G.add_weighted_edges_from([( 'Evento A', 'Evento B',1),
3                               ( 'Evento B', 'Evento A',2),
4                               ( 'Evento A', 'Evento C',3),
5                               ( 'Evento C', 'Evento A',2),
6                               ( 'Evento A', 'Evento D',4),
7                               ( 'Evento D', 'Evento A',1),
8                               ( 'Evento A', 'Evento E',3),
9                               ( 'Evento E', 'Evento A',2),
10                              ( 'Evento B', 'Evento C',4),
11                              ( 'Evento C', 'Evento B',1),
12                              ( 'Evento B', 'Evento D',5),
13                              ( 'Evento D', 'Evento B',1),
14                              ( 'Evento B', 'Evento E',2),
15                              ( 'Evento E', 'Evento B',3),
16                              ( 'Evento C', 'Evento D',4),
17                              ( 'Evento D', 'Evento C',2),
18                              ( 'Evento C', 'Evento E',3),
19                              ( 'Evento E', 'Evento C',1),
20                              ( 'Evento D', 'Evento E',5),
21                              ( 'Evento E', 'Evento D',2)])
22 pos=nx.spectral_layout(G)
23 nx.draw(G, pos, node_size=800, with_labels=True,
24         font_weight="bold", edgecolors="black", node_color='red')
25 plt.savefig('DDoceavo.eps', format='eps', dpi=1000)
```

Grafos.layout.py

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.

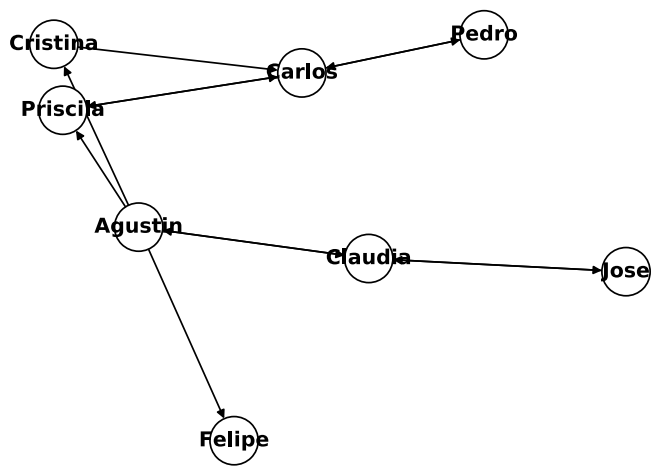


Figura 7: Grafo spring

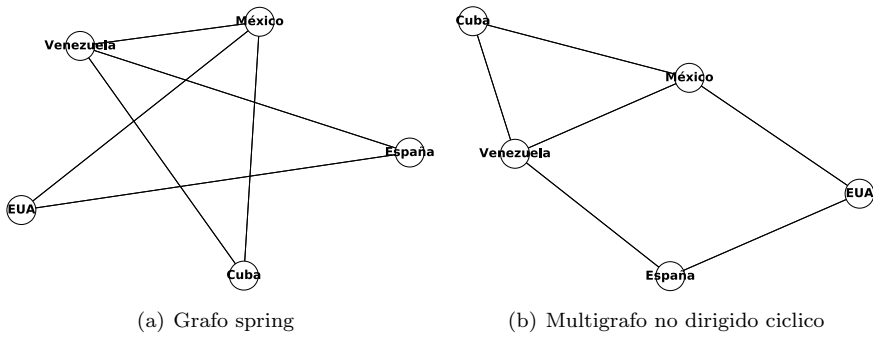


Figura 8: Comparativa entre grafo con acomodo spring vs grafo con acomodo por default

El siguiente código muestra como se hace el grafo de la figura 8.

```

1 H=nx.MultiGraph()
2 H.add_edges_from([('M xico','EUA'),('EUA','M xico'),('EUA','
   Espa a '),
3                   ('Espa a ','EUA'),('M xico','Venezuela'),('
   Venezuela','M xico'),
4                   ('Cuba','M xico'),('M xico','Cuba'),('Cuba','
   Venezuela'),('Venezuela','Cuba'),
5                   ('Venezuela','Espa a '), ('Espa a ','Venezuela')
   ])
6 pos=nx.spring_layout(H)
7 nx.draw(H, pos, node_color="white", node_size=800, with_labels=True
   ,
8         font_weight="bold", edgecolors="black")
9 plt.savefig('OOctavo.eps', format='eps', dpi=1000)

```

Grafos_layout.py

100.1

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()`.

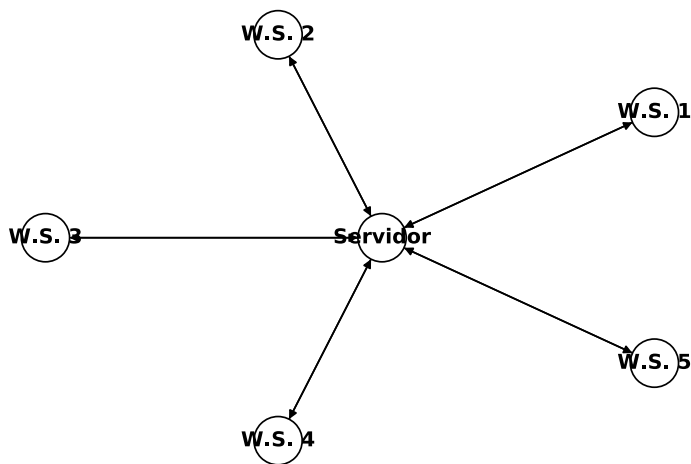


Figura 9: Grafo con diseño Kamada Kawai

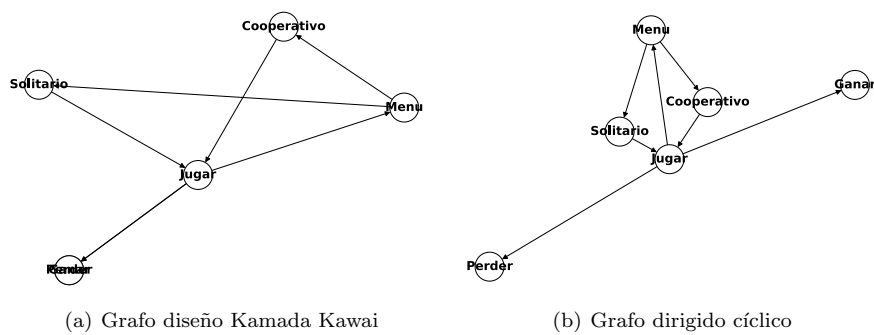


Figura 10: Comparativa entre grafo con acomodo Kamada Kawai **vs** grafo acomodo por default

El código con el que se hizo la red de la figura 10 con el acomodo Kamada Kawai es.

```

1 H=nx.DiGraph()
2 H.add_edges_from([( 'Menu', 'Cooperativo' ),( 'Menu', 'Solitario' ),
3                   ( 'Solitario', 'Jugar' ),( 'Cooperativo', 'Jugar' ),
4                   ( 'Jugar', 'Menu' ),( 'Jugar', 'Ganar' ),
5                   ( 'Jugar', 'Perder' )])
6 pos=nx.kamada_kawai_layout(H)
7 nx.draw(H, pos, node_color="white", node_size=800, with_labels=True
8         , font_weight="bold", edgecolors="black")
9 plt.savefig('QQuinto.eps', format='eps', dpi=1000)

```

Grafos.layout.py

6. Diseño shell

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

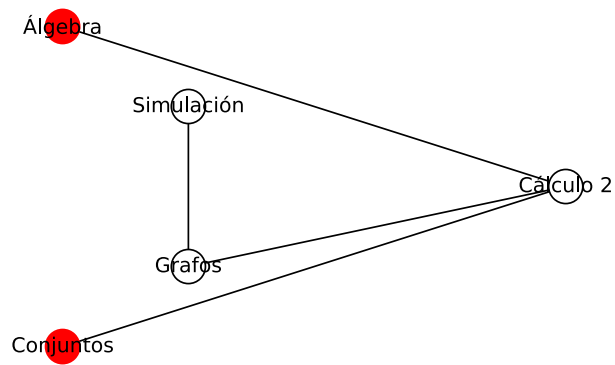
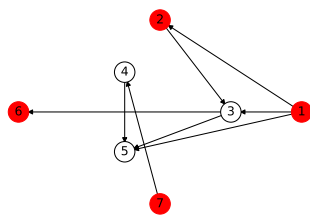
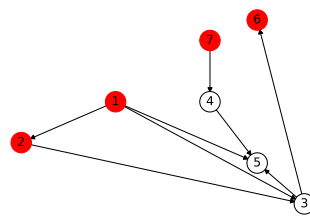


Figura 11: Grafo dirigido reflexivo



(a) Grafo diseño shell



(b) Grafo dirigido cíclico

Figura 12: Comparativa entre grafo con acomodo shell vs grafo acomodo por default

emacs
vim

Para hacer la red de la figura 11 con el acomodo shell es.

```
1 G=nx.Graph()
2 G.add_edges_from([( ' lgebra  ', 'C lculo 2'), ('C lculo 2', 'Grafos')
3     ,
4     ( 'Simulaci n ', 'Grafos'), ('C lculo 2', 'Conjuntos
5     ' )])
6
7 nodos1 = { ' lgebra  ', 'Conjuntos' }
8 nodos2 = { 'C lculo 2', 'Simulaci n ', 'Grafos' }
9
10 shells=[['C lculo 2', 'Simulaci n ', 'Grafos'], ['C lculo 1', '
11     lgebra  ',
12     'Conjuntos' ]]
13
14 pos=nx.shell_layout(G, nlist=shells)
15
16 nx.draw_networkx_nodes(G, pos, nodelist=nodos1, node_size=400,
17     node_color='r', node_shape='o')
18 nx.draw_networkx_nodes(G, pos, nodelist=nodos2, node_size=400,
19     node_color='w', node_shape='o',
20     edgecolors="k")
21
22 nx.draw_networkx_edges(G, pos)
23 nx.draw_networkx_labels(G, pos)
24 plt.axis('off')
25 plt.savefig('TTercero.eps', format='eps', dpi=1000)
```

Grafos_layout.py

Referencias

- [1] A. C. Barrero, G. W. de García, and R. M. M. Parra, *Introducción a la Teoría de Grafos*. ELIZCOM SAS, 2010.
- [2] N. Developers, “Networkx documentation,” 2012. *unc*
- [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.

{ → }

Diseño de los grafos

1484166

18 de marzo de 2019

Introducción

La importancia de los grafos es que con ellos se puede modelar ciertos 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 5 algoritmos de **Networkx** y se mide su tiempo de ejecución.

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.

Algoritmos

Los algoritmos implementados fueron los siguientes.

`dfs_tree()`

Este algoritmo recibe como entrada 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 mas grande a la mas chica.

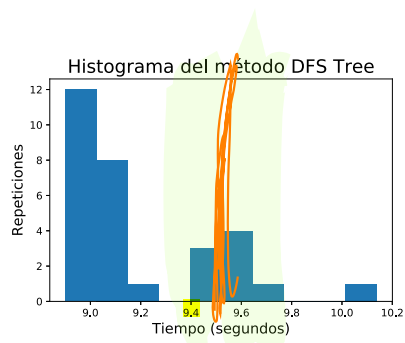
`treewidth_min_degree()`

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

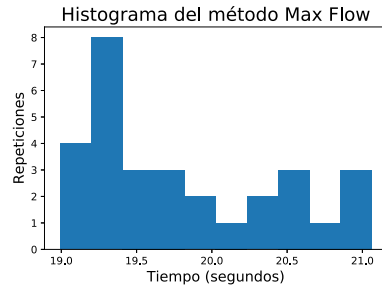
Metodología

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.

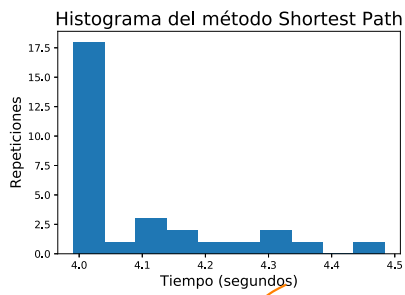
Con esa información se obtuvo los 5 histogramas de la figura 1.



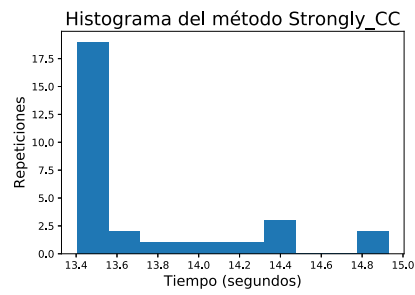
(a) Búsqueda a profundidad



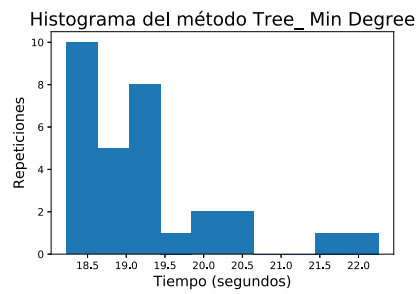
(b) Flujo máximo



(c) Caminos mas cortos



(d) Componentes fuertemente conectados



(e) Descomposición del ancho de árbol

Figura 1: Histogramas de tiempos promedios por algoritmo

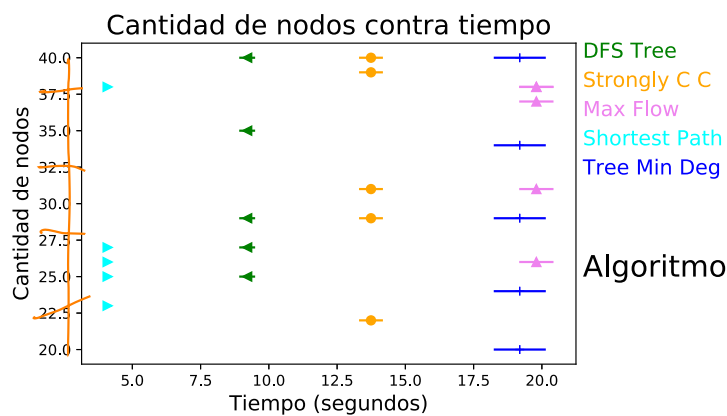
Una vez guardada la información en una variable tipo diccionario, el código que da paso a los histogramas es el siguiente

