Portafolio Optimización de Flujo en Redes

Orlando Lázaro Ruiloba Torres 5270

4 de junio de 2019

UANL

FIME

Docente: Dra. Satu Elisa Schaeffer

Semestre: Enero - Junio 2019

Posgrado en Ingeniería de Sistemas

1. Introdución

Como parte de este ejercicio, a continuación se muestran las tareas realizadas a lo largo del curso con sus debidos arreglos. Las tareas aparecen de forma que primero se encuentra la versión calificada por la profesora en clase y posteriormente la versión corregida.

Tarea 1 Optimización de Flujo en Redes

Orlando Lázaro Ruiloba Torres

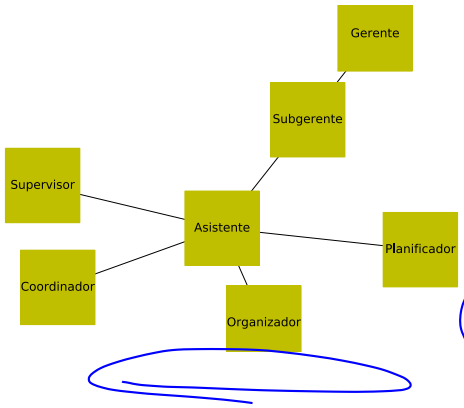
11 de febrero de 2019

1. Grafo simple no dirigido acíclico

Un ejemplo de la aplicación práctica de un grafo simple no dirigido acíclico, pudiera ser el organigrama de una empresa o institución [1], ya que este constituye una representación gráfica de la estructura de la misma, donde los vértices o nodos son los puestos de trabajo, y los arcos representan las relaciones existentes entre ellos ya sea directa o indirectamente.

El código en Python se muestra a continuación:

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

GNDA = nx.Graph()

GNDA.add_node("Gerente")
GNDA.add_nodes_from(["Subgerente","Asistente","Coordinador","Planificador","Organizador","Supervisor"])

GNDA.add_edges_from([("Gerente","Subgerente"),("Subgerente","Asistente")])
GNDA.add_edges_from([("Asistente","Coordinador"),("Asistente","Planificador")])
GNDA.add_edges_from([("Asistente","Organizador"),("Asistente","Supervisor")])

nx.draw_spring(GNDA, node_size = 5500, node_color = 'y', node_shape = 's', with_labels=True)
plt.draw()
plt.savefig("GNDA.eps")
plt.show()
```

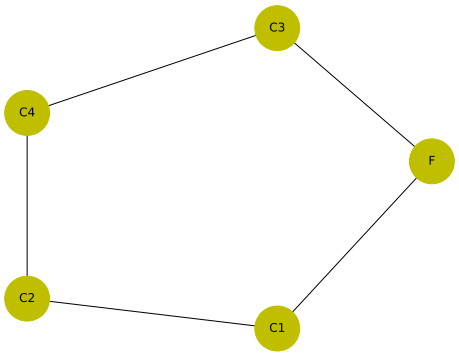


2. Grafo simple no dirigido cíclico

Un ejemplo práctico de la aplicación de este tipo de grafo, pudiera ser la representación de una red de distribución directa según [2], donde los productos salen de la fábrica y van directamente al consumidor final, los vértices serían los clientes y la fábrica, y los arcos el camino que los une, debiendo volver finalmente al vértice que representa el punto inicial (fábrica), repitiendo este proceso tantas veces como sea necesario, sin seguir un orden específico.

El código en Python se muestra a continuación:

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

GNDC = nx.Graph()

GNDC.add_node("F")
GNDC.add_nodes_from(["C1","C2","C3","C4"])

GNDC.add_edges_from([("F","C3"),("C3","C4"),("C4","C2"),("C2","C1"),("C1","F")])

nx.draw_spectral(GNDC, node_size = 2000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GNDC.eps")
plt.show()
```

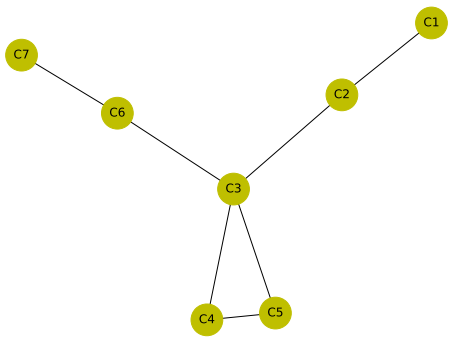


3. Grafo simple no dirigido reflexivo

Para este caso, un ejemplo en la práctica sería, de acuerdo a [2] una red de carreteras que interconecta diferentes ciudades, donde los vértices serían las ciudades y los arcos las carreteras que las unen.

El código en Python es:

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

GNDR = nx.Graph()

GNDR.add_nodes_from(["C1","C7"])
GNDR.add_edges_from([("C1","C2"),("C2","C3"),("C3","C4"),("C4","C5"),("C5","C3"),("C3","C6"),
                     ("C6","C7"),("C7","C7")])

nx.draw(GNDR, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GNDR.eps")
plt.show()
```

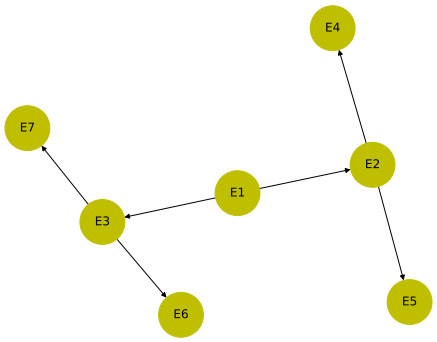


4. Grafo simple dirigido acíclico

Un ejemplo para este caso en particular, pudiera ser de ~~acuerdo a~~ [2], una estructura jerárquica de datos, donde los vértices corresponden a entidades y los arcos a las relaciones existentes entre las mismas.

El código en Python es el siguiente:

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

GDA = nx.DiGraph()

GDA.add_nodes_from(["E1","E7"])
GDA.add_edges_from([("E1","E2"),("E1","E3"),("E2","E4"),("E2","E5"),("E3","E6"),("E3","E7")])

nx.draw(GDA, node_size = 2000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GDA.eps")
plt.show()
```

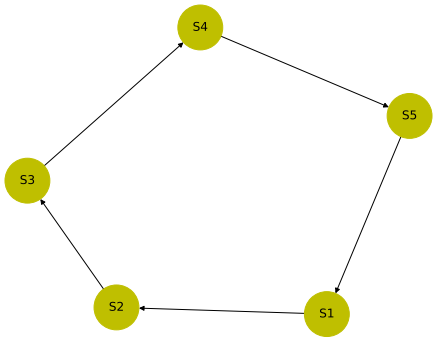


5. Grafo simple dirigido cíclico

Un ejemplo en la práctica de este tipo de grafo ~~de acuerdo a~~ [2], puede ser la representación del proceso para la confección en masa de un determinado producto en una fábrica, ya que esto implica el desarrollo de varios subprocesos que tienen una secuencia determinada y que se van a repetir un número específico de veces, donde los vértices serían los subprocesos y los arcos la relación ordenada existente entre ellos.

El código en Python se presenta a continuación:

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

GDC = nx.DiGraph()

GDC.add_nodes_from(["S1","S5"])
GDC.add_edges_from([("S1","S2"),("S2","S3"),("S3","S4"),("S4","S5"),("S5","S1")])

nx.draw(GDC, node_size = 2000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GDC.eps")
plt.show()
```

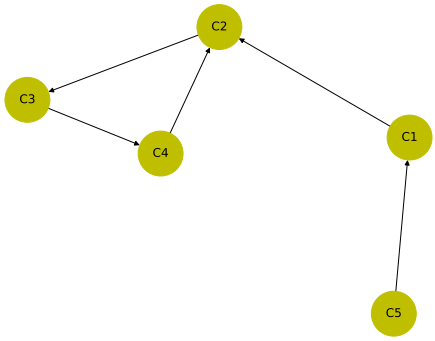


6. Grafo simple dirigido reflexivo

Un ejemplo en la práctica de este tipo de grafo se vería reflejado de acuerdo a [2], en una red de aviación donde los vértices serían las ciudades o países a visitar y los arcos representan el camino recorrido por el avión.

El código en Python es:

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

GDR = nx.DiGraph()

GDR.add_nodes_from(["C1","C5"])
GDR.add_edges_from([("C1","C2"),("C2","C3"),("C3","C4"),("C4","C2"),("C5","C1"),("C5","C5")])

nx.draw(GDR, node_size = 2000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GDR.eps")
plt.show()
```

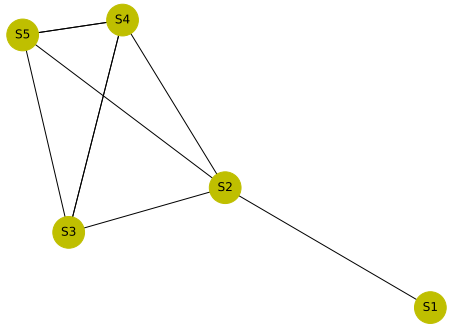


7. Multigrafo no dirigido acíclico

Un ejemplo en la práctica de este tipo de multigrafo según la explicación en [4] pudiera ser el caso de un museo, que desarrolla una exposición en varias salas al unísono, donde la situación ideal sería poder transitar por todas las salas sin tener que pasar necesariamente dos veces por la misma, se consideran las salas como los vértices, y las aristas, los caminos que las unen.

El código en Python es:

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

MNDA = nx.MultiGraph()

MNDA.add_nodes_from(["S1","S5"])
MNDA.add_edges_from([("S1","S2"),("S2","S3"),("S2","S4"),("S2","S5"),("S3","S4"),("S3","S4"),("S4","S5"),
                     ("S4","S5"),("S3","S5")])

nx.draw(MNDA, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MNDA.eps")
plt.show()
```

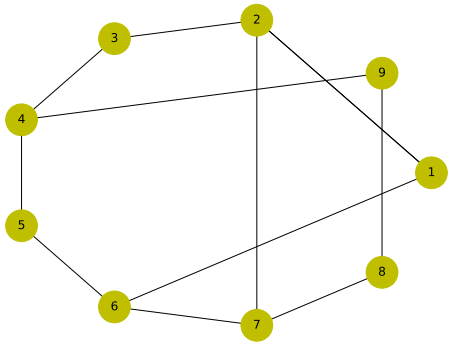


8. Multigrafo no dirigido cíclico

Un ejemplo práctico para este caso en particular, se [illegible] a [3] en la generación de redes tipo pequeño mundo, concepto que de hecho es muy común y nada alejado de nuestras experiencias diarias, ya que poco después de conocer a un extraño, uno se da cuenta de que tiene un amigo en común con él, este concepto se basa en el principio de los seis grados de separación, que establece que entre cualesquiera dos personas del mundo, existe una media de seis conexiones de amistad.

El código en Python es:

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

MNDC = nx.MultiGraph()

MNDC.add_nodes_from(["1","9"])
MNDC.add_edges_from([("1","2"),("2","3"),("3","4"),("4","5"),("5","6"),("6","7"),("7","8"),("8","9"),
                     ("2","1"),("1","6"),("2","7"),("4","9")])

nx.draw_circular(MNDC, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MNDC.eps")
plt.show()
```

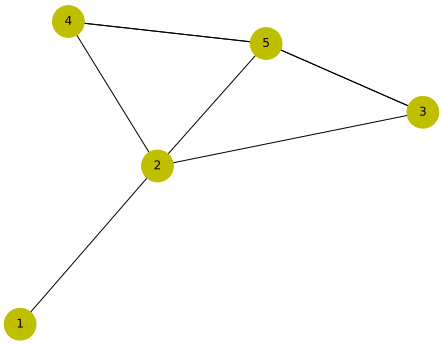


9. Multigrafo no dirigido reflexivo

Un ejemplo de este tipo de multigrafo en la vida cotidiana puede ser ~~según lo expuesto en~~ [7] una red social de personas, donde los vértices pueden representar hombres o mujeres, personas con diferentes nacionalidades, edades, ingresos, etc y los arcos pueden simbolizar amistad, proximidad geográfica, conocimiento profesional, entre otros.

El código en Python es:

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

MNDR = nx.MultiGraph()

MNDR.add_nodes_from(["1","5"])
MNDR.add_edges_from([("1","2"),("2","3"),("2","4"),("2","5"),("3","5"),("4","5"),("3","5"),
    ("4","5"),("5","5")])

nx.draw(MNDR, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MNDR.eps")
plt.show()
```

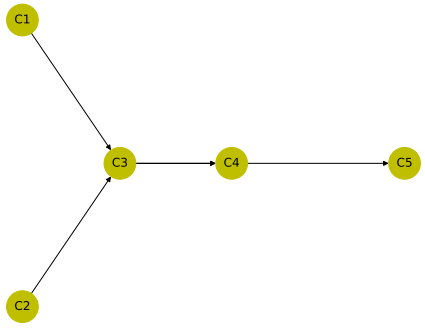


10. Multigrafo dirigido acíclico

Un ejemplo de una aplicación práctica de este tipo de multigrafo se pone de manifiesto en una pequeña red de electricidad entre cinco ciudades, donde los vértices serían las ciudades y el tendido eléctrico serían las aristas que las unen, en el caso de C4, recibe la energía a través de C3 en dos canales que alimentan a la totalidad de la ciudad.

El código en Python es:

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

MDA = nx.MultiDiGraph()

MDA.add_nodes_from(["C1","C5"])
MDA.add_edges_from([("C1","C3"),("C2","C3"),("C3","C4"),("C3","C4"),("C4","C5")])

nx.draw_spectral(MDA, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MDA.eps")
plt.show()
```

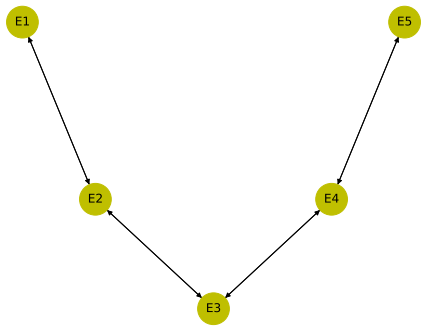


11. Multigrafo dirigido cíclico

Un ejemplo de la puesta en práctica de este tipo de multigrafo pudiera ser según [5] la modelación de 5 estaciones de una red de metro, donde cada estación da lugar a un vértice y los arcos simbolizan el camino de ida y vuelta entre cada una de las estaciones.