```
1 for i in ('DFS Tree', 'Max Flow', 'Shortest Path', 'Strongly_CC', 'Tree_Min_Degree'):
2     plt.xlabel('Tiempo (segundos)', size=14)
3     plt.ylabel('Repeticiones', size=14)
4     plt.title('Histograma del m todo '+i, size=18)
5     plt.hist(dicrio[i]['medias_repeticiones'])
6     plt.savefig(i+'.eps', format='eps', dpi=1000)
7     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 2.



¿grafos?

Figura 2. Error bar del tiempo promedio de ejecución vs la cantidad de nodos.

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 3.

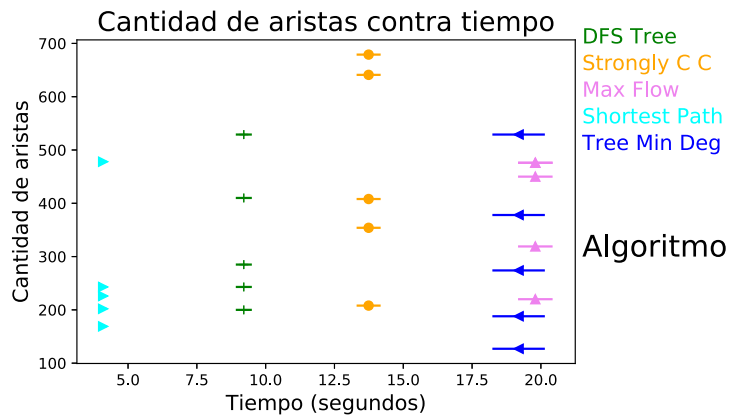


Figura 3: Error bar del tiempo promedio de ejecución vs la cantidad de aristas.

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 3 se hace con el siguiente código.

```

1 y=[]
2 for i in ('DFS Tree', 'Max Flow', 'Shortest Path', 'Strongly_CC', '
   Tree_Min_Degree'):
3     for j in (1,2,3,4,5):
4         y.append(dicrio[i][j]['aristas'])
5 for i in (1,2,3,4,5):
6     if i==1:
7         x=[dicrio['DFS Tree']['media_algoritmo']]
8         xe=[dicrio['DFS Tree']['std_algoritmo']]
9         plt.errorbar(5*x,y[0:5], xerr=5*xe, fmt='+',color='g',alpha
   =0.5)
10    if i==2:
11        x=[dicrio['Max Flow']['media_algoritmo']]
12        xe=[dicrio['Max Flow']['std_algoritmo']]
13        plt.errorbar(5*x,y[5:10], xerr=5*xe, fmt='^',color='violet',
   ,alpha=0.5)
14    if i==3:
15        x=[dicrio['Shortest Path']['media_algoritmo']]
16        xe=[dicrio['Shortest Path']['std_algoritmo']]
17        plt.errorbar(5*x, y[10:15], xerr=5*xe, fmt='>',color='aqua',
   alpha=0.5)
18    if i==4:
19        x=[dicrio['Strongly_CC']['media_algoritmo']]
20        xe=[dicrio['Strongly_CC']['std_algoritmo']]
21        plt.errorbar(5*x,y[15:20], xerr=5*xe, fmt='o',color='orange',
   ,alpha=0.5)
22    if i==5:
23        x=[dicrio['Tree_Min_Degree']['media_algoritmo']]
24        xe=[dicrio['Tree_Min_Degree']['std_algoritmo']]
25        plt.errorbar(5*x,y[20:25], xerr=5*xe, fmt='<',color='blue',
   alpha=0.5)
26 plt.xlabel('Tiempo (segundos)', size=14)
27 plt.ylabel('Cantidad de aristas', size=14)
28 plt.title('Cantidad de aristas contra tiempo',size=18)
29 plt.text(21.5, 700, r'DFS Tree',color='g',size=14)
30 plt.text(21.5, 650, r'Strongly C C',color='orange',size=14)
31 plt.text(21.5, 600, r'Max Flow',color='violet',size=14)
32 plt.text(21.5, 550, r'Shortest Path',color='aqua',size=14)
33 plt.text(21.5, 500, r'Tree Min Deg',color='blue',size=14)
34 plt.text(21.5, 300, r'Algoritmo',color='black',size=20)
35 plt.savefig('scatter2.eps', format='eps', bbox_inches='tight', dpi
   =1000)

```

Algoritmos_grafos.py

Referencias

[1] N. Developers, "Networkx documentation," 2012. *UNC?* *¿vesish?*

2. Tarea tres corregida

Introducción

La importancia de los grafos es que con ellos se puede modelar ciertos 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.

2.1. 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 LPDDR3

2.2. Algoritmos

Los algoritmos a implementar son los siguientes.

`dfs_tree()`

Este algoritmo recibe como entrada 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_paths()`

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()`

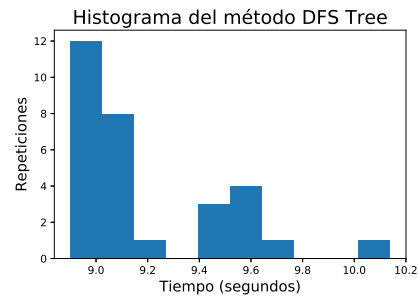
Devuelve una descomposición del ancho de árbol usando la heurística de Grado Mínimo.

2.3. Metodología

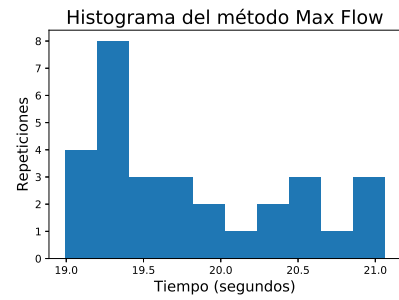
Lo que se hizo para cada algoritmo es correr cinco grafos, cada uno con diferente cantidad de nodos y diferente cantidad de aristas, durante un tiempo de cinco segundos, a estas réplicas se les obtuvo la media, y lo mismo se hizo durante treinta veces, obteniendo así treinta datos de la media.

Con esa información se obtuvo los 5 histogramas de la figura 13.

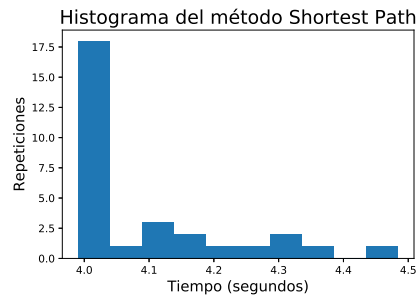
Una vez guardada la información en una variable tipo diccionario, el código que da paso a los histogramas es el siguiente de donde se puede apreciar que no se aproximan a una gráfica normal.



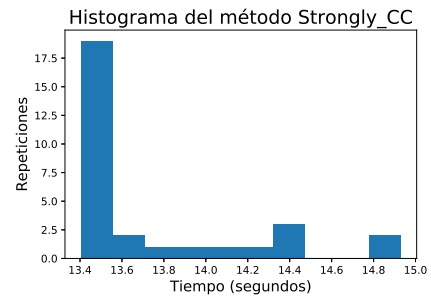
(a) Búsqueda a profundidad



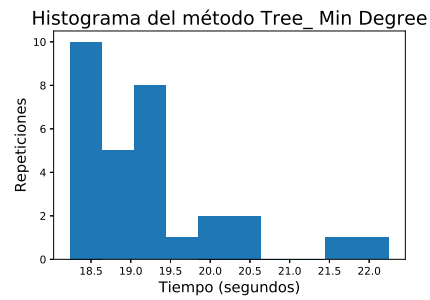
(b) Flujo máximo



(c) Caminos mas cortos



(d) Componentes fuertemente conectados



(e) Descomposición del ancho de árbol

Figura 13: Histogramas de tiempos promedios por algoritmo


```

1 for i in ('DFS Tree', 'Max Flow', 'Shortest Path', 'Strongly_CC', 'Tree_ Min Degree'):
2     plt.xlabel('Tiempo (segundos)', size=14)
3     plt.ylabel('Repeticiones', size=14)
4     plt.title('Histograma del m todo '+i, size=18)
5     plt.hist(dicrio[i]['medias-repeticiones'])
6     plt.savefig(i+'.eps', format='eps', dpi=1000)
7     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 17.

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 15.

2.4. 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 15 se hace con el siguiente código.