El código en Python es:

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

MDC = nx.MultiDiGraph()

MDC.add_nodes_from(["E1","E5"])
MDC.add_edges_from([("E1","E2"),("E2","E1"),("E2","E3"),("E3","E2"),("E3","E4"),("E4","E3"),
                    ("E4","E5"),("E5","E4")])

nx.draw_spectral(MDC, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MDC.eps")
plt.show()
```

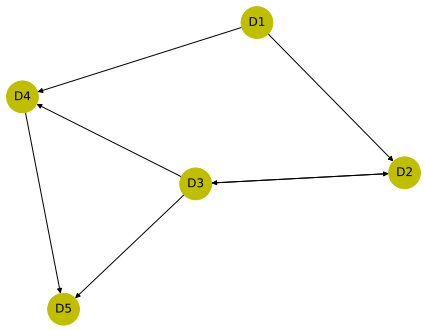


12. Multigrafo dirigido reflexivo

Un ejemplo de la puesta en práctica de este tipo de multigrafo pudiera ser según se muestra en [6] una red de comunicación en donde los vértices son los dispositivos de red y los arcos son los medios de networking (medios de transmisión basados en cobre o fibra óptica) que aseguran el envío y recepción de los mensajes y la información a través de la red de datos.

El código en Python es:

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

MDR = nx.MultiDiGraph()

MDR.add_nodes_from(["D1","D5"])
MDR.add_edges_from([("D1","D2"),("D1","D4"),("D2","D3"),("D3","D2"),("D3","D4"),("D3","D5"),("D4","D5"),
                    ("D5","D5")])

nx.draw(MDR, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MDR.eps")
plt.show()
```



Referencias

[1] Universidad de Valladolid. Grafos no dirigidos acíclicos: Árboles. 2008.

[2] Prof. Kryscia Daviana Ramírez Benavides. Teoría de grafos y Árboles. Universidad de Costa Rica. Escuela de Ciencias de la Computación e Informática.

[3] J.Torres. Tema: 2 red compleja como ejemplo de sistema complejo. 2018.

[4] Hernán S. Nottoli. Teoría de grafos. aplicaciones al diseño arquitectónico. 1998.

[5] Alfonso Recuero. Aplicaciones de la teoría de grafos: Búsqueda de caminos en una red y análisis de su conectividad.

[6] C. Ortiz y J. Sendra E. Martín, A. Méndez. Tema 5: Grafos. 2011.

[7] Nidia Lizzeth Gómez Duarte. Efectos estructurales en la sincronización de sistemas. Master's thesis, Universidad Autónoma de Nuevo León, 2010.

Tarea 1 Optimización de Flujo en Redes

Orlando Lázaro Ruiloba Torres 5270

3 de junio de 2019

1. Grafo simple no dirigido acíclico

Un ejemplo de la aplicación práctica de un grafo simple no dirigido acíclico, pudiera ser el organigrama de una empresa o institución [2], ya que este constituye una representación gráfica de la estructura de la misma, donde los vértices o nodos son los puestos de trabajo, y los arcos representan las relaciones existentes entre ellos ya sea directa o indirectamente.

El código en Python se muestra a continuación:

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

GNDA = nx.Graph()

GNDA.add_node("Gerente")
GNDA.add_nodes_from(["Subgerente","Asistente","Coordinador","Planificador","Organizador","Supervisor"])

GNDA.add_edges_from([("Gerente","Subgerente"),("Subgerente","Asistente")])
GNDA.add_edges_from([("Asistente","Coordinador"),("Asistente","Planificador")])
GNDA.add_edges_from([("Asistente","Organizador"),("Asistente","Supervisor")])

nx.draw_spring(GNDA, node_size = 5500, node_color = 'y', node_shape = 's', with_labels=True)
plt.draw()
plt.savefig("GNDA.eps")
plt.show()
```

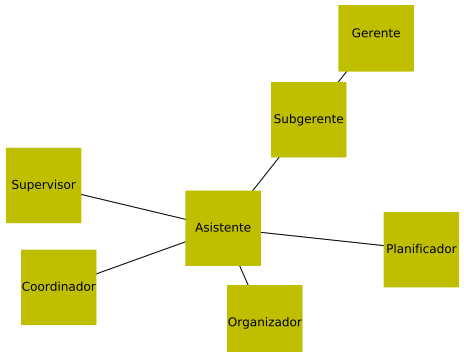


Figura 1: Grafo simple no dirigido acíclico

2. Grafo simple no dirigido cíclico

Un ejemplo práctico de la aplicación de este tipo de grafo, es la representación de una red de distribución directa [1], donde los productos salen de la fábrica y van directamente al consumidor final, los vértices son los clientes y la fábrica, y los arcos el camino que los une, debiendo volver finalmente al vértice que representa el punto inicial (fábrica), repitiendo este proceso tantas veces como sea necesario, sin seguir un orden específico.

El código en Python se muestra a continuación:

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

GNDC = nx.Graph()

GNDC.add_node("F")
GNDC.add_nodes_from(["C1","C2","C3","C4"])

GNDC.add_edges_from([("F","C3"),("C3","C4"),("C4","C2"),("C2","C1"),("C1","F")])

nx.draw_spectral(GNDC, node_size = 2000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GNDC.eps")
plt.show()
```

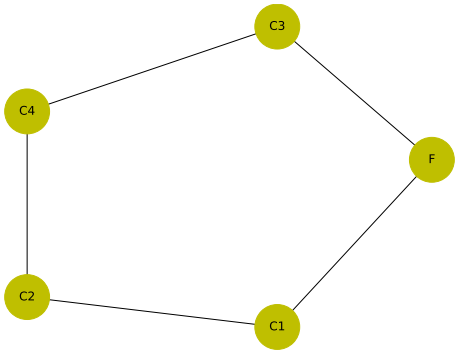


Figura 2: Grafo simple no dirigido cíclico

3. Grafo simple no dirigido reflexivo

Para este caso, un ejemplo en la práctica es una red de carreteras [1] que interconecta diferentes ciudades, donde los vértices serían las ciudades y los arcos las carreteras que las unen.

El código en Python es:

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

GNDR = nx.Graph()

GNDR.add_nodes_from(["C1","C7"])
GNDR.add_edges_from([("C1","C2"),("C2","C3"),("C3","C4"),("C4","C5"),("C5","C3"),("C3","C6"),
                     ("C6","C7"),("C7","C7")])

nx.draw(GNDR, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GNDR.eps")
plt.show()
```

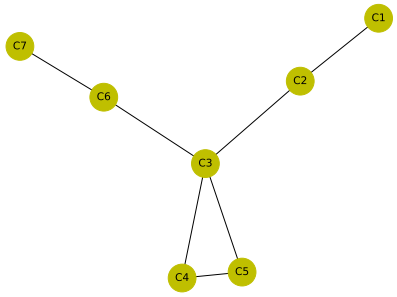


Figura 3: Grafo simple no dirigido reflexivo

4. Grafo simple dirigido acíclico

Un ejemplo para este caso en particular, es una estructura jerárquica de datos [1], donde los vértices corresponden a entidades y los arcos a las relaciones existentes entre las mismas.

El código en Python es el siguiente:

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

GDA = nx.DiGraph()

GDA.add_nodes_from(["E1","E7"])
GDA.add_edges_from([("E1","E2"),("E1","E3"),("E2","E4"),("E2","E5"),("E3","E6"),("E3","E7")])

nx.draw(GDA, node_size = 2000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GDA.eps")
plt.show()
```

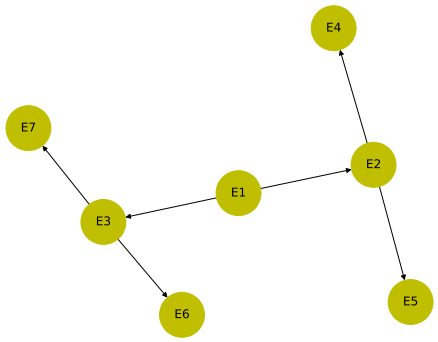


Figura 4: Grafo simple dirigido acíclico

5. Grafo simple dirigido cíclico

Un ejemplo en la práctica de este tipo de grafo es, la representación del proceso para la confección en masa de un determinado producto en una fábrica [1], ya que esto implica el desarrollo de varios subprocesos que tienen una secuencia determinada y que se van a repetir un número específico de veces, donde los vértices son los subprocesos y los arcos la relación ordenada existente entre ellos.

El código en Python se presenta a continuación:

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

GDC = nx.DiGraph()

GDC.add_nodes_from(["S1","S5"])
GDC.add_edges_from([("S1","S2"),("S2","S3"),("S3","S4"),("S4","S5"),("S5","S1")])

nx.draw(GDC, node_size = 2000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GDC.eps")
plt.show()
```

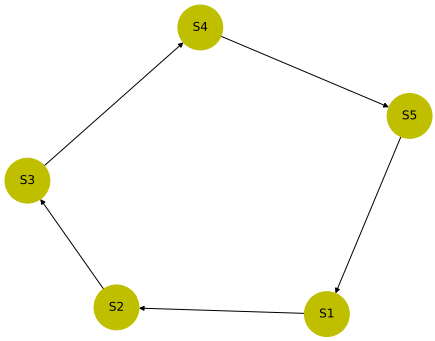


Figura 5: Grafo simple dirigido cíclico

6. Grafo simple dirigido reflexivo

Un ejemplo en la práctica de este tipo de grafo es, en una red de aviación [1] donde los vértices son las ciudades o países a visitar y los arcos representan el camino recorrido por el avión.

El código en Python es:

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

GDR = nx.DiGraph()

GDR.add_nodes_from(["C1","C5"])
GDR.add_edges_from([("C1","C2"),("C2","C3"),("C3","C4"),("C4","C2"),("C5","C1"),("C5","C5")])

nx.draw(GDR, node_size = 2000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GDR.eps")
plt.show()
```

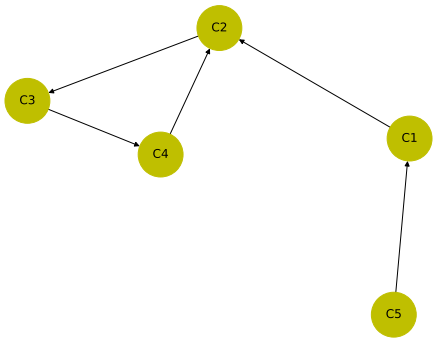


Figura 6: Grafo simple dirigido reflexivo

7. Multigrafo no dirigido acíclico

Un ejemplo en la práctica de este tipo de multigrafo es el caso de un museo, que desarrolla una exposición en varias salas al unísono [6], donde la situación ideal es poder transitar por todas las salas sin tener que pasar necesariamente dos veces por la misma, se consideran las salas como los vértices, y las aristas, los caminos que las unen.

El código en Python es:

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

MNDA = nx.MultiGraph()

MNDA.add_nodes_from(["S1","S5"])
MNDA.add_edges_from([("S1","S2"),("S2","S3"),("S2","S4"),("S2","S5"),("S3","S4"),("S3","S4"),("S4","S5"),
                     ("S4","S5"),("S3","S5")])

nx.draw(MNDA, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MNDA.eps")
plt.show()
```

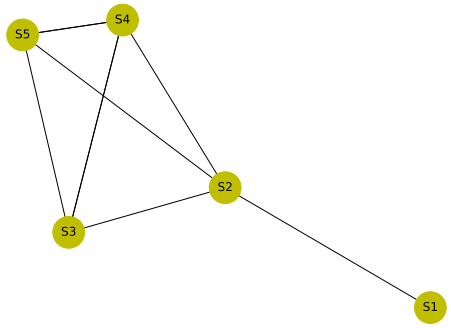


Figura 7: Multigrafo no dirigido acíclico

8. Multigrafo no dirigido cíclico

Un ejemplo práctico para este caso en particular es,la generación de redes tipo pequeño mundo [5], concepto que de hecho es muy común y nada alejado de nuestras experiencias diarias, ya que poco después de conocer a un extraño, uno se da cuenta de que tiene un amigo en común con él, este concepto se basa en el principio de los seis grados de separación, que establece que entre cualesquiera dos personas del mundo, existe una media de seis conexiones de amistad.

El código en Python es:

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

MNDC = nx.MultiGraph()

MNDC.add_nodes_from(["1","9"])
MNDC.add_edges_from([("1","2"),("2","3"),("3","4"),("4","5"),("5","6"),("6","7"),("7","8"),("8","9"),
                     ("2","1"),("1","6"),("2","7"),("4","9")])

nx.draw_circular(MNDC, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MNDC.eps")
plt.show()
```

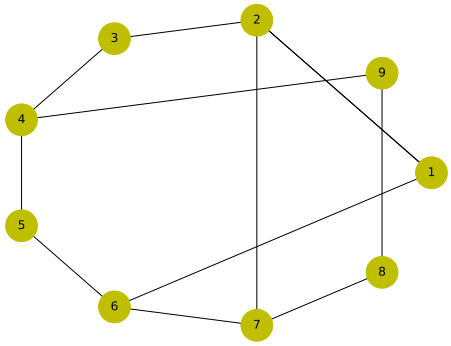


Figura 8: Multigrafo no dirigido cíclico

9. Multigrafo no dirigido reflexivo

Un ejemplo de este tipo de multigrafo en la vida cotidiana es una red social de personas [3], donde los vértices pueden representar hombres o mujeres, personas con diferentes nacionalidades, edades, ingresos, etc y los arcos pueden simbolizar amistad, proximidad geográfica, conocimiento profesional, entre otros.

El código en Python es:

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

MNDR = nx.MultiGraph()

MNDR.add_nodes_from(["1","5"])
MNDR.add_edges_from([("1","2"),("2","3"),("2","4"),("2","5"),("3","5"),("4","5"),("3","5"),
    ("4","5"),("5","5")])

nx.draw(MNDR, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MNDR.eps")
plt.show()
```

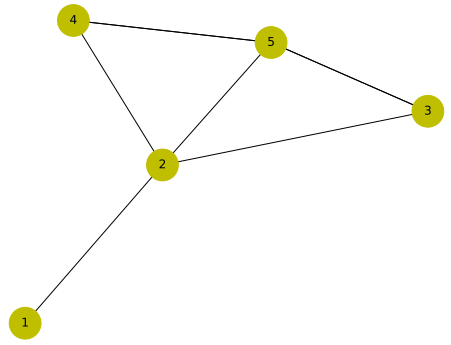


Figura 9: Multigrafo no dirigido reflexivo

10. Multigrafo dirigido acíclico

Un ejemplo de una aplicación práctica de este tipo de multigrafo se pone de manifiesto en una pequeña red de electricidad entre cinco ciudades, donde los vértices serían las ciudades y el tendido eléctrico constituye las aristas que las unen, en el caso de C4, recibe la energía a través de C3 en dos canales que alimentan a la totalidad de la ciudad.

El código en Python es:

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

MDA = nx.MultiDiGraph()

MDA.add_nodes_from(["C1","C5"])
MDA.add_edges_from([("C1","C3"),("C2","C3"),("C3","C4"),("C3","C4"),("C4","C5")])

nx.draw_spectral(MDA, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MDA.eps")
plt.show()
```

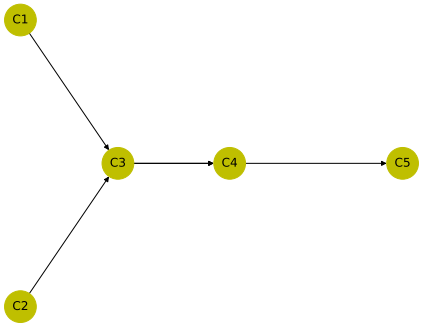


Figura 10: Multigrafo dirigido acíclico

11. Multigrafo dirigido cíclico

Un ejemplo de la puesta en práctica de este tipo de multigrafo es la modelación de 5 estaciones de una red de metro [7], donde cada estación da lugar a un vértice y los arcos simbolizan el camino de ida y vuelta entre cada una de las estaciones.

El código en Python es:

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

MDC = nx.MultiDiGraph()

MDC.add_nodes_from(["E1","E5"])
MDC.add_edges_from([("E1","E2"),("E2","E1"),("E2","E3"),("E3","E2"),("E3","E4"),("E4","E3"),
                    ("E4","E5"),("E5","E4")])

nx.draw_spectral(MDC, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MDC.eps")
plt.show()
```

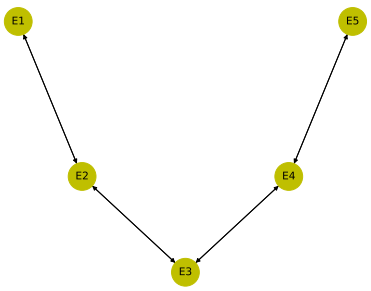


Figura 11: Multigrafo dirigido cíclico

12. Multigrafo dirigido reflexivo

Un ejemplo de la puesta en práctica de este tipo de multigrafo es una red de comunicación [4] en donde los vértices son los dispositivos de red y los arcos son los medios de networking (medios de transmisión basados en cobre o fibra óptica) que aseguran el envío y recepción de los mensajes y la información a través de la red de datos.

El código en Python es:

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

MDR = nx.MultiDiGraph()

MDR.add_nodes_from(["D1","D5"])
MDR.add_edges_from([("D1","D2"),("D1","D4"),("D2","D3"),("D3","D2"),("D3","D4"),("D3","D5"),("D4","D5"),
                    ("D5","D5")])

nx.draw(MDR, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MDR.eps")
plt.show()
```

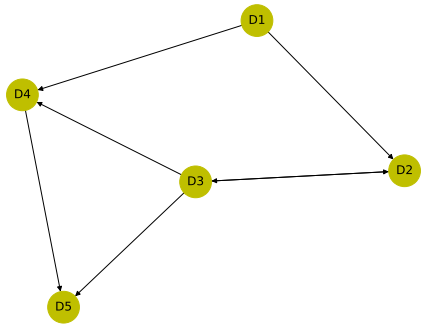


Figura 12: Multigrafo dirigido reflexivo

Referencias

[1] Kryscia Daviana Ramírez Benavides. Teoría de grafos y Árboles. Universidad de Costa Rica. Escuela de Ciencias de la Computación e Informática.

[2] Universidad de Valladolid. Grafos no dirigidos acíclicos: Árboles. 2008. URL `http://www.ma.uva.es/~antonio/Industriales/Apuntes_07-08/LabM/Grafos_2008-4.pdf`.

[3] Nidia Lizzeth Gómez Duarte. Efectos estructurales en la sincronización de sistemas. 2010.

[4] C. Ortiz y J. Sendra E. Martín, A. Méndez. Tema 5: Grafos. 2011.

[5] J.Torres. Tema: 2 red compleja como ejemplo de sistema complejo. 2018. URL `https://www.ugr.es/~jtorres/Tema_2_redes_complejas.pdf`.

[6] Hernán S. Nottoli. Teoría de grafos. aplicaciones al diseño arquitectónico. 1998.

[7] Alfonso Recuero. Aplicaciones de la teoría de grafos: Búsqueda de caminos en una red y análisis de su conectividad.

2. Conclusión Tarea 1

Con la realización de este ejercicio se tuvo la oportunidad de aprender y, poder apreciar de forma más cercana, la utilidad práctica que tiene en nuestra vida cotidiana el empleo de los grafos, ya que los mismos se pueden utilizar para modelar redes pluviales, sociales, de carreteras. etc, sin dejar de mencionar que se estudiaron también las características de cada tipo de grafo en particular.

Tarea 2 Optimización de Flujo en Redes

Orlando Lázaro Ruiloba Torres 5270

26 de febrero de 2019

1. Algoritmo de acomodo bipartito

El diseño de este algoritmo se crea colocando primero los vértices en dos filas, de acuerdo con sus tipos. Luego, las posiciones dentro de las filas se optimizan para minimizar los cruces de arcos, utilizando el algoritmo de Sugiyama [1]. Este algoritmo de acomodo actualmente solo funciona en dos dimensiones [8].

El código en python se muestra a continuación:

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

GDA = nx.DiGraph()

GDA.add_nodes_from(["E1","E2","E3"], bipartite=0)
GDA.add_nodes_from(["E4","E5","E6","E7"], bipartite=1)

GDA.add_edges_from([("E1","E4"),("E1","E5"),("E2","E6"),("E2","E7"),("E3","E4"),("E3","E7")])

nx.draw(GDA, pos=nx.bipartite_layout(GDA,["E1","E2","E3"]), node_size = 2000, node_color = 'y',
       node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GDA.eps")
plt.show()
```

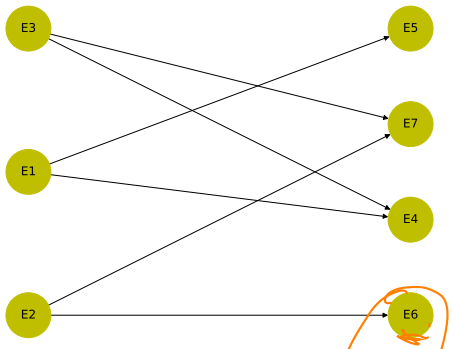


Figura 1: Algoritmo bipartito en grafo dirigido acíclico

2. Algoritmo de acomodo circular

Es un algoritmo de diseño que permite colocar los vértices en topologías de anillo y estrella interconectados. Produce diseños que enfatizan las estructuras de grupo y árbol dentro de una red. Crea particiones de nodo mediante el análisis de la estructura de conectividad de la red, y organiza las particiones como círculos separados. Los círculos en sí están dispuestos en forma de árbol radial [2].

El código en python es:

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

MNDC = nx.MultiGraph()
MNDC.add_nodes_from(["1","9"])

MNDC.add_edges_from([("1","2")], color='lightblue', weight=8)
MNDC.add_edges_from([("2","3"),("3","4"),("4","5"),("5","6"),("6","7"),("7","8"),("8","9"),
                     ("2","1"),("1","6"),("2","7"),("4","9")], color='black', weight=2)

edges = MNDC.edges()

colors = []
weight = []

for (u,v,attrib_dict) in list(MNDC.edges.data()):
    colors.append(attrib_dict['color'])
    weight.append(attrib_dict['weight'])

color_map = []
for node in MNDC:
    if (node == "1" or node == "2"):
        color_map.append('blue')
    else:
        color_map.append('red')

nx.draw_circular(MNDC, edges=edges, edge_color=colors, width=weight, node_size = 1000,
                 node_color = color_map, node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MNDC.eps")
plt.show()
```

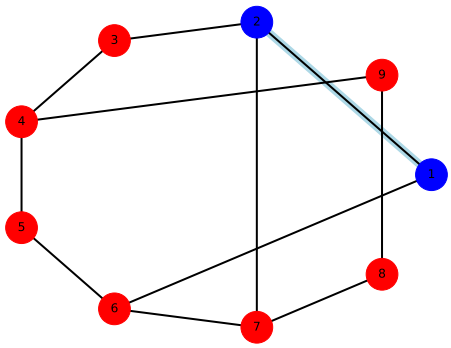


Figura 2: Algoritmo circular en multigrafo no dirigido cíclico

3. Algoritmo de acomodo kamada kawai

Este algoritmo de acomodo se basa en la idea de usar solo fuerzas de resorte entre todos los pares de vértices, el criterio estético que considera es que la distancia entre los vértices debe ser igual a la distancia teórica. Mantiene implícitamente los dos criterios anteriores [11].

El código en python se muestra a continuación:

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

GNDR = nx.Graph()

GNDR.add_nodes_from(["C1","C7"])
GNDR.add_edges_from([("C1","C2"),("C2","C3"),("C3","C4"),("C4","C5"),("C5","C3"),("C3","C6"),
                     ("C6","C7"),("C7","C7")])

color_map = []
for node in GNDR:
    if (node == "C7"):
        color_map.append('blue')
    else:
        color_map.append('red')

layout = nx.kamada_kawai_layout(GNDR)

nx.draw(GNDR, pos=layout, node_size = 1000, node_color = color_map, node_shape = 'o',
     with_labels=True)
plt.draw()
plt.savefig("GNDR.eps")
plt.show()
```

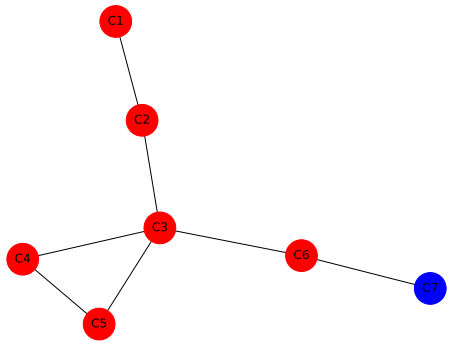


Figura 3: Algoritmo kamada kawai en grafo simple no dirigido reflexivo

4. Algoritmo de acomodo aleatorio

Es un algoritmo que posiciona los vértices uniformemente al azar en el cuadrado unitario. Para cada vértice se genera una posición, al elegir cada una de las coordenadas tenues de la forma mencionada anteriormente en el intervalo [0.0, 1.0] [6].

El código en pyhton es el siguiente:

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

GNDC = nx.Graph()

GNDC.add_node("F")
GNDC.add_nodes_from(["C1","C2","C3","C4"])

GNDC.add_edges_from([("F","C3"),("C3","C4"),("C4","C2"),("C2","C1"),("C1","F")])

nx.draw_random(GNDC, node_size = 2000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GNDC.eps")
plt.show()
```

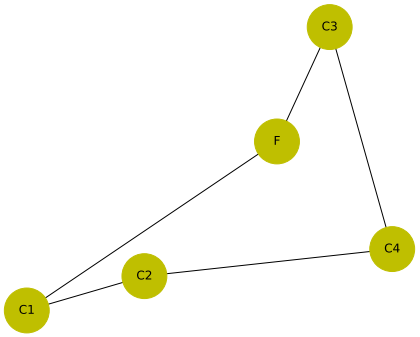


Figura 4: Algoritmo aleatorio en grafo no dirigido cíclico

5. Algoritmo de acomodo espectral

El código en python se muestra a continuación:

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

MNDR = nx.MultiGraph()

MNDR.add_nodes_from(["1","5"])
MNDR.add_edges_from([("1","2"),("2","3"),("2","4"),("2","5"),("3","5"),("4","5"),("3","5"),
    ("4","5"),("5","5")])

color_map = []
for node in MNDR:
    if (node == "5"):
        color_map.append('yellow')
    else:
        color_map.append('red')

nx.draw_spectral(MNDR, node_size = 1000, node_color = color_map, node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MNDR.eps")
plt.show()
```

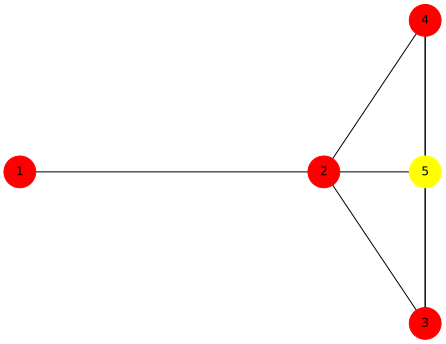


Figura 5: Algoritmo espectral en multigrafo no dirigido reflexivo

6. Algoritmo de acomodo shell

Este algoritmo de acomodo es interesante, ya que le permite al diseñador seleccionar subgrupos de nodos para posicionarlos en círculos concéntricos, en este sentido es menos automático que otros diseños, pero tiene la ventaja crítica de que se puede usar para resaltar características específicas de la red [3].

El código en python es:

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

GDR = nx.DiGraph()

GDR.add_nodes_from(["C1","C5"])
GDR.add_edges_from([("C1","C2"),("C2","C3"),("C3","C4"),("C4","C2"),("C5","C1"),("C5","C5")])

color_map = []
for node in GDR:
    if (node == "C5"):
        color_map.append('blue')
    else:
        color_map.append('red')

nx.draw_shell(GDR, node_size = 2000, node_color = color_map, node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GDR.eps")
plt.show()
```

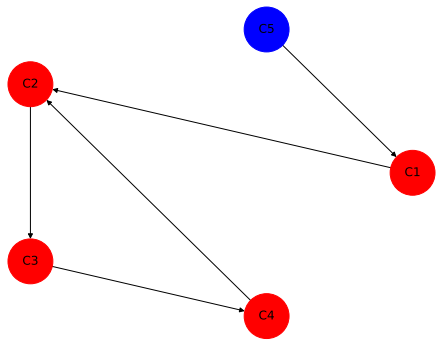


Figura 6: Algoritmo shell en grafo dirigido reflexivo

7. Algoritmo de acomodo resorte

En este caso el algoritmo posiciona los nodos basándose en el algoritmo de fuerza dirigida de Fruchterman-Reingold [7]. Cada elemento se trata como una partícula con una carga eléctrica similar que repele otros elementos. Los conectores actúan como resortes que vuelven a unir los elementos conectados. El algoritmo es bueno para resaltar grupos de objetos relacionados e identificar simetría en el grafo [10].

El código en python se muestra a continuación:

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

GNDA = nx.Graph()

GNDA.add_node("Gerente")
GNDA.add_nodes_from(["Subgerente","Asistente","Coordinador","Planificador","Organizador","Supervisor"])

GNDA.add_edges_from([("Gerente","Subgerente"),("Subgerente","Asistente")])
GNDA.add_edges_from([("Asistente","Coordinador"),("Asistente","Planificador")])
GNDA.add_edges_from([("Asistente","Organizador"),("Asistente","Supervisor")])

nx.draw_spring(GNDA, node_size = 5500, node_color = 'y', node_shape = 's', with_labels=True)
plt.draw()
plt.savefig("GNDA.eps")
plt.show()
```

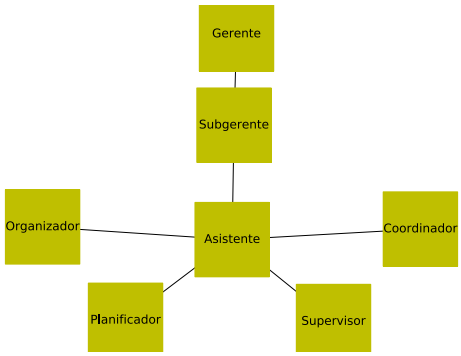


Figura 7: Algoritmo resorte en grafo no dirigido acíclico

8. Algoritmo de acomodo espectral

Este algoritmo posiciona los nodos utilizando los vectores propios del grafo Laplaciano, el diseño muestra posibles agrupaciones de nodos que son una aproximación de la relación de corte [9].

El código en python es el siguiente:

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

MDC = nx.MultiDiGraph()

MDC.add_nodes_from(["E1","E5"])
MDC.add_edges_from([("E1","E2"),("E2","E1"),("E2","E3"),("E3","E2"),("E3","E4"),("E4","E3"),
                    ("E4","E5"),("E5","E4")])

nx.draw_spectral(MDC, node_size = 1000, node_color = 'yellow', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MDC.eps")
plt.show()
```

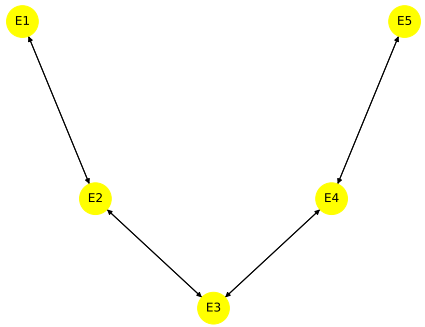


Figura 8: Algoritmo espectral en multigrafo dirigido cíclico

9. Algoritmo de acomodo Fruchterman-Reingold

La idea de este algoritmo de acomodo es considerar una fuerza entre dos nodos cualesquiera, en este algoritmo los nodos están representados por anillos de acero y los arcos son resortes entre ellos. La idea básica es minimizar la energía del sistema moviendo los nodos y cambiando las fuerzas entre ellos. En este algoritmo, la suma de los vectores de fuerza determina en que dirección se debe mover un nodo [4].

El código en python es:

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

MDA = nx.MultiDiGraph()

MDA.add_nodes_from(["C1","C5"])
MDA.add_edges_from([("C3","C4")], color='lightblue', weight=4)
MDA.add_edges_from([("C1","C3"),("C2","C3"),("C3","C4"),("C3","C4"),("C4","C5")], color='black',
    weight=1)

edges = MDA.edges()

colors = []
weight = []

for (u,v,attrib_dict) in list(MDA.edges.data()):
    colors.append(attrib_dict['color'])
    weight.append(attrib_dict['weight'])

color_map = []
for node in MDA:
    if (node == "C3" or node == "C4"):
        color_map.append('yellow')
    else:
        color_map.append('blue')

frl = nx.fruchterman_reingold_layout(MDA, k=0.40, iterations=40)

nx.draw(MDA, pos=frl, edges=edges, edge_color=colors, width=weight, node_size = 1000,
        node_color = color_map, node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MDA.eps")
plt.show()
```

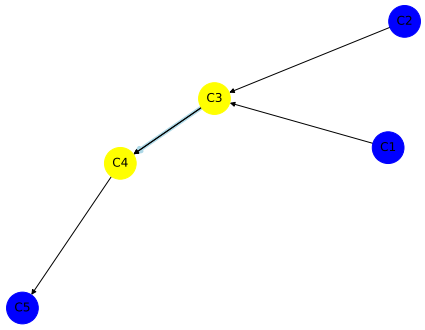


Figura 9: Algoritmo fruchterman-reingold en multigrafo dirigido acíclico

10. Algoritmo de acomodo ForceAtlas2

Es un algoritmo contínuo que permite manipular el grafo mientras se está renderizando. Cuenta con una velocidad de convergencia adaptativa única que permite que la mayoría de los grafos converjan de manera más eficiente. Propone ajustes resumidos, enfocados en qué impacto tiene la forma del grafo (escalado, gravedad,...) [5].

El código en python es el siguiente:

```
import networkx as nx
from fa2 import ForceAtlas2
import matplotlib.pyplot as plt

MNDA = nx.MultiGraph()

MNDA.add_nodes_from(["S1", "S5"])
MNDA.add_edges_from([("S3", "S4"), ("S4", "S5")], color='blue', weight=5)
MNDA.add_edges_from([("S1", "S2"), ("S2", "S3"), ("S2", "S4"), ("S2", "S5"), ("S3", "S4"),
                     ("S4", "S5"),("S3", "S5")], color='black', weight=2)

edges = MNDA.edges()
colors = []
weight = []
for (u, v, attrib_dict) in list(MNDA.edges.data()):
    colors.append(attrib_dict['color'])
    weight.append(attrib_dict['weight'])

#Tomado de: https://github.com/bhargavchippada/forceatlas2/blob/master/README.md
forceatlas2 = ForceAtlas2(
    outboundAttractionDistribution=True,
    linLogMode=False,
    adjustSizes=False,
    edgeWeightInfluence=1.0,

    jitterTolerance=1.0,
    barnesHutOptimize=True,
    barnesHutTheta=1.2,
    multiThreaded=False,

    scalingRatio=2.0,
    strongGravityMode=False,
    gravity=1.0,

    verbose=True)

positions = forceatlas2.forceatlas2_networkx_layout(MNDA, pos=None, iterations=1000)
nx.draw_networkx_nodes(MNDA, positions, node_size=500, with_labels=False, node_color="blue",
    alpha=0.4)
nx.draw_networkx_edges(MNDA, positions, edge_color="black", alpha=0.05)
plt.axis('off')
plt.savefig("MNDA.eps")
plt.show()
```

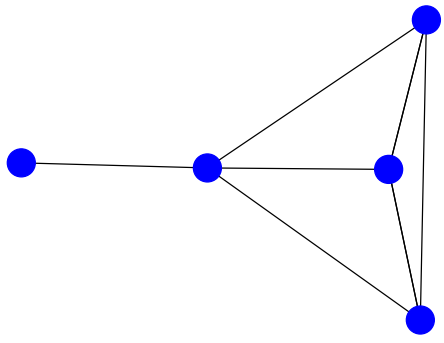


Figura 10: Algoritmo ForceAtlas2 en multigrafo no dirigido acíclico

11. Algoritmo de acomodo resorte

El código en python se muestra a continuación:

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

GDC = nx.DiGraph()

GDC.add_nodes_from(["S1","S5"])
GDC.add_edges_from([("S1","S2"),("S2","S5"),("S3","S4"),("S4","S5"),("S5","S1")])

nx.draw_spring(GDC, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GDC.eps")
plt.show()
```

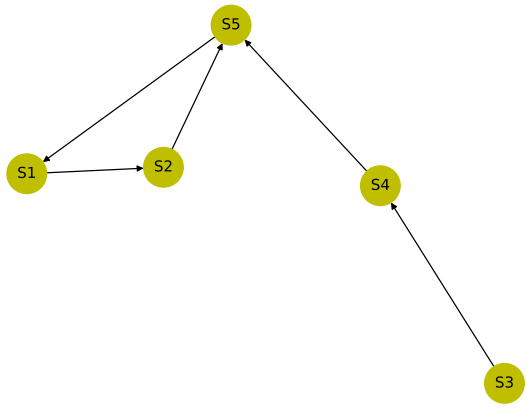


Figura 11: Algoritmo resorte en grafo dirigido cíclico

12. Algoritmo de acomodo circular

El código en python es:

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

MDR = nx.MultiDiGraph()