```

1 y=[]
2 for i in ('DFS Tree', 'Max Flow', 'Shortest Path', 'Strongly_CC', 'Tree_ Min Degree'):
3     for j in (1,2,3,4,5):
4         y.append(dicrio[i][j]['aristas'])
5 for i in (1,2,3,4,5):
6     if i==1:
7         x=[dicrio['DFS Tree']['media_algoritmo']]
8         xe=[dicrio['DFS Tree']['std_algoritmo']]
9         plt.errorbar(5*x,y[0:5], xerr=5*xe, fmt='+',color='g',alpha=0.5)
10    if i==2:
11        x=[dicrio['Max Flow']['media_algoritmo']]
12        xe=[dicrio['Max Flow']['std_algoritmo']]
13        plt.errorbar(5*x,y[5:10], xerr=5*xe, fmt='^',color='violet',alpha=0.5)
14    if i==3:
15        x=[dicrio['Shortest Path']['media_algoritmo']]
16        xe=[dicrio['Shortest Path']['std_algoritmo']]
17        plt.errorbar(5*x, y[10:15], xerr=5*xe, fmt='>',color='aqua',alpha=0.5)
18    if i==4:
19        x=[dicrio['Strongly_CC']['media_algoritmo']]
20        xe=[dicrio['Strongly_CC']['std_algoritmo']]
21        plt.errorbar(5*x,y[15:20], xerr=5*xe, fmt='o',color='orange',alpha=0.5)
22    if i==5:
23        x=[dicrio['Tree_ Min Degree']['media_algoritmo']]
24        xe=[dicrio['Tree_ Min Degree']['std_algoritmo']]
25        plt.errorbar(5*x,y[20:25], xerr=5*xe, fmt='<',color='blue',alpha=0.5)
26 plt.xlabel('Tiempo (segundos)', size=14)
27 plt.ylabel('Cantidad de aristas', size=14)
28 plt.title('Cantidad de aristas contra tiempo',size=18)
29 plt.text(21.5, 700, r'DFS Tree',color='g',size=14)
30 plt.text(21.5, 650, r'Strongly C C',color='orange',size=14)
31 plt.text(21.5, 600, r'Max Flow',color='violet',size=14)
32 plt.text(21.5, 550, r'Shortest Path',color='aqua',size=14)
33 plt.text(21.5, 500, r'Tree Min Deg',color='blue',size=14)
34 plt.text(21.5, 300, r'Algoritmo',color='black',size=20)
35 plt.savefig('scater2.eps', format='eps', bbox_inches='tight', dpi=1000)

```

Algoritmos_grafos.py

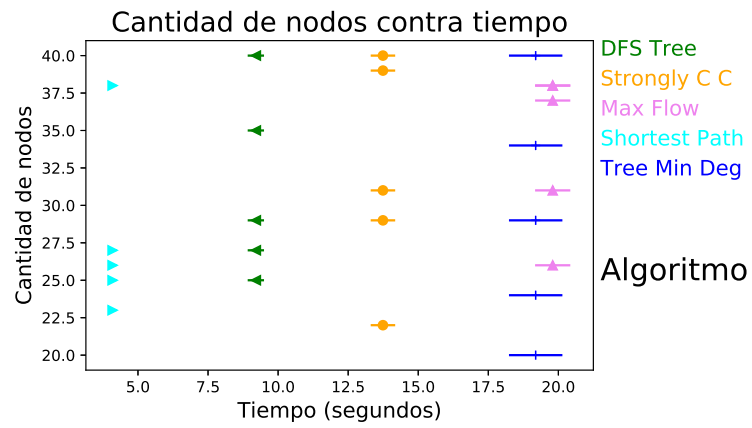


Figura 14: Barras de error del tiempo promedio de ejecución vs la cantidad de nodos.

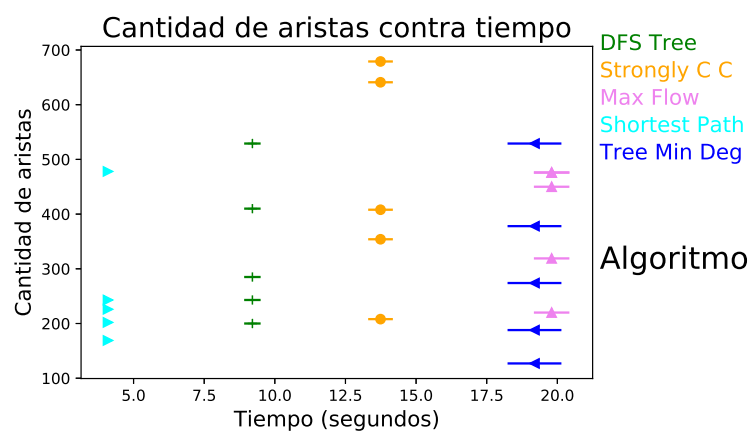


Figura 15: Barras de error del tiempo promedio de ejecución vs la cantidad de aristas.

Complejidad asintótica

4166

2 de abril de 2019



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 el 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**.

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.

2. Algoritmos

Los algoritmos a implementar son los siguientes.

`maximum_flow_value()`

Este algoritmo recibe como entrada un grafo no orientado con capacidad en sus aristas junto con un par de nodos, el nodo *fuelle* 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 *fuelle* 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.

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 devuelve 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.

`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.

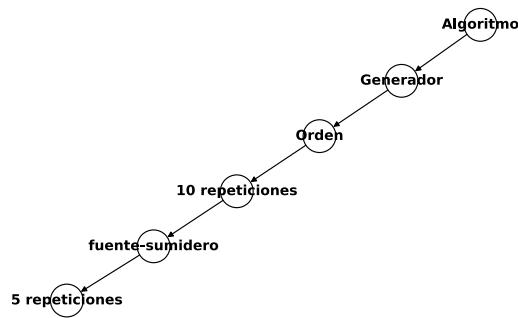


Figura 1: Diagrama de flujo aplicada en la metodología.

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 1 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.

```

1 def correr(generator, algorithm, cardinalidad, l):
2     if generator==0:
3         G=AddEdges(nx.dense_gnm_random_graph(cardinalidad, int
4         ((0.5*size*(size-1))-(1-((density+1)/10))*(0.5*size*(size-1)))
5         ))
6         nodes=RandNodes(cardinalidad-1,G)
7         data[contador,3]=nx.density(G)
8         if algorithm==0:
9             nx.maximum_flow_value(G,nodes[l][0],nodes[l][1])
10        elif algorithm==1:
11            nx.algorithms.flow.edmonds_karp(G,nodes[l][0],nodes[l][1])
12        elif algorithm==2:
13            nx.algorithms.flow.boykov_kolmogorov(G,nodes[l][0],nodes[l][1])
14        elif generator==1:
15            F=AddEdges(nx.generators.random_graphs.
16            barabasi_albert_graph(cardinalidad, random.randint(1,
17            cardinalidad-1)))
18            nodes=RandNodes(cardinalidad-1,F)
19            data[contador,3]=nx.density(F)
20            if algorithm==0:

```

0: nx.m... alg Ed
1: n...

```

17     nx.maximum_flow_value(F, nodes[1][0], nodes[1][1])
18     elif algorithm==1:
19         nx.algorithms.flow.edmonds_karp(F, nodes[1][0], nodes[1
20 ][1])
21     elif algorithm==2:
22         nx.algorithms.flow.boykov_kolmogorov(F, nodes[1][0],
23 nodes[1][1])
24     elif generator==2:
25         J=AddEdges(nx.generators.duplication.
26 duplication_divergence_graph(cardinalidad, random.random(),
27 seed=None))
28         nodes=RandNodes(cardinalidad-1,J)
29         data[contador,3]=nx.density(J)
30         if algorithm==0: desla J(J, ...)
31             nx.maximum_flow_value(J, nodes[1][0], nodes[1][1])
32         elif algorithm==1:
33             nx.algorithms.flow.edmonds_karp(J, nodes[1][0], nodes[1
34 ][1])
35         elif algorithm==2:
36             nx.algorithms.flow.boykov_kolmogorov(J, nodes[1][0],
37 nodes[1][1])
38
39 data=np.arange(1800*5, dtype=float).reshape(1800,5)
40 contador=0
41 for generador in range(3):
42     for algoritmo in range(3):
43         for orden in (16,64,256,1024):
44             size=orden
45             for densidad in range(10):
46                 for repeticion in range(5):
47                     tiempo_inicial=time()
48                     correr(generador, algoritmo, size, repeticion)
49                     tiempo_final=time()
50                     tiempo_ejecucion=tiempo_final-tiempo_inicial
51                     data[contador,0]=generador
52                     data[contador,1]=algoritmo
53                     data[contador,2]=orden
54                     data[contador,4]=tiempo_ejecucion
55                     contador+=1

```

Complejidad_experimental.py

5. Resultados

Con esa información se grafica la figura 2 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 3 es que el orden y la densidad se relaciona directamente proporcional con el tiempo de ejecución.

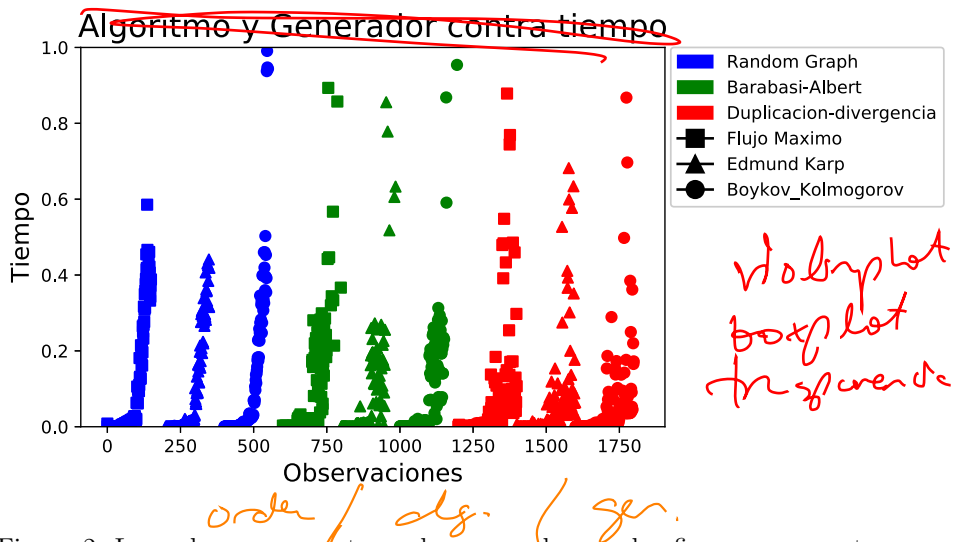


Figura 2: Los colores representan a los generadores y las figuras representan algoritmos.

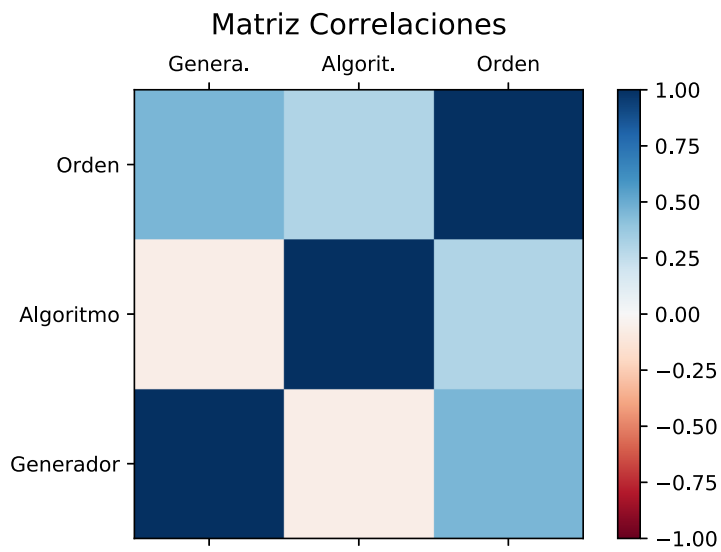


Figura 3: Correlaciones entre los datos cuantitativos.

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

	sum_sq	...	PR>F	
Generador	6.731003	...	1.081764e-01	0.108
Algoritmo	41.698816	...	6.612653e-05	
Orden	2999.602915	...	4.550752e-195	
Densidad	643.580915	...	3.316542e-52	
Generador:Algoritmo	22.285871	...	3.500409e-03	
Algoritmo:Orden	122.376829	...	1.001579e-11	
Orden:Densidad	1701.601229	...	5.875774e-123	
Generador:Orden	7.450435	...	9.022655e-02	0.092
Generador:Densidad	31.800908	...	4.868091e-04	<0.001
Algoritmo:Densidad	168.318921	...	1.671812e-15	
Generador:Algoritmo:Orden	49.306891	...	1.429967e-05	
Generador:Algoritmo:Densidad	54.606493	...	4.985981e-06	
Generador:Densidad:Orden	96.210373	...	1.492816e-09	
Algoritmo:Densidad:Orden	470.111502	...	3.064382e-39	
Generador:Algoritmo:Densidad:Orden	154.603625	...	2.217023e-14	
Residual	4650.515320	...		

[16 rows x 4 columns]

Referencias

- [1] NetworkX Developers, "Networkx documentation," 2012.
- [2] A. C. Barrero, G. W. de García, and R. M. M. Parra, *Introducción a la Teoría de Grafos*. ELIZCOM SAS, 2010.
- [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.

3. 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 el 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`.

3.1. 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.

3.2. Algoritmos

Los algoritmos a implementar son los siguientes.

`maximum_flow_value()`

Este algoritmo recibe como entrada un grafo no orientado con capacidad en sus aristas junto con un par de nodos, el nodo *fuelle* 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.

3.3. 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 devuelve 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.

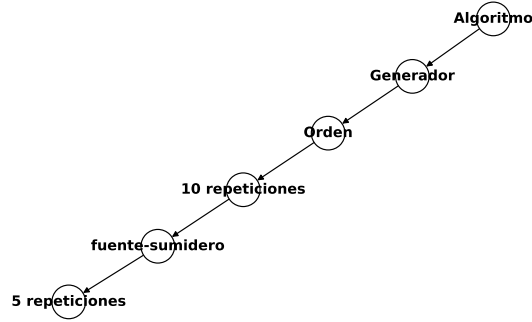


Figura 16: Diagrama de flujo aplicada en la metodología.

`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.

3.4. 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 21 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.