MDR.add_nodes_from(["D1","D5"])
MDR.add_edges_from([("D1","D2"),("D1","D4"),("D2","D3"),("D3","D2"),("D3","D4"),("D3","D5"),("D4","D5"),
                    ("D5","D5")])

color_map = []
for node in MDR:
    if (node == "D5"):
        color_map.append('yellow')
    else:
        color_map.append('red')

nx.draw_circular(MDR, node_size = 1000, node_color = color_map, node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MDR.eps")
plt.show()
```

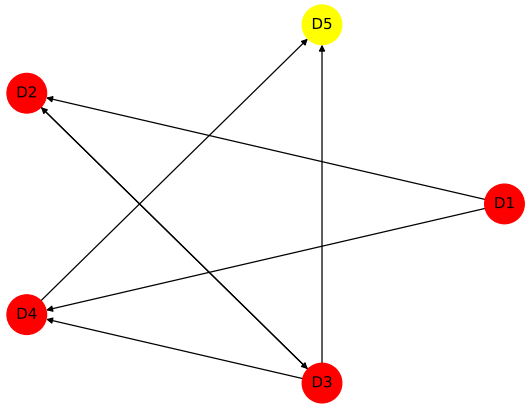


Figura 12: Algoritmo circular en multigrafo dirigido reflexivo

Referencias

[1] Gabor Csardi. Layout as bipartite: Simple two-row layout for bipartite graphs. Last access in 02-17-2019 to urlhttp://rdrr.io/cran/igraph/man/layout as bipartite.htmlheading-5, 2019.

[2] The diagramming company. Developers guide: Analysis and layout. Last access in 02-17-2019 to urlhttp://docs.yworks.com/yfilesflex/doc/dguide-layout/circular layouter.html, 2015.

[3] Simon Dobson. Concepts: networks and geometry. Last access in 02-23-2019 to urlhttp://simondobson.org/2017/04/21/concepts-networks-and-geometry/, 2017.

[4] Sébastien Heymann. Fruchterman reingold. Last access in 02-23-2019 to urlhttp://github.com/gephi/gephi/wiki/Fruchterman-Reingold, 2015.

[5] Mathieu Jacomy. Forceatlas2, the new version of our home-brew layout. Last access in 02-25-2019 to urlhttp://gephi.wordpress.com/2011/06/06/forceatlas2-the-new-version-of-our-home-brew-layout/, 2011.

[6] Networkx. Random layout. Last access in 02-21-2019 to urlhttp://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.drawing.layout.randomlayout.html, 2015.

[7] Networkx. Networkx drawing layout spring layout. Last access in 02-23-2019 to urlhttp://networkx.github.io/documentation/stable/reference/generated/networkx.drawing.layout.spring layout.html, 2018.

[8] Networkx. Networkx drawing layout bipartite layout. Last access in 02-17-2019 to urlhttp://networkx.github.io/documentation/latest/reference/generated/networkx.drawing.layout.bipartite layout.html, 2019.

[9] Networkx. Spectral layout. Last access in 02-23-2019 to urlhttp://networkx.github.io/documentation/latest/reference/generated/networkx.drawing.layout.spectral layout.html, 2019.

[10] Sparx Systems. Spring layout. Last access in 02-23-2019 to urlhttp://sparxsystems.com/enterprisearchitectuserguide/14.0/modelingtools/spring layout.html, 2019.

[11] Andrés Aiello y Rodrigo I. Silveira. Trazado de grafos mediante métodos dirigidos por fuerza. Last access in 02-21-2019 to urlhttp://slideplayer.es/slide/2343740/, 2018.

Tarea 2 Optimización de Flujo en Redes

Orlando Lázaro Ruiloba Torres 5270

3 de junio de 2019

1. Algoritmo de acomodo bipartito

El diseño de este algoritmo se crea colocando primero los vértices en dos filas, de acuerdo con sus tipos. Luego, las posiciones dentro de las filas se optimizan para minimizar los cruces de arcos, utilizando el algoritmo de Sugiyama [1]. Este algoritmo de acomodo actualmente solo funciona en dos dimensiones [8].

El código en python se muestra a continuación:

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

GDA = nx.DiGraph()

GDA.add_nodes_from(["E1","E2","E3"], bipartite=0)
GDA.add_nodes_from(["E4","E5","E6","E7"], bipartite=1)

GDA.add_edges_from([("E1","E4"),("E1","E5"),("E2","E6"),("E2","E7"),("E3","E4"),("E3","E7")])

nx.draw(GDA, pos=nx.bipartite_layout(GDA,["E1","E2","E3"]), node_size = 2000, node_color = 'y',
       node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GDA.eps")
plt.show()
```

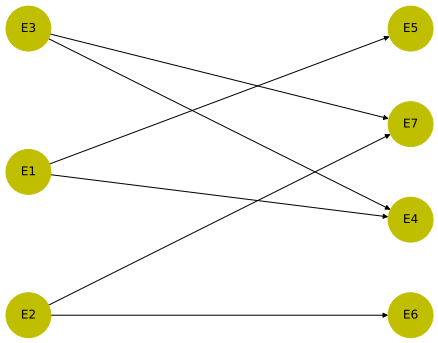


Figura 1: Algoritmo bipartito en grafo dirigido acíclico

2. Algoritmo de acomodo circular

Es un algoritmo de diseño que permite colocar los vértices en topologías de anillo y estrella interconectados. Produce diseños que enfatizan las estructuras de grupo y árbol dentro de una red. Crea particiones de nodo mediante el análisis de la estructura de conectividad de la red, y organiza las particiones como círculos separados. Los círculos en sí están dispuestos en forma de árbol radial [2].

El código en python es:

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

MNDC = nx.MultiGraph()
MNDC.add_nodes_from(["1","9"])

MNDC.add_edges_from([("1","2")], color='lightblue', weight=8)
MNDC.add_edges_from([("2","3"),("3","4"),("4","5"),("5","6"),("6","7"),("7","8"),("8","9"),
                     ("2","1"),("1","6"),("2","7"),("4","9")], color='black', weight=2)

edges = MNDC.edges()

colors = []
weight = []

for (u,v,attrib_dict) in list(MNDC.edges.data()):
    colors.append(attrib_dict['color'])
    weight.append(attrib_dict['weight'])

color_map = []
for node in MNDC:
    if (node == "1" or node == "2"):
        color_map.append('blue')
    else:
        color_map.append('red')

nx.draw_circular(MNDC, edges=edges, edge_color=colors, width=weight, node_size = 1000,
                 node_color = color_map, node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MNDC.eps")
plt.show()
```

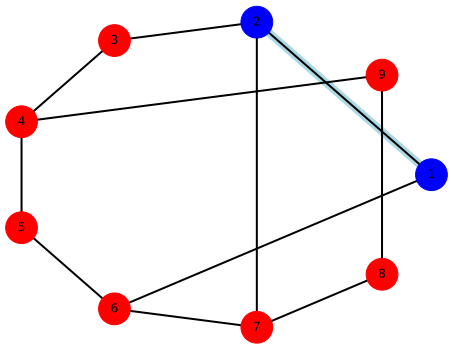


Figura 2: Algoritmo circular en multigrafo no dirigido cíclico

3. Algoritmo de acomodo kamada kawai

Este algoritmo de acomodo se basa en la idea de usar solo fuerzas de resorte entre todos los pares de vértices, el criterio estético que considera es que la distancia entre los vértices debe ser igual a la distancia teórica. Mantiene implícitamente los dos criterios anteriores [11].

El código en python se muestra a continuación:

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

GNDR = nx.Graph()

GNDR.add_nodes_from(["C1","C7"])
GNDR.add_edges_from([("C1","C2"),("C2","C3"),("C3","C4"),("C4","C5"),("C5","C3"),("C3","C6"),
                     ("C6","C7"),("C7","C7")])

color_map = []
for node in GNDR:
    if (node == "C7"):
        color_map.append('blue')
    else:
        color_map.append('red')

layout = nx.kamada_kawai_layout(GNDR)

nx.draw(GNDR, pos=layout, node_size = 1000, node_color = color_map, node_shape = 'o',
     with_labels=True)
plt.draw()
plt.savefig("GNDR.eps")
plt.show()
```

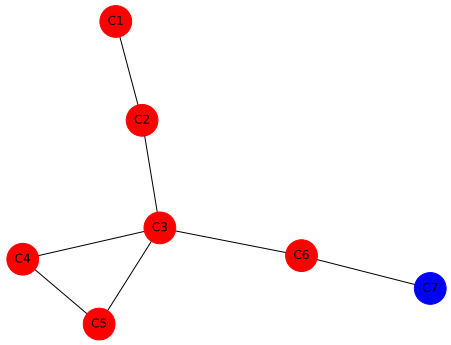


Figura 3: Algoritmo kamada kawai en grafo simple no dirigido reflexivo

4. Algoritmo de acomodo aleatorio

Es un algoritmo que posiciona los vértices uniformemente al azar en el cuadrado unitario. Para cada vértice se genera una posición, al elegir cada una de las coordenadas tenues de la forma mencionada anteriormente en el intervalo [0.0, 1.0] [6].

El código en python es el siguiente:

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

GNDC = nx.Graph()

GNDC.add_node("F")
GNDC.add_nodes_from(["C1","C2","C3","C4"])

GNDC.add_edges_from([("F","C3"),("C3","C4"),("C4","C2"),("C2","C1"),("C1","F")])

nx.draw_random(GNDC, node_size = 2000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GNDC.eps")
plt.show()
```

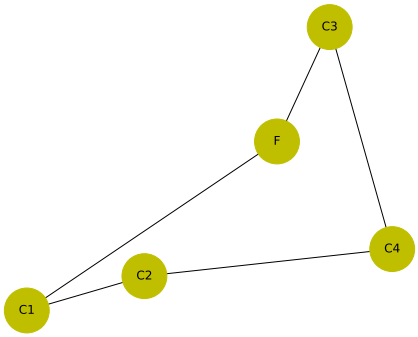


Figura 4: Algoritmo aleatorio en grafo no dirigido cíclico

5. Algoritmo de acomodo espectral

El código en python se muestra a continuación:

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

MNDR = nx.MultiGraph()

MNDR.add_nodes_from(["1","5"])
MNDR.add_edges_from([("1","2"),("2","3"),("2","4"),("2","5"),("3","5"),("4","5"),("3","5"),
    ("4","5"),("5","5")])

color_map = []
for node in MNDR:
    if (node == "5"):
        color_map.append('yellow')
    else:
        color_map.append('red')

nx.draw_spectral(MNDR, node_size = 1000, node_color = color_map, node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MNDR.eps")
plt.show()
```

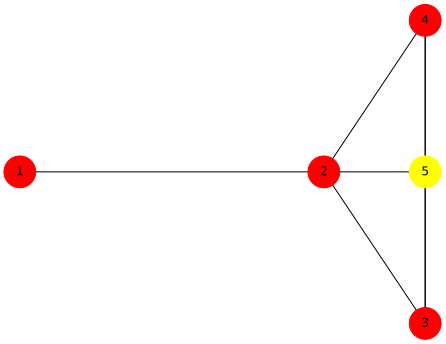


Figura 5: Algoritmo espectral en multigrafo no dirigido reflexivo

6. Algoritmo de acomodo shell

Este algoritmo de acomodo es interesante, ya que le permite al diseñador seleccionar subgrupos de nodos para posicionarlos en círculos concéntricos, en este sentido es menos automático que otros diseños, pero tiene la ventaja crítica de que se puede usar para resaltar características específicas de la red [3].

El código en python es:

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

GDR = nx.DiGraph()

GDR.add_nodes_from(["C1","C5"])
GDR.add_edges_from([("C1","C2"),("C2","C3"),("C3","C4"),("C4","C2"),("C5","C1"),("C5","C5")])

color_map = []
for node in GDR:
    if (node == "C5"):
        color_map.append('blue')
    else:
        color_map.append('red')

nx.draw_shell(GDR, node_size = 2000, node_color = color_map, node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GDR.eps")
plt.show()
```

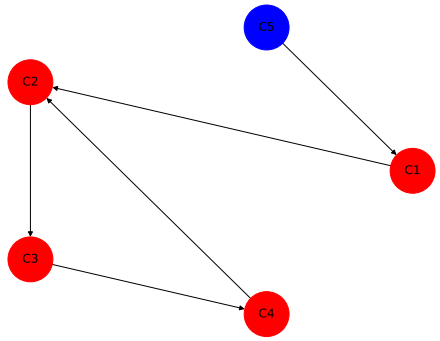


Figura 6: Algoritmo shell en grafo dirigido reflexivo

7. Algoritmo de acomodo resorte

En este caso el algoritmo posiciona los nodos basándose en el algoritmo de fuerza dirigida de Fruchterman-Reingold [7]. Cada elemento se trata como una partícula con una carga eléctrica similar que repele otros elementos. Los conectores actúan como resortes que vuelven a unir los elementos conectados. El algoritmo es bueno para resaltar grupos de objetos relacionados e identificar simetría en el grafo [10].

El código en python se muestra a continuación:

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

GNDA = nx.Graph()

GNDA.add_node("Gerente")
GNDA.add_nodes_from(["Subgerente","Asistente","Coordinador","Planificador","Organizador","Supervisor"])

GNDA.add_edges_from([("Gerente","Subgerente"),("Subgerente","Asistente")])
GNDA.add_edges_from([("Asistente","Coordinador"),("Asistente","Planificador")])
GNDA.add_edges_from([("Asistente","Organizador"),("Asistente","Supervisor")])

nx.draw_spring(GNDA, node_size = 5500, node_color = 'y', node_shape = 's', with_labels=True)
plt.draw()
plt.savefig("GNDA.eps")
plt.show()
```

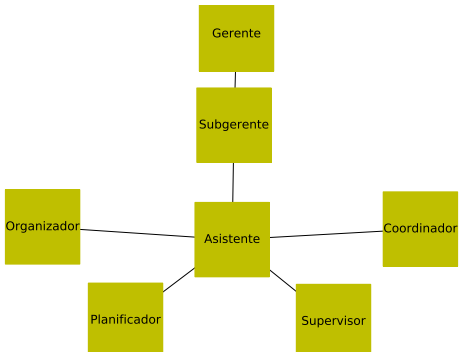


Figura 7: Algoritmo resorte en grafo no dirigido acíclico

8. Algoritmo de acomodo espectral

Este algoritmo posiciona los nodos utilizando los vectores propios del grafo Laplaciano, el diseño muestra posibles agrupaciones de nodos que son una aproximación de la relación de corte [9].