```

1 def correr(generator, algorithm, crdl, 1):
2     if generator==0:
3         G=AddEdges(nx.dense_gnm_random_graph(crdl,
4             int((0.5*sz*(sz-1))-(1-((dnsd+1)/10))*(0.5*sz*(sz-1))))))
5         nds=RandNodes(crdl-1,G)
6         data[contador,3]=nx.density(G)
7         if algorithm==0:
8             nx.maximum_flow_value(G,nds[1][0],nds[1][1])
9         elif algorithm==1:
10            nx.algorithms.flow.edmonds_karp(G,nds[1][0],nds[1][1])
11        elif algorithm==2:
12            nx.algorithms.flow.boykov_kolmogorov(G,nds[1][0],nds[1][1])
13    elif generator==1:
14        F=AddEdges(nx.generators.random_graphs.barabasi_albert_graph(crdl,
15            random.randint(1,crdl-1)))
16        nds=RandNodes(crdl-1,F)
17        data[contador,3]=nx.density(F)
18        if algorithm==0:
19            nx.maximum_flow_value(F,nds[1][0],nds[1][1])
20        elif algorithm==1:
21            nx.algorithms.flow.edmonds_karp(F,nds[1][0],nds[1][1])
22        elif algorithm==2:
23            nx.algorithms.flow.boykov_kolmogorov(F,nds[1][0],nds[1][1])
24    elif generator==2:
25        J=AddEdges(nx.generators.duplication.duplication_divergence_graph(crdl,
26            random.random(), seed=None))
27        nds=RandNodes(crdl-1,J)
28        data[contador,3]=nx.density(J)
29        if algorithm==0:
30            nx.maximum_flow_value(J,nds[1][0],nds[1][1])
31        elif algorithm==1:
32            nx.algorithms.flow.edmonds_karp(J,nds[1][0],nds[1][1])
33        elif algorithm==2:
34            nx.algorithms.flow.boykov_kolmogorov(J,nds[1][0],nds[1][1])
35
36 data=np.arange(1800*5, dtype=float).reshape(1800,5)
37 contador=0
38 for generador in range(3):
39     for algoritmo in range(3):
40         for orden in (16,64,256,1024):
41             sz=orden
42             for dnsd in range(10):
43                 for repeticion in range(5):
44                     tiempo_inicial=time()
45                     correr(generator, algoritmo, sz, repeticion)
46                     tiempo_final=time()
47                     tiempo_ejecucion=tiempo_final-tiempo_inicial
48                     data[contador,0]=generador
49                     data[contador,1]=algoritmo
50                     data[contador,2]=orden
51                     data[contador,4]=tiempo_ejecucion
52                     contador+=1

```

Complejidad.experimental.py

3.5. Resultados

Con esa información se grafica la figura 17 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 36 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

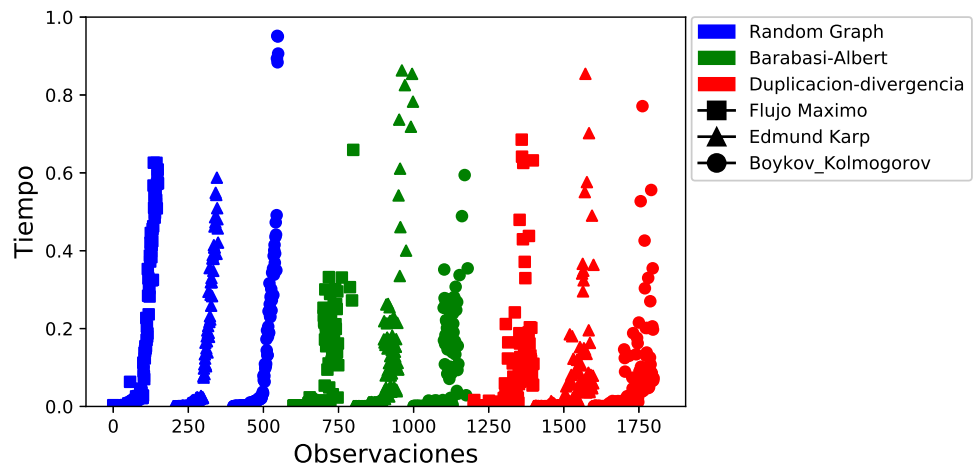


Figura 17: Los colores representan a los generadores y las figuras representan algoritmos.

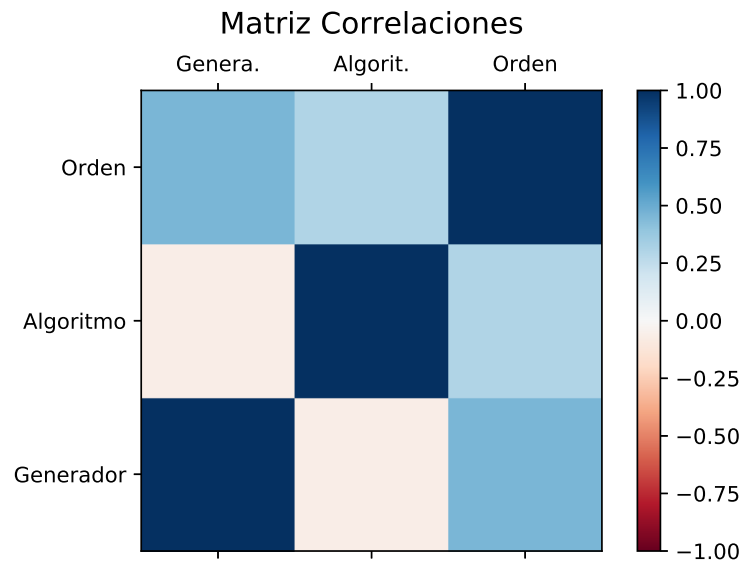


Figura 18: Correlaciones entre los datos cuantitativos.

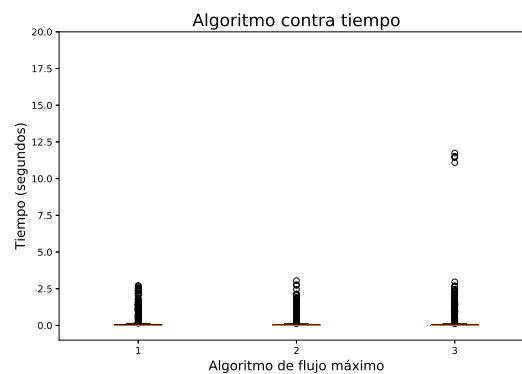


Figura 19: Diagrama de caja de algoritmo.

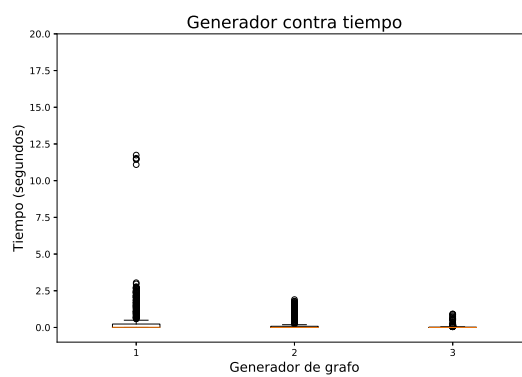


Figura 20: Diagrama de caja del generador.

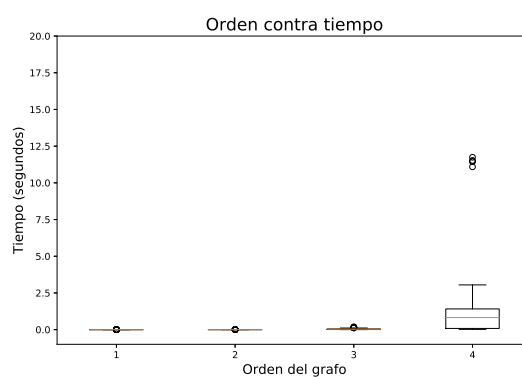


Figura 21: Diagrama de caja del orden.

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		



Caracterización estructural de instancias~~X~~

4166

30 de abril de 2019

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 **fuelle** 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**.

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.

1. 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,

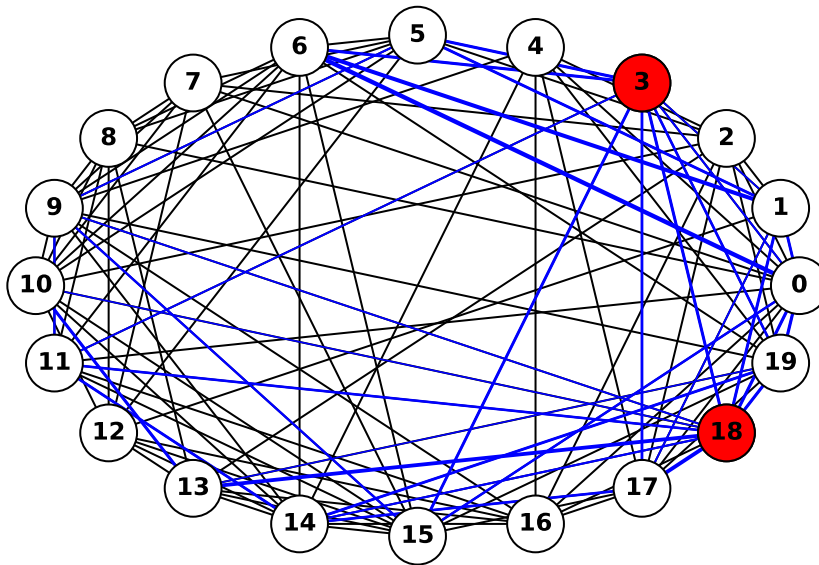


Figura 1: Grafo con rutas de flujo máximo.

todos los nodos son accesibles desde otro nodo.

En las figuras 1, 2, 3, 4, 5 se muestran grafos que son hechos con este generador.

Algoritmos

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

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 ~~fuentes~~ 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.

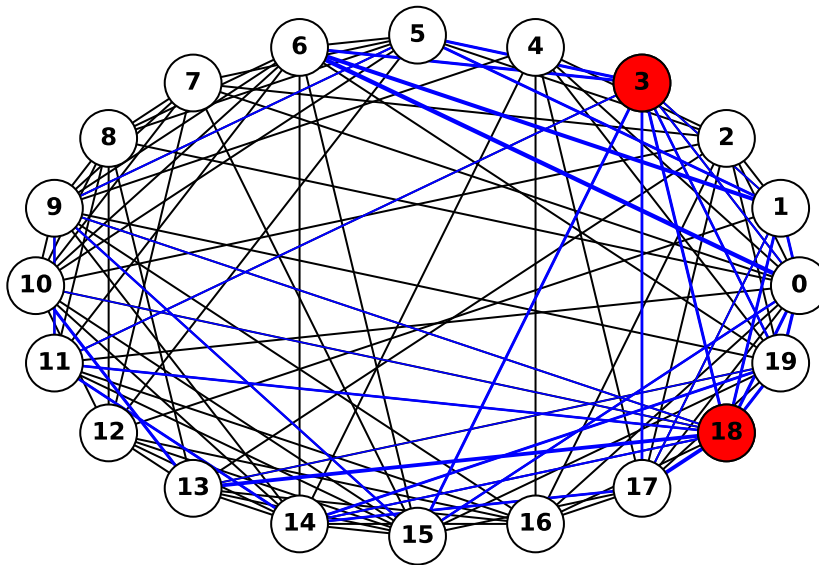


Figura 2: Grafo con rutas de flujo máximo.

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 qué 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.

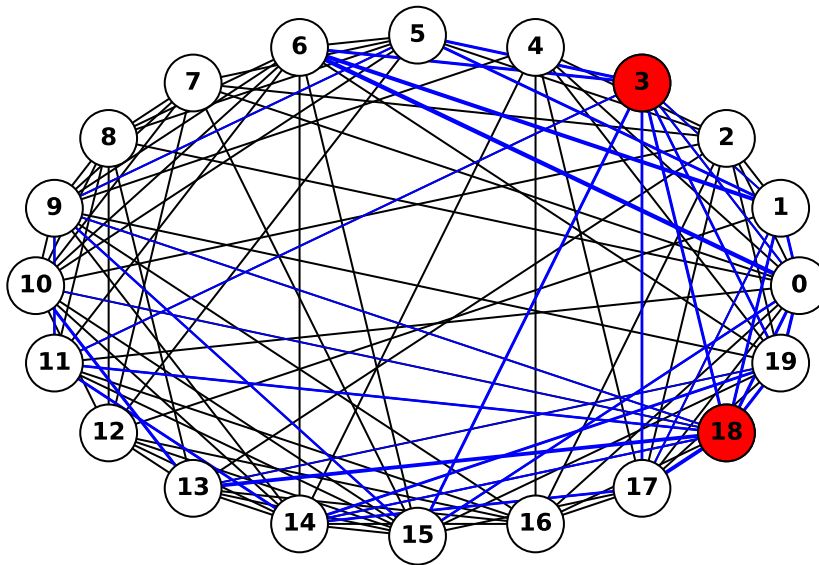


Figura 3: Grafo con rutas de flujo máximo.

`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 ~~nod~~os en el gráfico de acuerdo a las características de los nodos con quien esta conectado.

2. 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 ~~es~~ estos nodos.

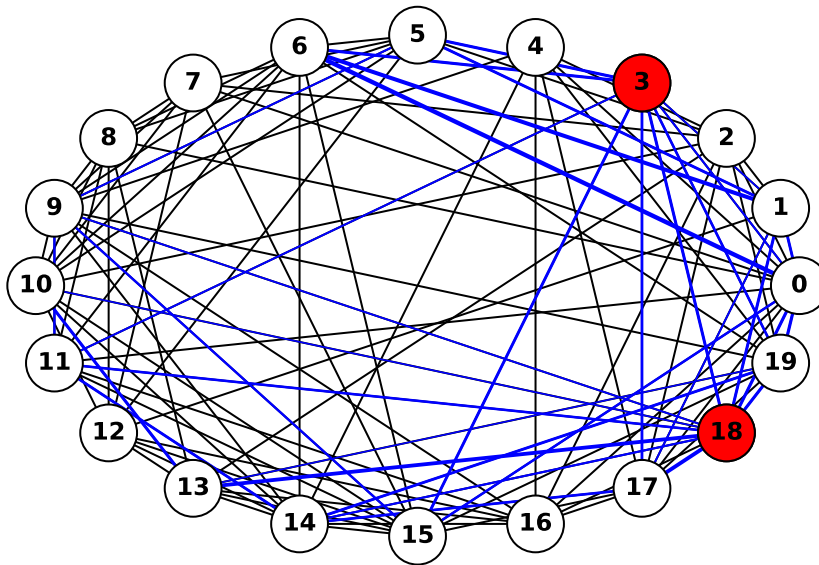


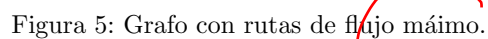
Figura 4: Grafo con rutas de flujo máximo.

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