El código en python es el siguiente:

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

MDC = nx.MultiDiGraph()

MDC.add_nodes_from(["E1","E5"])
MDC.add_edges_from([("E1","E2"),("E2","E1"),("E2","E3"),("E3","E2"),("E3","E4"),("E4","E3"),
                    ("E4","E5"),("E5","E4")])

nx.draw_spectral(MDC, node_size = 1000, node_color = 'yellow', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MDC.eps")
plt.show()
```

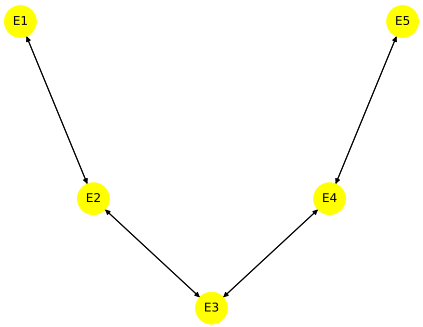


Figura 8: Algoritmo espectral en multigrafo dirigido cíclico

9. Algoritmo de acomodo Fruchterman-Reingold

La idea de este algoritmo de acomodo es considerar una fuerza entre dos nodos cualesquiera, en este algoritmo los nodos están representados por anillos de acero y los arcos son resortes entre ellos. La idea básica es minimizar la energía del sistema moviendo los nodos y cambiando las fuerzas entre ellos. En este algoritmo, la suma de los vectores de fuerza determina en que dirección se debe mover un nodo [4].

El código en python es:

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

MDA = nx.MultiDiGraph()

MDA.add_nodes_from(["C1","C5"])
MDA.add_edges_from([("C3","C4")], color='lightblue', weight=4)
MDA.add_edges_from([("C1","C3"),("C2","C3"),("C3","C4"),("C3","C4"),("C4","C5")], color='black',
    weight=1)

edges = MDA.edges()

colors = []
weight = []

for (u,v,attrib_dict) in list(MDA.edges.data()):
    colors.append(attrib_dict['color'])
    weight.append(attrib_dict['weight'])

color_map = []
for node in MDA:
    if (node == "C3" or node == "C4"):
        color_map.append('yellow')
    else:
        color_map.append('blue')

frl = nx.fruchterman_reingold_layout(MDA, k=0.40, iterations=40)

nx.draw(MDA, pos=frl, edges=edges, edge_color=colors, width=weight, node_size = 1000,
        node_color = color_map, node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MDA.eps")
plt.show()
```

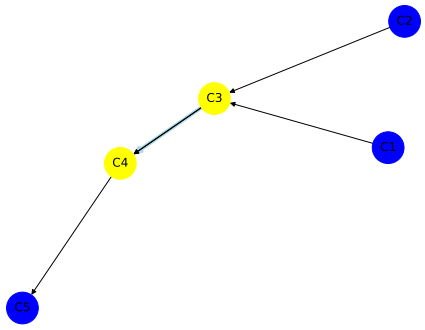


Figura 9: Algoritmo Fruchterman-Reingold en multigrafo dirigido acíclico

10. Algoritmo de acomodo ForceAtlas2

Es un algoritmo contínuo que permite manipular el grafo mientras se está generando. Cuenta con una velocidad de convergencia adaptativa única que permite que la mayoría de los grafos converjan de manera más eficiente. Propone ajustes resumidos, enfocados en qué impacto tiene la forma del grafo (escalado, gravedad,...) [5].

El código en python es el siguiente:

```python
import networkx as nx
from fa2 import ForceAtlas2
import matplotlib.pyplot as plt

MNDA = nx.MultiGraph()

MNDA.add_nodes_from(["S1", "S5"])
MNDA.add_edges_from([("S3", "S4"), ("S4", "S5")], color='blue', weight=5)
MNDA.add_edges_from([("S1", "S2"), ("S2", "S3"), ("S2", "S4"), ("S2", "S5"), ("S3", "S4"),
                     ("S4", "S5"),("S3", "S5")], color='black', weight=2)

edges = MNDA.edges()
colors = []
weight = []
for (u, v, attrib_dict) in list(MNDA.edges.data()):
    colors.append(attrib_dict['color'])
    weight.append(attrib_dict['weight'])

#Tomado de: https://github.com/bhargavchippada/forceatlas2/blob/master/README.md
forceatlas2 = ForceAtlas2(
    outboundAttractionDistribution=True,
    linLogMode=False,
    adjustSizes=False,
    edgeWeightInfluence=1.0,

    jitterTolerance=1.0,
    barnesHutOptimize=True,
    barnesHutTheta=1.2,
    multiThreaded=False,

    scalingRatio=2.0,
    strongGravityMode=False,
    gravity=1.0,

    verbose=True)

positions = forceatlas2.forceatlas2_networkx_layout(MNDA, pos=None, iterations=1000)
nx.draw_networkx_nodes(MNDA, positions, node_size=500, with_labels=False, node_color="blue",
    alpha=0.4)
nx.draw_networkx_edges(MNDA, positions, edge_color="black", alpha=0.05)
plt.axis('off')
plt.savefig("MNDA.eps")
plt.show()
```

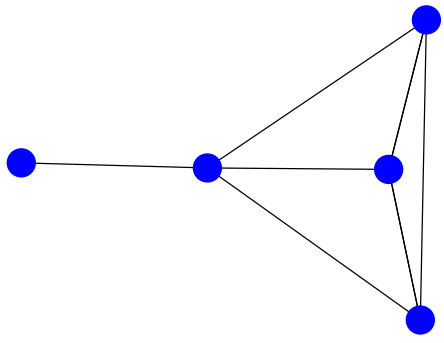


Figura 10: Algoritmo ForceAtlas2 en multigrafo no dirigido acíclico

11. Algoritmo de acomodo resorte

El código en python se muestra a continuación:

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

GDC = nx.DiGraph()

GDC.add_nodes_from(["S1","S5"])
GDC.add_edges_from([("S1","S2"),("S2","S5"),("S3","S4"),("S4","S5"),("S5","S1")])

nx.draw_spring(GDC, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GDC.eps")
plt.show()
```

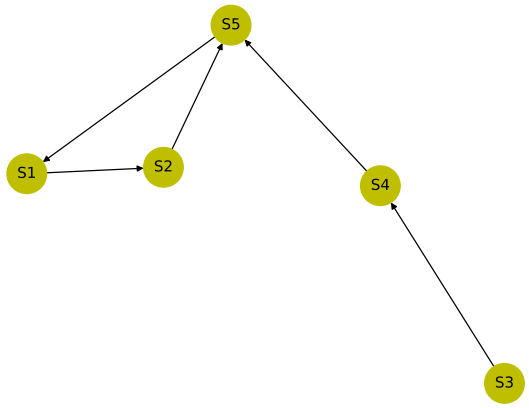


Figura 11: Algoritmo resorte en grafo dirigido cíclico

12. Algoritmo de acomodo circular

El código en python es:

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

MDR = nx.MultiDiGraph()

MDR.add_nodes_from(["D1","D5"])
MDR.add_edges_from([("D1","D2"),("D1","D4"),("D2","D3"),("D3","D2"),("D3","D4"),("D3","D5"),("D4","D5"),
                    ("D5","D5")])

color_map = []
for node in MDR:
    if (node == "D5"):
        color_map.append('yellow')
    else:
        color_map.append('red')

nx.draw_circular(MDR, node_size = 1000, node_color = color_map, node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MDR.eps")
plt.show()
```

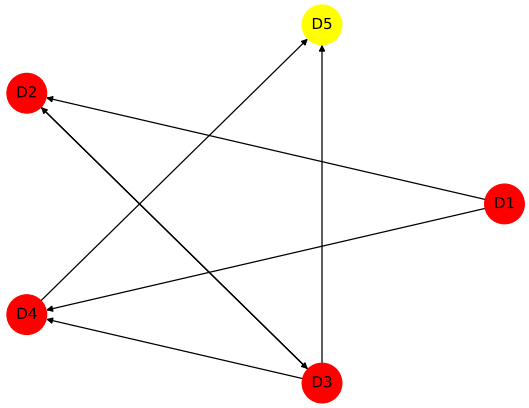


Figura 12: Algoritmo circular en multigrafo dirigido reflexivo

Referencias

[1] Gabor Csardi. Layout as bipartite: Simple two-row layout for bipartite graphs. Last access in 02-17-2019 to http://rdrr.io/cran/igraph/man/layout as bipartite.htmlheading-5, 2019.

[2] The diagramming company. Developers guide: Analysis and layout. Last access in 02-17-2019 to http://docs.yworks.com/yfilesflex/doc/dguide-layout/circular layouter.html, 2015.

[3] Simon Dobson. Concepts: networks and geometry. Last access in 02-23-2019 to http://simondobson.org/2017/04/21/concepts-networks-and-geometry/, 2017.

[4] Sébastien Heymann. Fruchterman reingold. Last access in 02-23-2019 to http://github.com/gephi/gephi/wiki/Fruchterman-Reingold, 2015.

[5] Mathieu Jacomy. Forceatlas2, the new version of our home-brew layout. Last access in 02-25-2019 to http://gephi.wordpress.com/2011/06/06/forceatlas2-the-new-version-of-our-home-brew-layout/, 2011.

[6] Networkx. Random layout. Last access in 02-21-2019 to http://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.drawing.layout.randomlayout.html, 2015.

[7] Networkx. Networkx drawing layout spring layout. Last access in 02-23-2019 to http://networkx.github.io/documentation/stable/reference/generated/networkx.drawing.layout.spring layout.html, 2018.

[8] Networkx. Networkx drawing layout bipartite layout. Last access in 02-17-2019 to http://networkx.github.io/documentation/latest/reference/generated/networkx.drawing.layout.bipartite layout.html, 2019.

[9] Networkx. Spectral layout. Last access in 02-23-2019 to http://networkx.github.io/documentation/latest/reference/generated/networkx.drawing.layout.spectral layout.html, 2019.

[10] Sparx Systems. Spring layout. Last access in 02-23-2019 to http://sparxsystems.com/enterprisearchitectuserguide/14.0/modelingtools/spring layout.html, 2019.

[11] Andrés Aiello y Rodrigo I. Silveira. Trazado de grafos mediante métodos dirigidos por fuerza. Last access in 02-21-2019 to http://slideplayer.es/slide/2343740/, 2018.

3. Conclusión Tarea 2

Con la realización de este ejercicio se tuvo la posibilidad de aprender acerca de los diversos algoritmos de acomodo que ofrece la librería *networkx* de Python, así como cuando es conveniente aplicar cada uno de ellos en dependencia de la instancia (grafo) con la que se esté trabajando.

Tarea 3 Optimización de Flujo en Redes

Orlando Lázaro Ruiloba Torres 5270

19 de marzo de 2019

1. Algoritmo ordenamiento topológico

Este algoritmo devuelve una lista de vértices en ordenamiento topológico, lo cual se traduce en que realiza una permutación no única de los vértices, de modo que una arista de u a v implica que u aparece antes de v en el orden de clasificación topológico [4].

El ordenamiento topológico es una adaptación simple pero útil de una búsqueda en profundidad. Sus pasos se describen a continuación [3]:

1.- Llamar a la búsqueda en profundidad para el grafo en cuestión. La principal razón por la que se invoca a la búsqueda en profundidad, es para calcular los tiempos de finalización para cada uno de los vértices.

2.- Almacenar los vértices en una lista en orden decreciente según el tiempo de finalización.

3.- Devolver la lista ordenada como resultado del ordenamiento topológico.

El código en python se muestra a continuación:

```
#Grafo 1

GDA = nx.DiGraph()

GDA.add_nodes_from(["E1","E2","E3"], bipartite=0)
GDA.add_nodes_from(["E4","E5","E6","E7"], bipartite=1)

GDA.add_edges_from([("E1","E4"),("E1","E5"),("E2","E6"),("E2","E7"),("E3","E4"),("E3","E7")])

replicas = []
for j in range(30):
    start_time = time()
    for i in range (100000):
        t_s = nx.topological_sort(GDA)
    elapsed_time = time() - start_time
    replicas.append(elapsed_time)
print(replicas)
print(len(replicas))

normality_test=stats.shapiro(replicas)
print(normality_test)

hist, bin_edges=np.histogram(replicas_f1,density=True)
first_edge, last_edge = np.min(replicas_f1),np.max(replicas_f1)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_f1, bins=bin_edges, rwidth=0.8, color='orange')
plt.ylabel('Frecuencia')
plt.xlabel('Tiempo (s)')
plt.grid(axis='y', alpha= 0.5)
plt.savefig("Histograma1.eps")
plt.show()

nx.draw(GDA, pos=nx.bipartite_layout(GDA,["E1","E2","E3"]), node_size = 2000, node_color = 'y',
        node_shape = 'o', with_labels=True)
plt.draw()
```

```
plt.savefig("GDA.eps")
plt.show(1)
```

A continuación en la figura 1, se muestra un histograma que ilustra el comportamiento de este algoritmo para las 30 veces en que se efectuaron las 100000 réplicas con sus respectivos tiempos de cómputo:

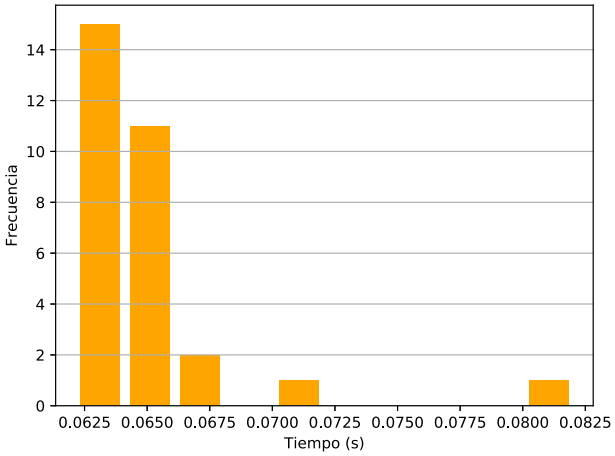


Figura 1: Histograma No.1

2. Algoritmo árbol de expansión mínimo

Un árbol de expansión mínimo es un subgrafo del grafo dado (un árbol) con la suma mínima de los pesos de sus aristas. Este algoritmo genera aristas en un bosque de expansión mínimo de un grafo ponderado no dirigido, este bosque es una unión de los árboles de expansión para cada componente conectado del grafo [6].