```

1 #-----Experimentacion-----
2 datos=np.arange(sum(orden)*15, dtype=float).reshape(sum(orden),15)
3 fila=0
4 for i in range(len(orden)):
5     H=nx.watts_strogatz_graph(orden[i], int(orden[i]/2) , 0.33 ,
6         seed=None)
7     initial=0
8     final=0
9     lista=[]
10    lista[:]=H.edges
11    width=np.arange(len(lista)*1,dtype=float).reshape(len(lista),1)
12    for r in range(len(lista)):
13        R=np.random.normal(loc=20, scale=5.0, size=None)
14        width[r]=R
15        H.add_edge(lista[r][0], lista[r][1], capacity=R)
16    for w in range(orden[i]):
17        initial=random.randint(0,round(len(H.nodes)/2))
18        final=random.randint(initial, len(H.nodes)-2)
19        while initial==final:
20            initial=random.randint(0,round(len(H.nodes)/2))
21            final=random.randint(initial, len(H.nodes)-2)
22        tiempo_inicial=time()
23        T=nx.maximum_flow(H, initial, final)
24        tiempo_final=time()
25        datos[fila,0]=T[0]

```



caracterizacion.py

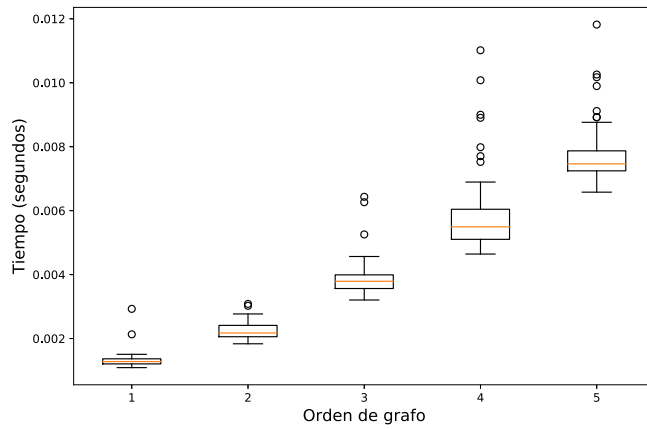


Figura 6: Diagrama de caja de tiempo.

3. Resultados

Se elaboró unos diagramas de caja para ver el comportamiento de las características de los nodos fuente y sumidero 6, 7, 8, 9, 10, 11, 12, 13, 14. Los valores del eje de las abscisas representan el orden de los grafos.

Con esa información se hace una matriz de correlación 15 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 15 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

	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

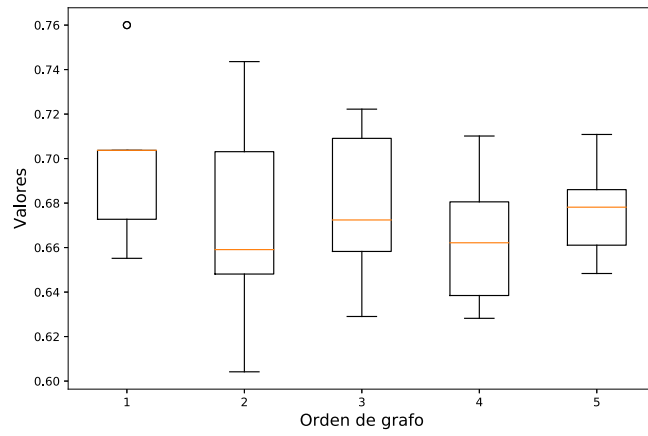


Figura 7: Diagrama de caja de centralidad de nodos fuentes.

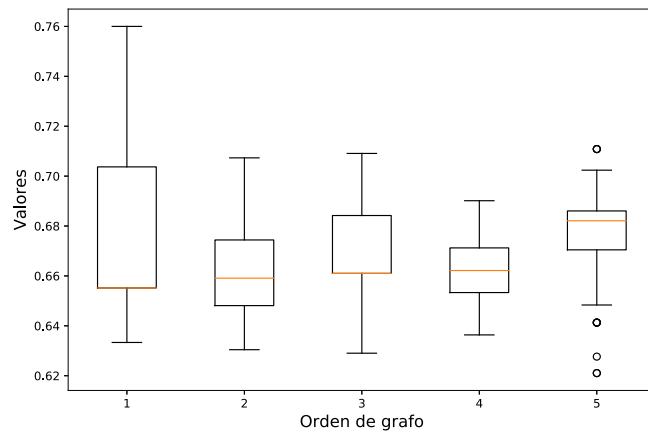


Figura 8: Diagrama de caja de centralidad de nodos sumidero.

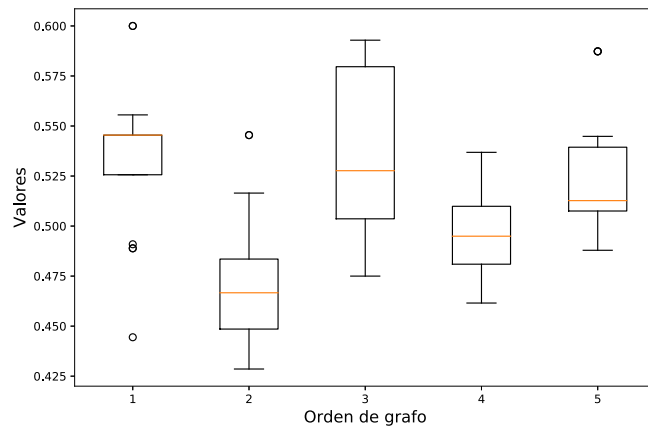


Figura 9: Diagrama de caja de clustering de nodos fuentes.

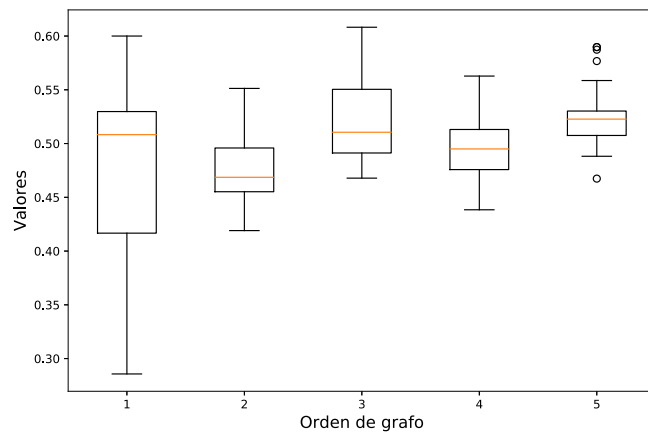


Figura 10: Diagrama de caja de clustering de nodos sumidero.

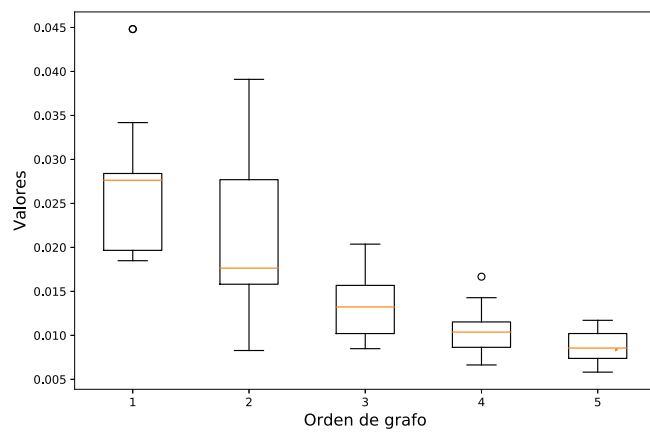


Figura 11: Diagrama de caja de load centrality de nodos fuentes.

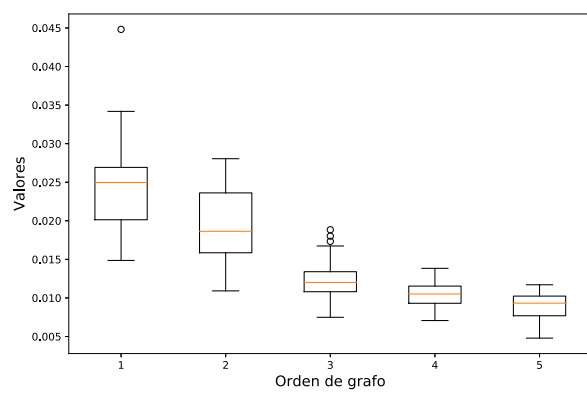


Figura 12: Diagrama de caja de load centrality de nodos sumidero.

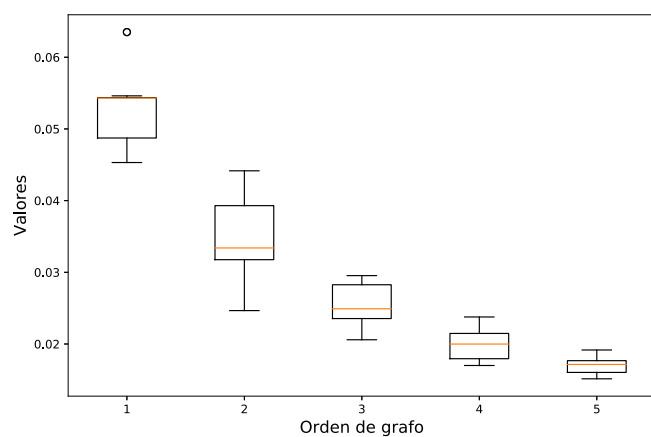


Figura 13: Diagrama de caja de pagerank de nodos fuentes.

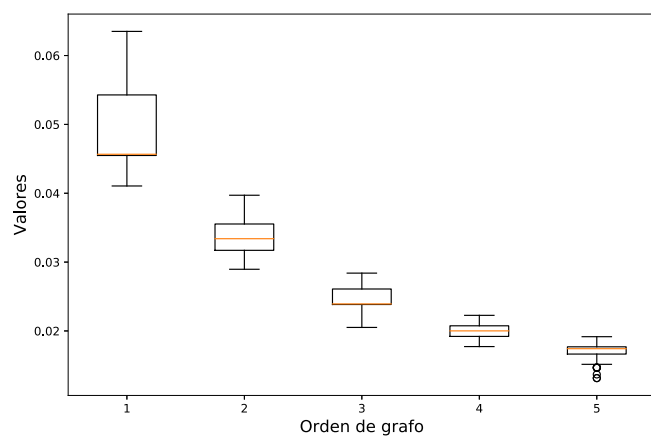


Figura 14: Diagrama de caja de load pagerank de nodos sumidero.

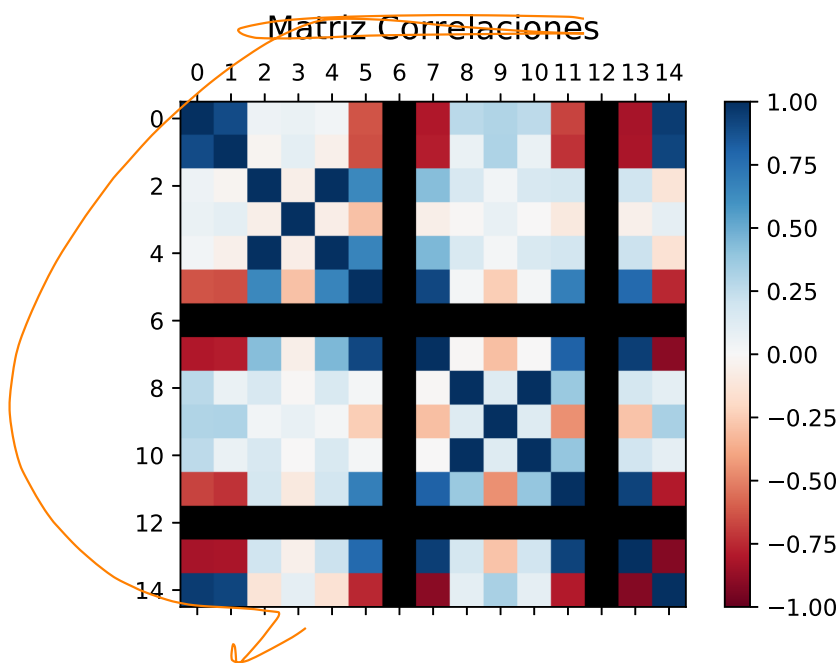


Figura 15: Correlaciones de las características de los nodos fuente y sumidero.

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

Referencias

- [1] NetworkX Developers, "Networkx documentation," 2012.
- [2] J. L. Molina, "La ciencia de las redes," *Apuntes de Ciencia y Tecnología*, vol. 11, no. 1, pp. 36–42, 2004.
- [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.

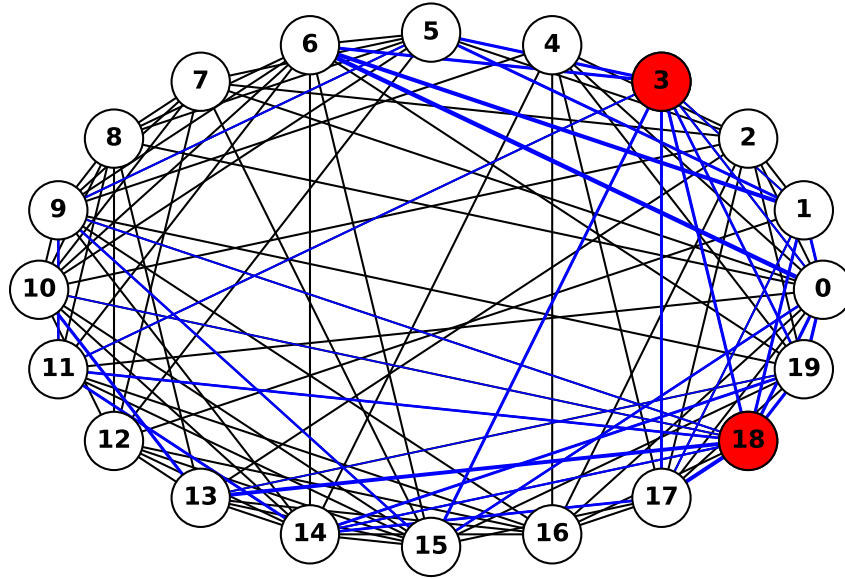


Figura 22: Grafo con rutas de flujo máximo.

4. 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 **fuelle** y **sumidero**, es saber cual es el valor del flujo máximo que puede transportar desde el fuele al sumidero. Otros aspectos importantes es ver como las características del nodo fuele 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**.

4.1. 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.

4.2. 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 22, 38, 39, 40, 41 se muestran grafos que son hechos con este generador.

4.3. Algoritmos

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

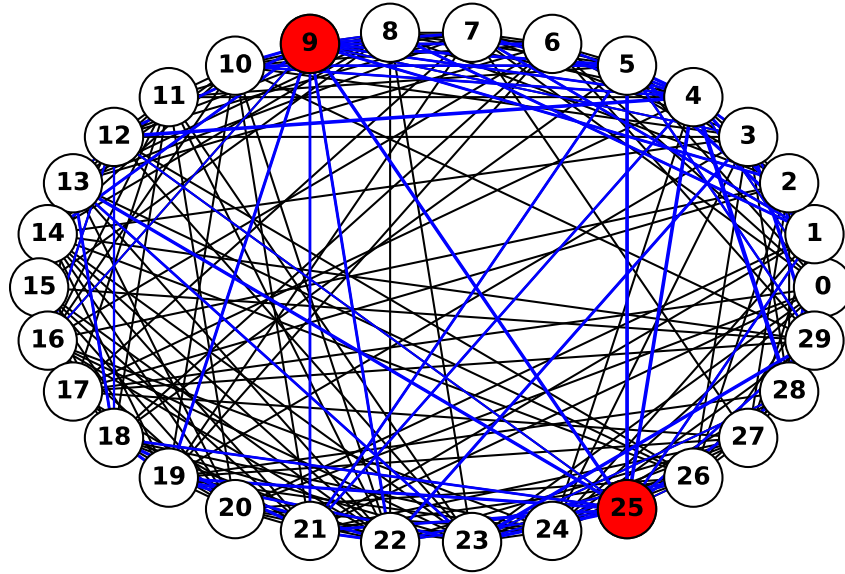


Figura 23: Grafo con rutas de flujo máximo.

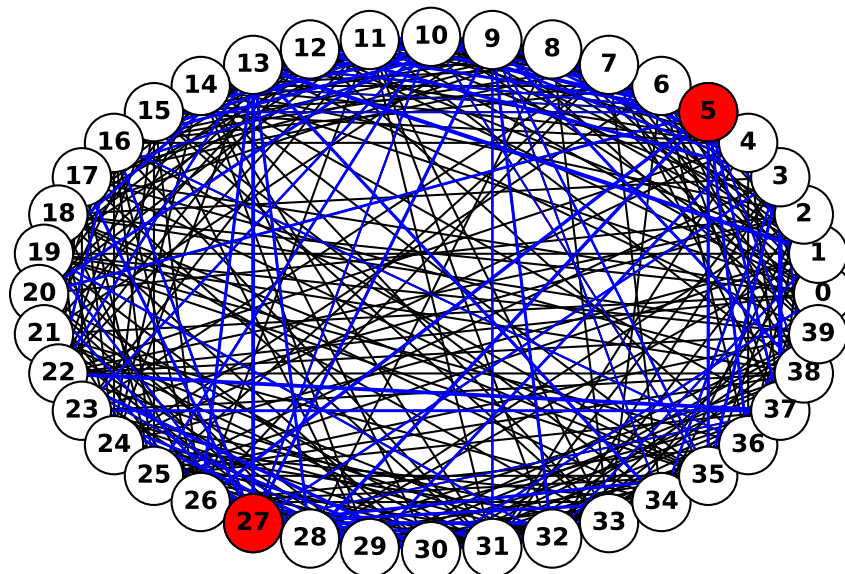


Figura 24: Grafo con rutas de flujo máximo.

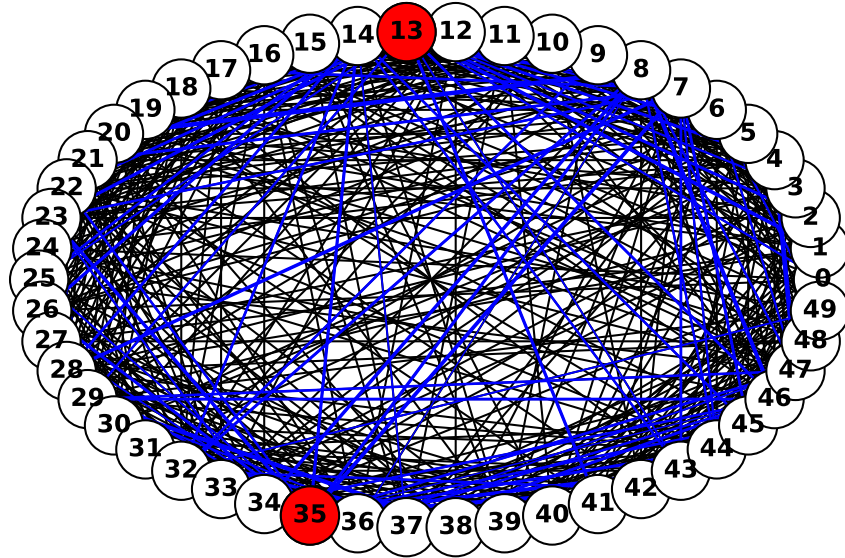


Figura 25: Grafo con rutas de flujo máximo.

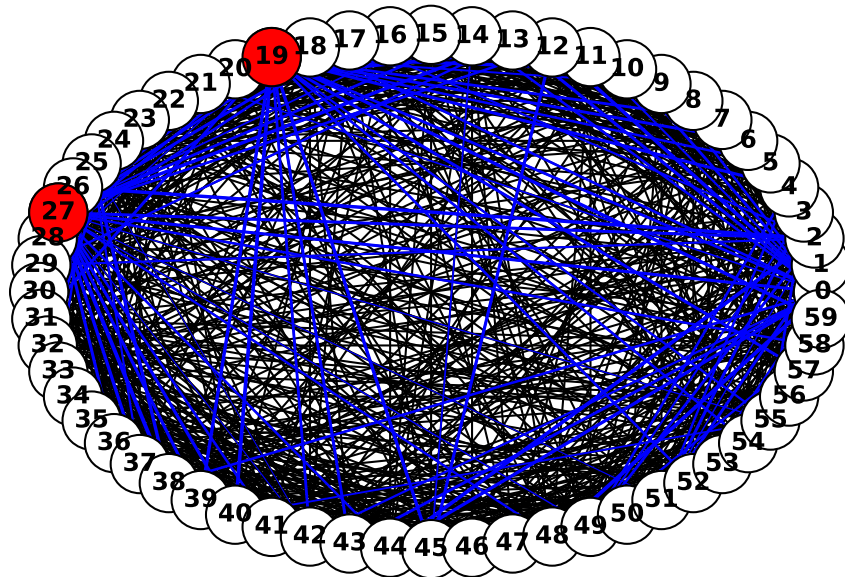


Figura 26: Grafo con rutas de flujo máximo.

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.

4.4. 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.