El código es el siguiente:

```
#Grafo 2

GNDC = nx.Graph()

GNDC.add_node("F")
GNDC.add_nodes_from(["C1","C2","C3","C4"])

GNDC.add_edge("F","C3", weight = 2)
GNDC.add_edge("C3","C4", weight = 1)
GNDC.add_edge("C4","C2", weight = 4)
GNDC.add_edge("C2","C1", weight = 1)
GNDC.add_edge("C1","F", weight = 2)

replicas_2 = []
for j in range(30):
    start_time = time()
    for i in range (100000):
        mst = nx.minimum_spanning_tree(GNDC)
    elapsed_time = time() - start_time
    replicas_2.append(elapsed_time)
print(replicas_2)
print(len(replicas_2))

normality_test=stats.shapiro(replicas_2)
print(normality_test)

hist, bin_edges=np.histogram(replicas_f2,density=True)
first_edge, last_edge = np.min(replicas_f2),np.max(replicas_f2)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

```

```
plt.hist(replicas_f2, bins=bin_edges, rwidth= 0.8, color= 'orange')
plt.ylabel('Frecuencia')
plt.xlabel('Tiempo (s)')
plt.grid(axis='y', alpha= 0.5)
plt.savefig("Histograma2.eps")
plt.show()

g = nx.random_layout(GNDC)

nx.draw(GNDC, node_size = 2000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GNDC.eps")
plt.show(2)
```

En la figura 2, se muestra el histograma correspondiente:

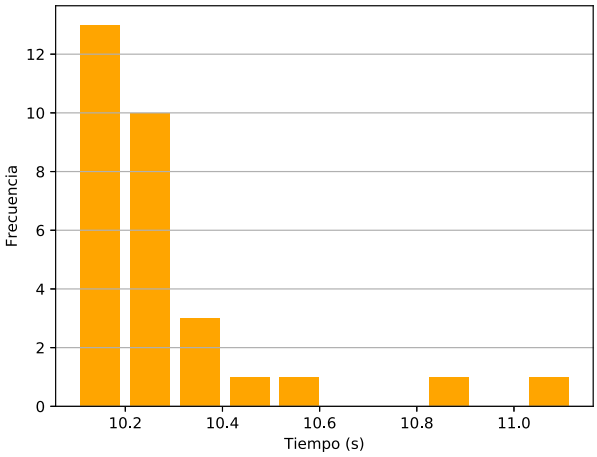


Figura 2: Histograma No.2

3. Algoritmo centralidad de intermediación

Este algoritmo calcula la centralidad de intermediación de cada vértice de un grafo dado [9], esta se encarga de medir en un grafo la tendencia de un vértice único a ser más central que todos los demás vértices en el mismo. Se basa en las diferencias entre la centralidad del vértice más central y la de todos los demás [1].

El código en python es:

```
#Grafo 3

GDC = nx.DiGraph()

GDC.add_nodes_from(["S1","S5"])
GDC.add_edges_from([("S1","S2"),("S2","S5"),("S3","S4"),("S4","S5"),("S5","S1")])

replicas_3 = []
for j in range(30):
    start_time = time()
    for i in range (100000):
        b_c = nx.betweenness_centrality(GDC, normalized=False)
    elapsed_time = time() - start_time
    replicas_3.append(elapsed_time)
print(replicas_3)
print(len(replicas_3))

normality_test=stats.shapiro(replicas_3)
print(normality_test)

hist, bin_edges=np.histogram(replicas_f3,density=True)
first_edge, last_edge = np.min(replicas_f3),np.max(replicas_f3)
```

```
23
24 n_equal_bins = 10
25 bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)
26
27 plt.hist(replicas_f3, bins=bin_edges, rwidth= 0.8, color= 'orange')
28 plt.ylabel('Frecuencia')
29 plt.xlabel('Tiempo (s)')
30 plt.grid(axis='y', alpha= 0.5)
31 plt.savefig("Histograma3.eps")
32 plt.show()
33
34 p = nx.spring_layout(GDC)
35
36 nx.draw(GDC, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
37 plt.draw()
38 plt.savefig("GDC.eps")
39 plt.show(3)
```

El histograma para este algoritmo aparece a continuación:

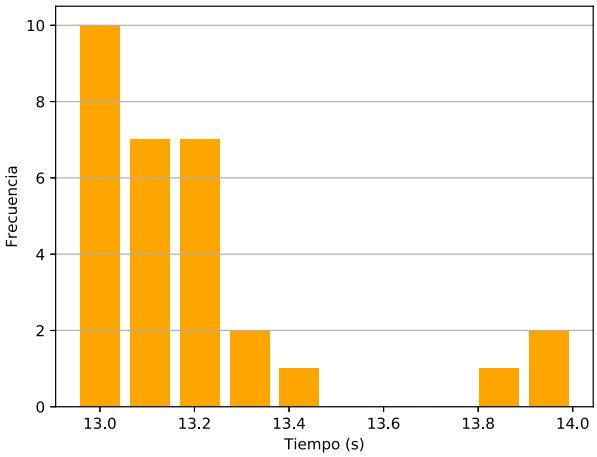


Figura 3: Histograma No.3

4. Algoritmo árbol dfs

Este algoritmo devuelve un árbol orientado, construido a partir de una búsqueda en profundidad desde el origen [7]. La técnica de cruce de DFS (Depth First Search o Búsqueda en Profundidad) de un grafo produce un árbol de expansión como resultado final, es decir, un grafo que no contiene ningún ciclo. En este caso, la estructura de datos se usa con el tamaño máximo del número total de vértices en el grafo para implementar el recorrido DFS [8].

El código en python es el siguiente:

```
1  #Grafo 4
2
3  GNDA = nx.Graph()
4
5  GNDA.add_node("Gerente")
6  GNDA.add_nodes_from(["Subgerente","Asistente","Coordinador","Planificador","Organizador","Supervisor"])
7
8  GNDA.add_edges_from([("Gerente","Subgerente"),("Subgerente","Asistente")])
9  GNDA.add_edges_from([("Asistente","Coordinador"),("Asistente","Planificador")])
10 GNDA.add_edges_from([("Asistente","Organizador"),("Asistente","Supervisor")])
11
12 replicas_4 = []
13 for j in range(30):
14     start_time = time()
15     for i in range (100000):
```

```
        dfs = nx.dfs_tree(GNDA)
    elapsed_time = time() - start_time
    replicas_4.append(elapsed_time)
print(replicas_4)
print(len(replicas_4))

normality_test=stats.shapiro(replicas_4)
print(normality_test)

hist, bin_edges=np.histogram(replicas_f4,density=True)
first_edge, last_edge = np.min(replicas_f4),np.max(replicas_f4)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_f4, bins=bin_edges, rwidth= 0.8, color= 'orange')
plt.ylabel('Frecuencia')
plt.xlabel('Tiempo (s)')
plt.grid(axis='y', alpha= 0.5)
plt.savefig("Histograma4.eps")
plt.show()

p = nx.spring_layout(GNDA)

nx.draw(GNDA, node_size = 5500, node_color = 'y', node_shape = 's', with_labels=True)
plt.draw()
plt.savefig("GNDA.eps")
plt.show(4)
```

En la figura 4 se puede apreciar el histograma referido al algoritmo que se analiza en esta sección:

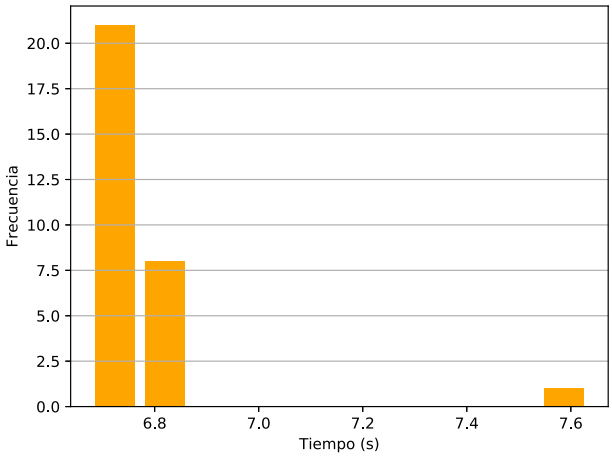


Figura 4: Histograma No.4

5. Algoritmo camarilla máxima

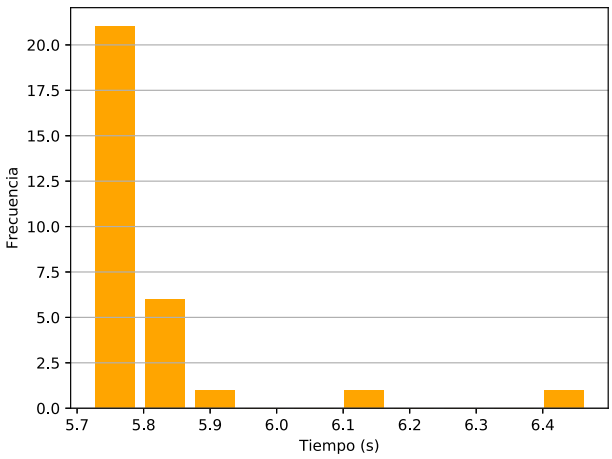
El objetivo que persigue este algoritmo, es encontrar el mayor grupo de nodos en un grafo, de forma tal que todos estén conectados entre sí [2]. Una camarilla en un grafo no dirigido G = (V, E) es un subconjunto del conjunto de vértices C, de manera que por cada dos vértices en C, existe una arista que conecta los dos. La camarilla máxima es aquella que posee el mayor tamaño posible en un grafo dado [5].

El código es:

```
#Grafo 5

MNDR = nx.MultiGraph()

MNDR.add_nodes_from(["1","5"])
MNDR.add_edges_from([("1","2"),("2","3"),("2","4"),("2","5"),("3","5"),("4","5"),("3","5"),
```

```
                    ("4","5"),("5","5")])

color_map = []
for node in MNDR:
    if (node == "5"):
        color_map.append('yellow')
    else:
        color_map.append('red')

replicas_5 = []
for j in range(30):
    start_time = time()
    for i in range (100000):
        cliques = nx.cliques_containing_node(MNDR)
    elapsed_time = time() - start_time
    replicas_5.append(elapsed_time)
print(replicas_5)
print(len(replicas_5))

normality_test=stats.shapiro(replicas_5)
print(normality_test)

hist, bin_edges=np.histogram(replicas_f5,density=True)
first_edge, last_edge = np.min(replicas_f5),np.max(replicas_f5)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_f5, bins=bin_edges, rwidth= 0.8, color= 'orange')
plt.ylabel('Frecuencia')
plt.xlabel('Tiempo (s)')
plt.grid(axis='y', alpha= 0.5)
plt.savefig("Histograma5.eps")
plt.show()

p = nx.spectral_layout(MNDR)

nx.draw(MNDR, node_size = 1000, node_color = color_map, node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MNDR.eps")
```

A continuación, en la figura siguiente se muestra el histograma:



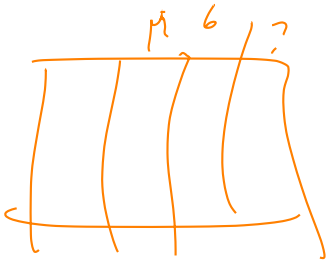


Figura 5: Histograma No.5

Vale la pena recalcar que se realizó la prueba de normalidad Shapiro Wilk, para los datos recolectados del análisis de cada algoritmo por separado, arrojando la misma, que en ninguno de los casos, los datos poseían una distribución normal, por lo cual no se pudo tomar los valores de la media, ni de la desviación estándar como referencia en el análisis.

6. Discusión de conclusiones

A continuación se muestran las dos gráficas de dispersión necesarias para efectuar el análisis del comportamiento de las combinaciones algoritmo-grafo:

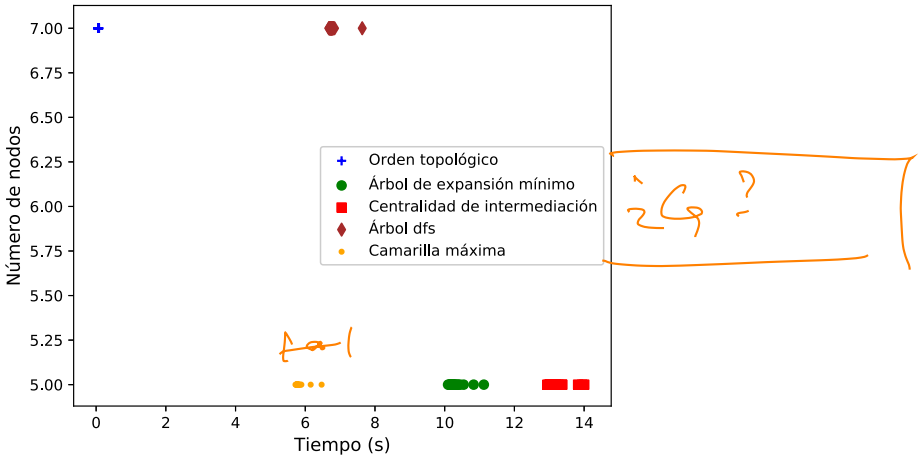


Figura 6: Gráfica No.1

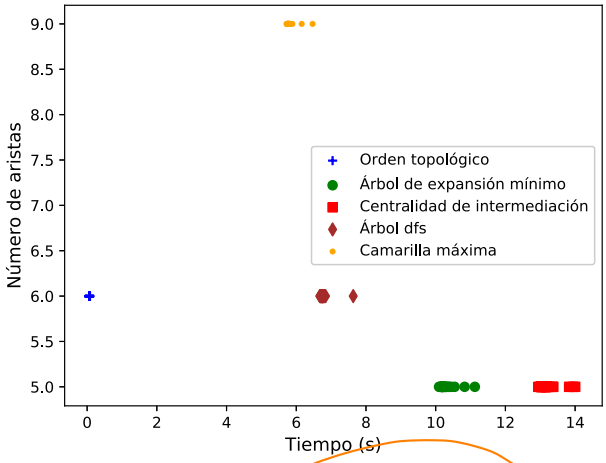


Figura 7: Gráfica No.2

A partir del análisis de las gráficas anteriores, se puede concluir que la combinación algoritmo-grafo que mostró tiempos de ejecución más pequeños, para el número de réplicas realizadas (100000), fue la de orden topológico y la que arrojó mayores tiempos, fue la correspondiente a centralidad de intermediación, en el resto de las combinaciones los tiempos toman valores mayores que 5 y menores a 12 segundos. También se puede apreciar que no existe mucha dispersión entre los tiempos obtenidos para las diferentes combinaciones ya que se encuentran en un rango de valores cercanos.