```

1 #-----Experimentacion-----
2 datos=np.arange(sum(orden)*15, dtype=float).reshape(sum(orden),15)
3 fila=0
4 for i in range(len(orden)):
5     H=nx.watts_strogatz_graph(orden[i], int(orden[i]/2) , 0.33 , seed=None)
6     initial=0
7     final=0
8     lista=[]
9     lista[:]=H.edges
10    width=np.arange(len(lista)*1,dtype=float).reshape(len(lista),1)
11    for r in range(len(lista)):
12        R=np.random.normal(loc=20, scale=5.0, size=None)
13        width[r]=R
14        H.add_edge(lista[r][0], lista[r][1], capacity=R)
15    for w in range(orden[i]):
16        initial=random.randint(0,round(len(H.nodes)/2))
17        final=random.randint(initial, len(H.nodes)-2)
18        while initial==final:
19            initial=random.randint(0,round(len(H.nodes)/2))
20            final=random.randint(initial, len(H.nodes)-2)
21        tiempo_inicial=time()
22        T=nx.maximum_flow(H, initial, final)
23        tiempo_final=time()
24        datos[fila,0]=T[0]
25        datos[fila,1]=tiempo_final-tiempo_inicial
26 #-----Info Fuente-----
27 datos[fila,2]=nx.degree_centrality(H)[initial]
28 datos[fila,3]=nx.clustering(H, nodes=initial)
29 datos[fila,4]=nx.closeness_centrality(H, u=initial)
30 datos[fila,5]=nx.load_centrality(H, v=initial)
31 datos[fila,6]=nx.eccentricity(H, v=initial)
32 datos[fila,7]=nx.pagerank(H, alpha=0.9, weight='weight')[initial]
33 #-----Info Sumidero-----
34 datos[fila,8]=nx.degree_centrality(H)[final]
35 datos[fila,9]=nx.clustering(H, nodes=final)
36 datos[fila,10]=nx.closeness_centrality(H, u=final)
37 datos[fila,11]=nx.load_centrality(H, v=final)
38 datos[fila,12]=nx.eccentricity(H, v=final)
39 datos[fila,13]=nx.pagerank(H, alpha=0.9, weight='weight')[final]
40 datos[fila,14]=orden[i]
41 fila+=1

```

caracterizacion.py

4.5. Resultados

Se elaboró unos diagramas de caja para ver el comportamiento de las características de los nodos fuente y sumidero 27, 28, 29, 30, 31, 32, 33, 34, 35. Los valores del eje de las abscisas representan el orden de los grafos.

Con esa información se hace una matriz de correlación 36 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 36 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

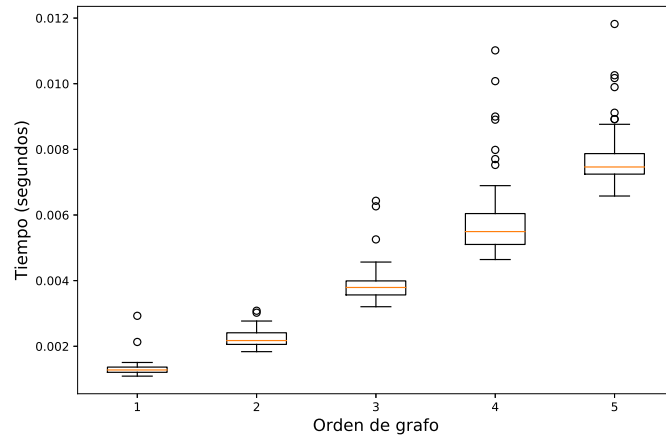


Figura 27: Diagrama de caja de tiempo.

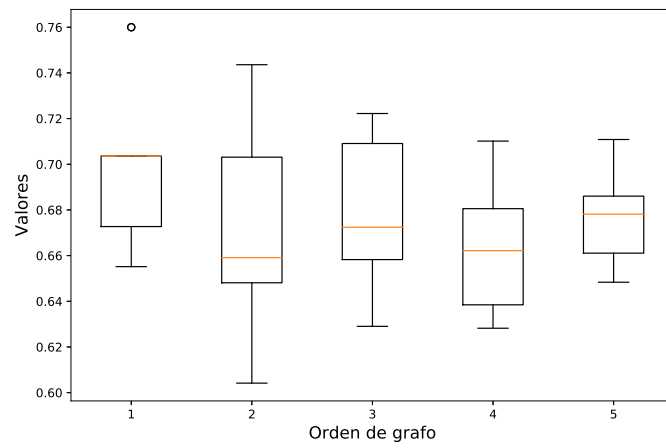


Figura 28: Diagrama de caja de centralidad de nodos fuentes.

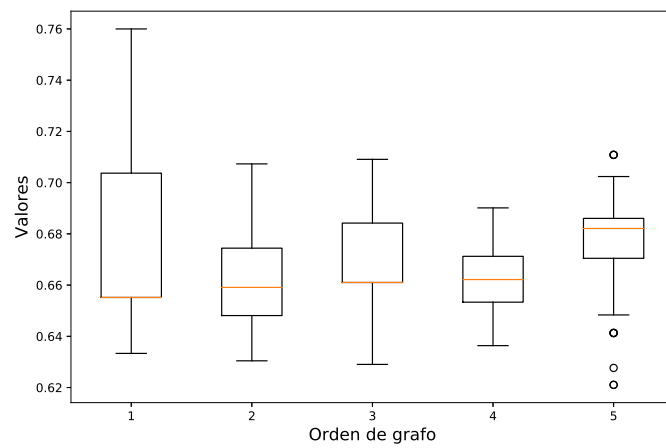


Figura 29: Diagrama de caja de centralidad de nodos sumidero.

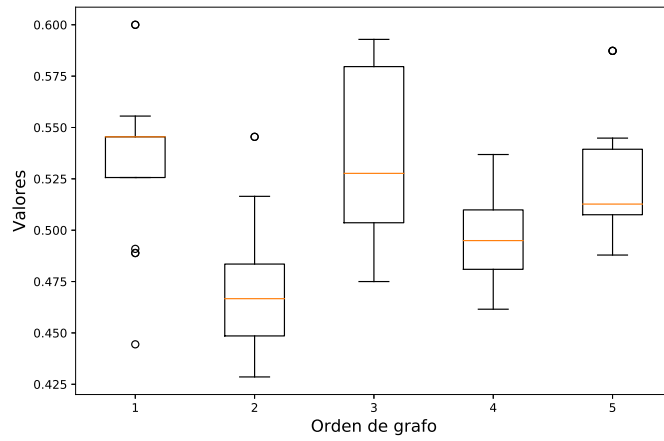


Figura 30: Diagrama de caja de clustering de nodos fuentes.

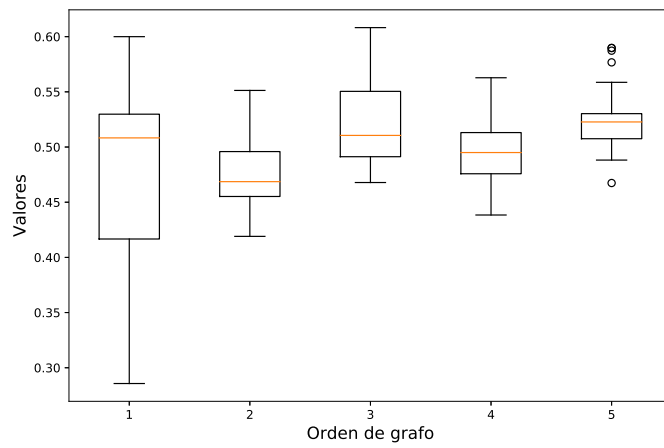


Figura 31: Diagrama de caja de clustering de nodos sumidero.

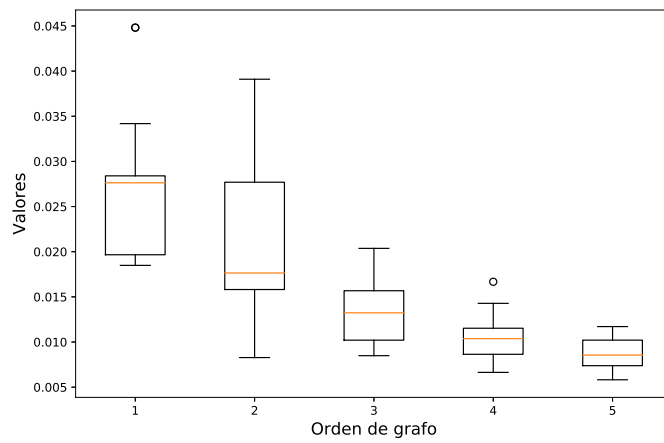


Figura 32: Diagrama de caja de load centrality de nodos fuentes.

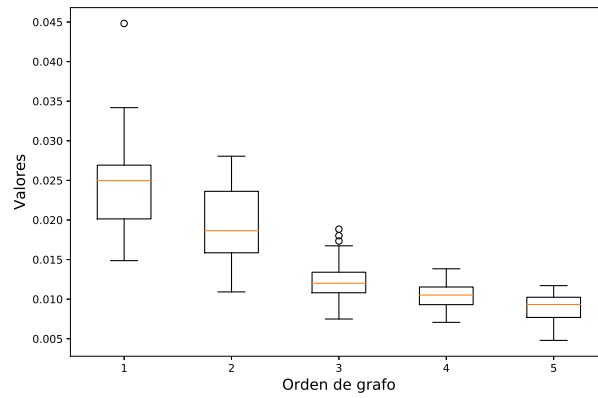


Figura 33: Diagrama de caja de load centrality de nodos sumidero.

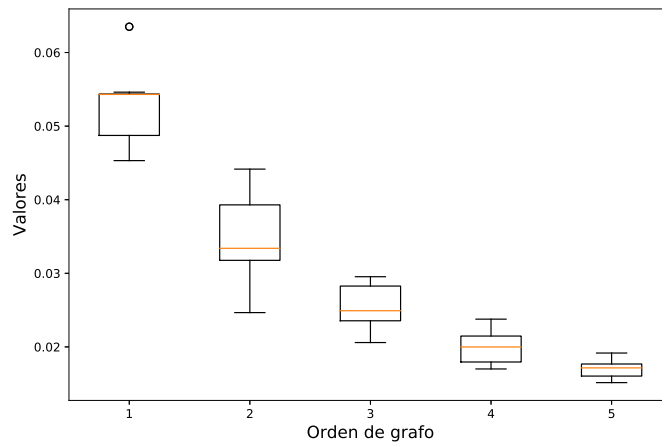


Figura 34: Diagrama de caja de pagerank de nodos fuentes.

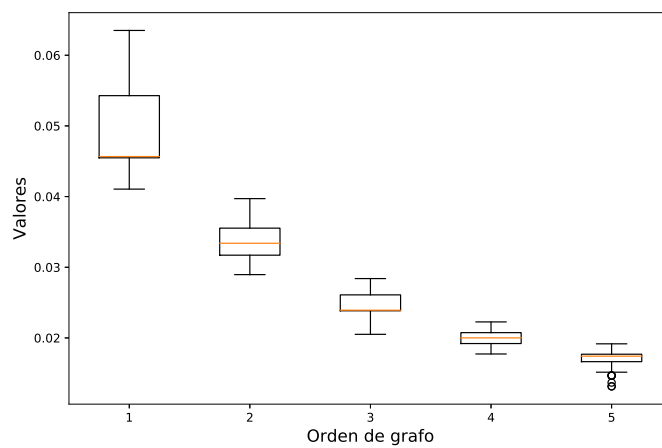


Figura 35: Diagrama de caja de load pagerank de nodos sumidero.

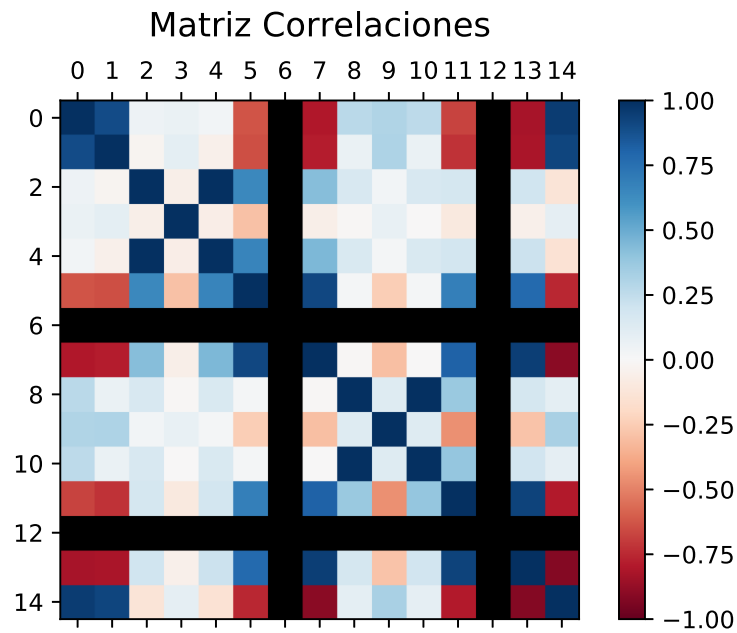


Figura 36: Correlaciones de las características de los nodos fuente y sumidero.

	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

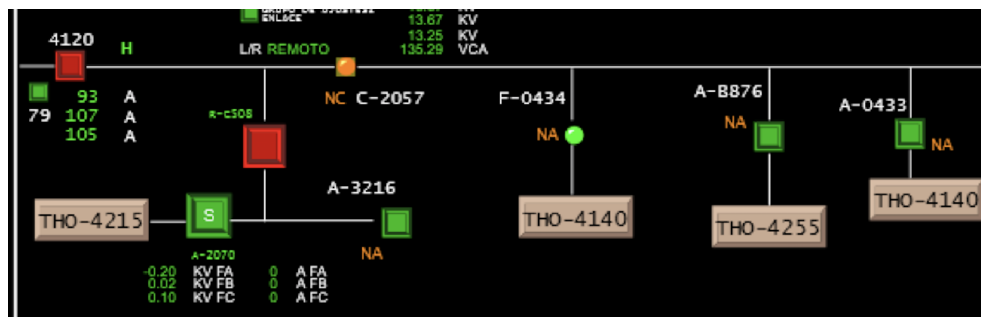


Figura 37: Ejemplo de circuito.

5. 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.

5.1. 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 líneas y para cambiar su topología.

Como se aprecia en la figura 37 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 37 representan componentes cerrados, es decir por donde fluye la energía. Y los nodos en color verde son aquellos donde el circuito está abierto.

5.2. Metodología

Primero se ingresaron los datos para poder representar la red, lo que se obtiene es un grafo como 38. 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 39 donde los nodos simbolizan las líneas y las aristas son los componentes que las unen. Las etiquetas de las líneas 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 líneas 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.

```

1 cerrados=['4120 ', 'R-c508 ', 'C-2057 ']
2 abiertos=['A-2070 ', 'A-B876 ', 'F-0434 ', 'A-0433 ']
3 aux=['Aux ', 'Aux1 ', 'Aux2 ', 'Aux3 ', 'Aux4 ']
4
5
6 Grafo_disp=nx.Graph()
7 Grafo_disp.add_edges_from([( '4120 ', 'Aux' ),( 'Aux ', 'R-c508 ' ),( 'R-c508 ', 'Aux1 ' ),
8                             ( 'Aux1 ', 'A-2070 ' ),( 'Aux ', 'C-2057 ' ),
9                             ( 'C-2057 ', 'Aux2 ' ),( 'Aux2 ', 'F-0434 ' ),
10                            ( 'Aux2 ', 'Aux3 ' ),( 'Aux3 ', 'A-B876 ' ),
11                            ( 'Aux3 ', 'Aux4 ' ),( 'Aux4 ', 'A-0433 ' )])
12
13 labels_disp={('4120 ', 'Aux'): 'H1 ', ('Aux ', 'R-c508 '): 'V1 ',
14              ('R-c508 ', 'Aux1 '): 'V2 ', ('Aux1 ', 'A-2070 '): 'H3 ',
15              ('Aux ', 'C-2057 '): 'H2 ', ('C-2057 ', 'Aux2 '): 'H4 ',
16              ('Aux2 ', 'F-0434 '): 'V3 ', ('Aux2 ', 'Aux3 '): 'H5 ',
17              ('Aux3 ', 'A-B876 '): 'V4 ', ('Aux3 ', 'Aux4 '): 'H6 ',
18              ('Aux4 ', 'A-0433 '): 'V5 '}
19
20 labels={('H1 ', 'H2 '): 'Aux ', ('H1 ', 'V1 '): 'Aux ', ('H2 ', 'H4 '): 'C-2057 ',
21         ('V1 ', 'V2 '): 'R-C508 ', ('V2 ', 'H3 '): 'Aux1 ', ('H4 ', 'H5 '): 'Aux2 ',
22         ('H4 ', 'V3 '): 'Aux2 ', ('H5 ', 'H6 '): 'Aux3 ', ('H5 ', 'V4 '): 'Aux3 ',
23         ('H6 ', 'V5 '): 'Aux4 ', ('H3 ', 'Sumidero '): 'A-2070 ',
24         ('V3 ', 'Sumidero '): 'F-0434 ', ('V4 ', 'Sumidero '): 'A-B876 ',
25         ('V5 ', 'Sumidero '): 'A-0433 ', ('Fuente ', 'H1 '): '4120 ',
26         ('Fuente ', 'H3 '): 'A-2070 ', ('Fuente ', 'V3 '): 'F-0434 ',
27         ('Fuente ', 'V4 '): 'A-B876 ', ('Fuente ', 'V5 '): 'A-0433 '}
28
29 Grafo_lineas=nx.Graph()
30 Grafo_lineas.add_edges_from(list(labels.keys()))
31
32 #-----Graficar-----
33 nx.draw_networkx_nodes(Grafo_disp, pos_disp, nodelist=abiertos, node_color='green',
34                        node_shape='s')
35 nx.draw_networkx_nodes(Grafo_disp, pos_disp, nodelist=cerrados, node_color='red',
36                        node_shape='s')
37 nx.draw_networkx_nodes(Grafo_disp, pos_disp, nodelist=aux, alpha=0)
38 nx.draw_networkx_edges(Grafo_disp, pos_disp, edge_color='white')
39 plt.axis('off')
40 plt.style.use('dark_background')
41 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 40, 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 41 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.

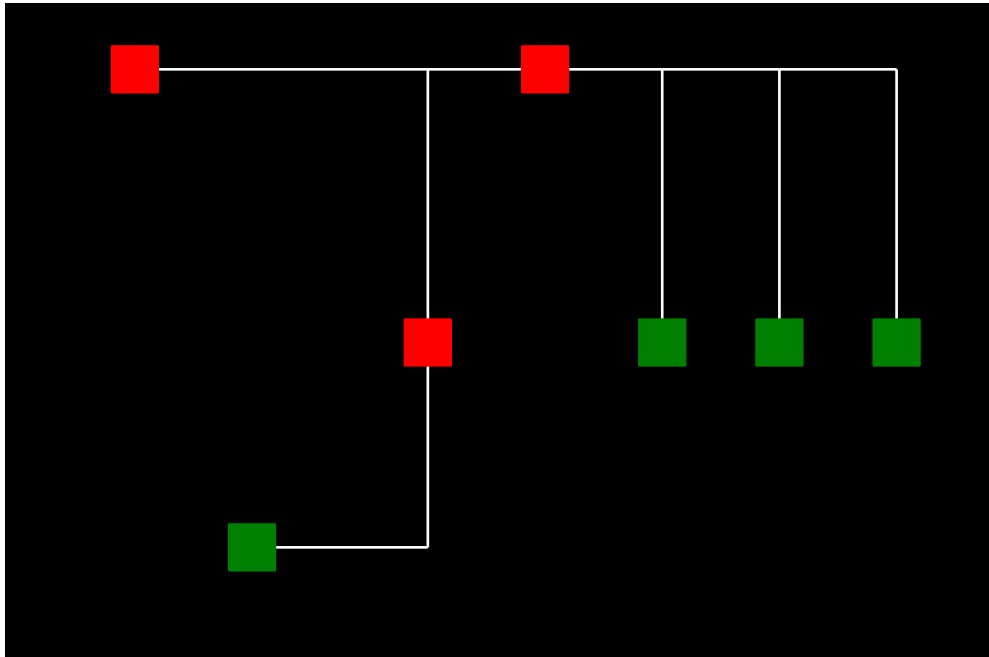


Figura 38: Representación de circuito.

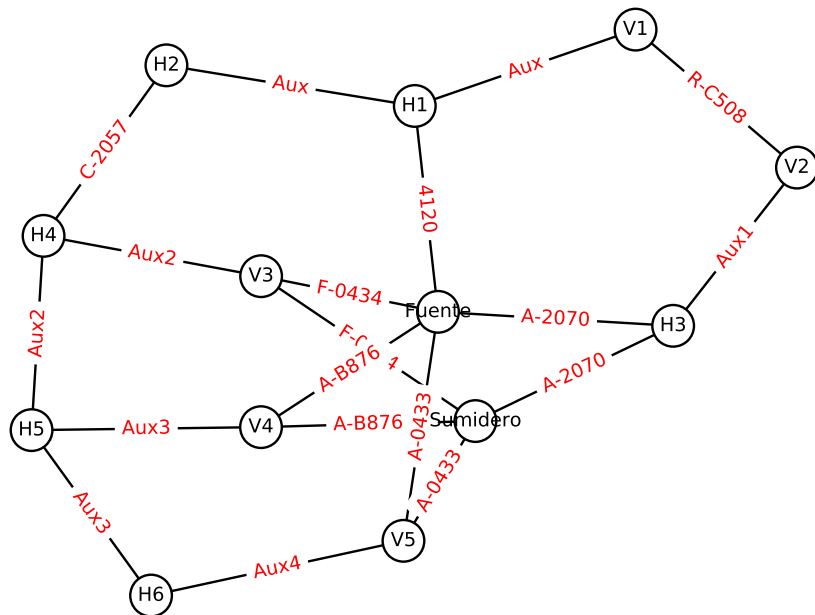


Figura 39: Grafo auxiliar.

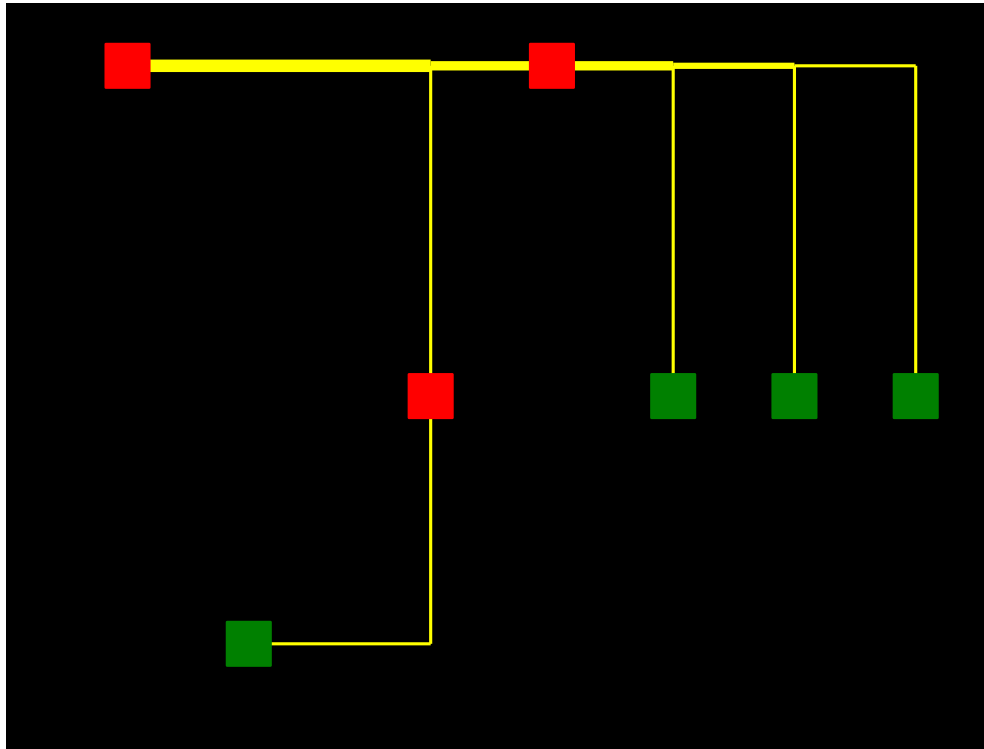


Figura 40: Visualización de configuración de la red.

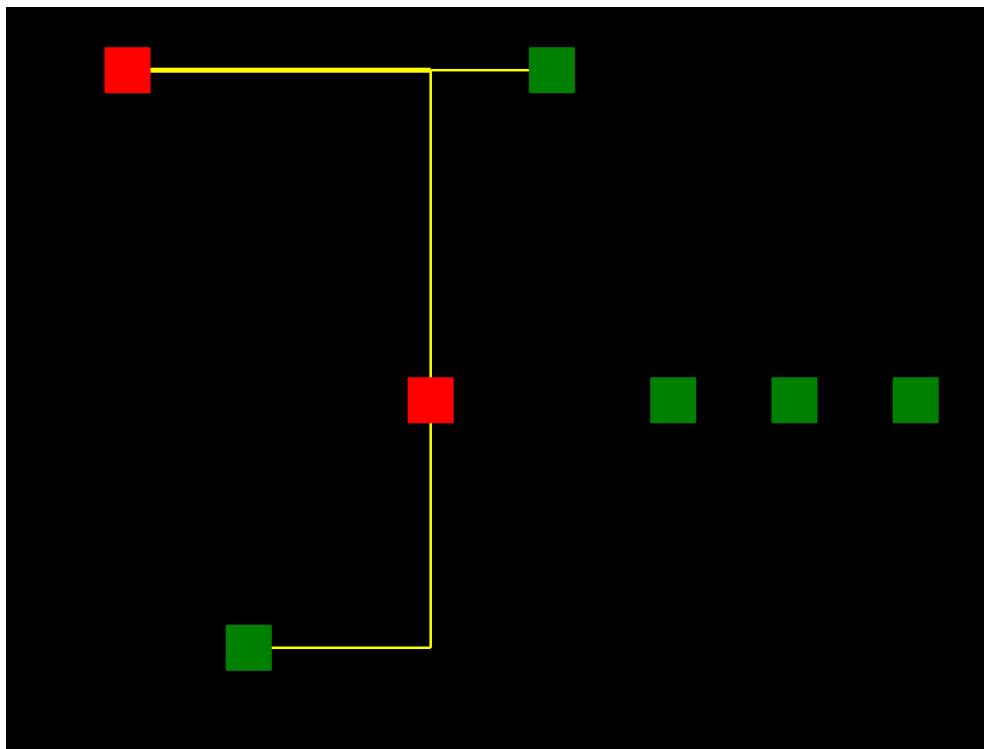


Figura 41: Visualización de configuración de la red.

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.