Referencias

[1] Sunil Kumar Raghavan Unnithan. Balakrishnan Kannan and Madambi Jathavedan. Betweenness centrality in some classes of graphs. Last access in 03-17-2019 to http://www.hindawi.com/journals/ijcom/2014/241723/, 2014.

[2] James MacCaffrey. Algoritmos tabú y el clique máximo. Last access in 03-14-2019 to http://msdn.microsoft.com/es-es/magazine/hh580741.aspx, 2011.

[3] Brad Miller and David Ranum. Problem solving with algorithms and data structures using python. Last access in 03-17-2019 to http://interactivepython.org/runestone/static/pythoned/Graphs/OrdenamientoTopologico.html, 2013.

[4] Networkx. Topological sort. Last access in 03-13-2019 to http://networkx.github.io/documentation/networkx-1.9/reference/generated/networkx.algorithms.dag.topologicalsort.html, 2014.

[5] Networkx. Max clique. Last access in 03-17-2019 to http://networkx.github.io/documentation/networkx-1.9/reference/generated/networkx.algorithms.approximation.clique.maxclique.html, 2014.

[6] Networkx. Minimum spanning tree. Last access in 03-17-2019 to http://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.mst.minimumspanningtree.html, 2015.

[7] Networkx. Dfs tree. Last access in 03-14-2019 to http://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.traversal.depthfirstsearch.dfstree.html, 2015.

[8] Ace Technosoft Team. Graph traversals. Last access in 03-17-2019 to http://acetechnosoft.com/ds/graphtraversal.php, 2018.

[9] GraphStream Team. Betweenness centrality. Last access in 03-14-2019 to http://graphstream-project.org/doc/Algorithms/Betweenness-Centrality/, 2018.

Tarea 3 Optimización de Flujo en Redes

Orlando Lázaro Ruiloba Torres 5270

3 de junio de 2019

1. Algoritmo ordenamiento topológico

Este algoritmo devuelve una lista de vértices en ordenamiento topológico, lo cual se traduce en que realiza una permutación no única de los vértices, de modo que una arista de u a v implica que u aparece antes de v en el orden de clasificación topológico [4].

El ordenamiento topológico es una adaptación simple pero útil de una búsqueda en profundidad. Sus pasos se describen a continuación [3]:

1.- Llamar a la búsqueda en profundidad para el grafo en cuestión. La principal razón por la que se invoca a la búsqueda en profundidad, es para calcular los tiempos de finalización para cada uno de los vértices.

2.- Almacenar los vértices en una lista en orden decreciente según el tiempo de finalización.

3.- Devolver la lista ordenada como resultado del ordenamiento topológico.

El código en python se muestra a continuación:

```
#Grafo 1

GDA = nx.DiGraph()

GDA.add_nodes_from(["E1","E2","E3"], bipartite=0)
GDA.add_nodes_from(["E4","E5","E6","E7"], bipartite=1)

GDA.add_edges_from([("E1","E4"),("E1","E5"),("E2","E6"),("E2","E7"),("E3","E4"),("E3","E7")])

replicas = []
for j in range(30):
    start_time = time()
    for i in range (100000):
        t_s = nx.topological_sort(GDA)
    elapsed_time = time() - start_time
    replicas.append(elapsed_time)
print(replicas)
print(len(replicas))

normality_test=stats.shapiro(replicas)
print(normality_test)

hist, bin_edges=np.histogram(replicas_f1,density=True)
first_edge, last_edge = np.min(replicas_f1),np.max(replicas_f1)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_f1, bins=bin_edges, rwidth=0.8, color='orange')
plt.ylabel('Frecuencia')
plt.xlabel('Tiempo (s)')
plt.grid(axis='y', alpha= 0.5)
plt.savefig("Histograma1.eps")
plt.show()

nx.draw(GDA, pos=nx.bipartite_layout(GDA,["E1","E2","E3"]), node_size = 2000, node_color = 'y',
        node_shape = 'o', with_labels=True)
plt.draw()
```

```
plt.savefig("GDA.eps")
plt.show(1)
```

A continuación en la figura 1, se muestra un histograma que ilustra el comportamiento de este algoritmo para las 30 veces en que se efectuaron las 100000 réplicas con sus respectivos tiempos de cómputo:

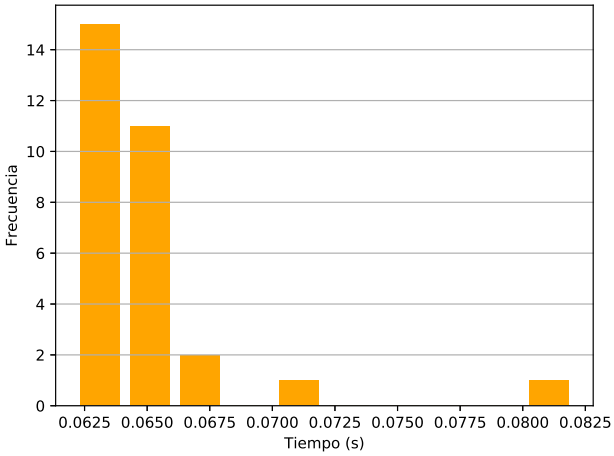


Figura 1: Histograma

2. Algoritmo árbol de expansión mínimo

Un árbol de expansión mínimo es un subgrafo del grafo dado (un árbol) con la suma mínima de los pesos de sus aristas. Este algoritmo genera aristas en un bosque de expansión mínimo de un grafo ponderado no dirigido, este bosque es una unión de los árboles de expansión para cada componente conectado del grafo [6].

El código es el siguiente:

```
#Grafo 2

GNDC = nx.Graph()

GNDC.add_node("F")
GNDC.add_nodes_from(["C1","C2","C3","C4"])

GNDC.add_edge("F","C3", weight = 2)
GNDC.add_edge("C3","C4", weight = 1)
GNDC.add_edge("C4","C2", weight = 4)
GNDC.add_edge("C2","C1", weight = 1)
GNDC.add_edge("C1","F", weight = 2)

replicas_2 = []
for j in range(30):
    start_time = time()
    for i in range (100000):
        mst = nx.minimum_spanning_tree(GNDC)
    elapsed_time = time() - start_time
    replicas_2.append(elapsed_time)
print(replicas_2)
print(len(replicas_2))

normality_test=stats.shapiro(replicas_2)
print(normality_test)

hist, bin_edges=np.histogram(replicas_f2,density=True)
first_edge, last_edge = np.min(replicas_f2),np.max(replicas_f2)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

```

```
plt.hist(replicas_f2, bins=bin_edges, rwidth= 0.8, color= 'orange')
plt.ylabel('Frecuencia')
plt.xlabel('Tiempo (s)')
plt.grid(axis='y', alpha= 0.5)
plt.savefig("Histograma2.eps")
plt.show()

g = nx.random_layout(GNDC)

nx.draw(GNDC, node_size = 2000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GNDC.eps")
plt.show(2)
```

En la figura 2, se muestra el histograma correspondiente:

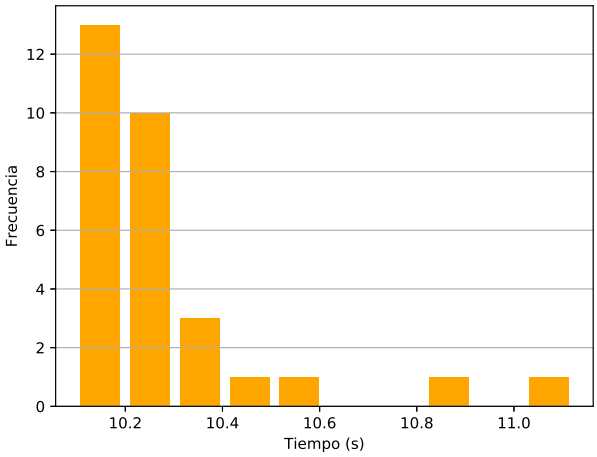


Figura 2: Histograma

3. Algoritmo centralidad de intermediación

Este algoritmo calcula la centralidad de intermediación de cada vértice de un grafo dado [9], esta se encarga de medir en un grafo la tendencia de un vértice único a ser más central que todos los demás vértices en el mismo. Se basa en las diferencias entre la centralidad del vértice más central y la de todos los demás [1].

El código en python es:

```
#Grafo 3

GDC = nx.DiGraph()

GDC.add_nodes_from(["S1","S5"])
GDC.add_edges_from([("S1","S2"),("S2","S5"),("S3","S4"),("S4","S5"),("S5","S1")])

replicas_3 = []
for j in range(30):
    start_time = time()
    for i in range (100000):
        b_c = nx.betweenness_centrality(GDC, normalized=False)
    elapsed_time = time() - start_time
    replicas_3.append(elapsed_time)
print(replicas_3)
print(len(replicas_3))

normality_test=stats.shapiro(replicas_3)
print(normality_test)

hist, bin_edges=np.histogram(replicas_f3,density=True)
first_edge, last_edge = np.min(replicas_f3),np.max(replicas_f3)
```

```
23
24 n_equal_bins = 10
25 bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)
26
27 plt.hist(replicas_f3, bins=bin_edges, rwidth= 0.8, color= 'orange')
28 plt.ylabel('Frecuencia')
29 plt.xlabel('Tiempo (s)')
30 plt.grid(axis='y', alpha= 0.5)
31 plt.savefig("Histograma3.eps")
32 plt.show()
33
34 p = nx.spring_layout(GDC)
35
36 nx.draw(GDC, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
37 plt.draw()
38 plt.savefig("GDC.eps")
39 plt.show(3)
```

El histograma para este algoritmo aparece a continuación:

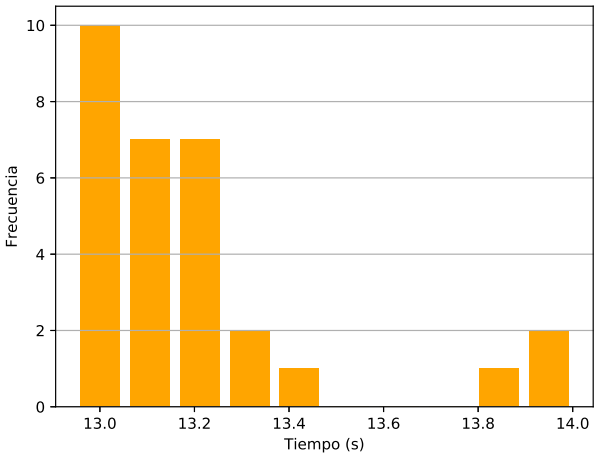


Figura 3: Histograma

4. Algoritmo árbol dfs

Este algoritmo devuelve un árbol orientado, construido a partir de una búsqueda en profundidad desde el origen [7]. La técnica de cruce de DFS (Depth First Search o Búsqueda en Profundidad) de un grafo produce un árbol de expansión como resultado final, es decir, un grafo que no contiene ningún ciclo. En este caso, la estructura de datos se usa con el tamaño máximo del número total de vértices en el grafo para implementar el recorrido DFS [8].

El código en python es el siguiente:

```
1  #Grafo 4
2
3  GNDA = nx.Graph()
4
5  GNDA.add_node("Gerente")
6  GNDA.add_nodes_from(["Subgerente","Asistente","Coordinador","Planificador","Organizador","Supervisor"])
7
8  GNDA.add_edges_from([("Gerente","Subgerente"),("Subgerente","Asistente")])
9  GNDA.add_edges_from([("Asistente","Coordinador"),("Asistente","Planificador")])
10 GNDA.add_edges_from([("Asistente","Organizador"),("Asistente","Supervisor")])
11
12 replicas_4 = []
13 for j in range(30):
14     start_time = time()
15     for i in range (100000):
```

```
        dfs = nx.dfs_tree(GNDA)
    elapsed_time = time() - start_time
    replicas_4.append(elapsed_time)
print(replicas_4)
print(len(replicas_4))

normality_test=stats.shapiro(replicas_4)
print(normality_test)

hist, bin_edges=np.histogram(replicas_f4,density=True)
first_edge, last_edge = np.min(replicas_f4),np.max(replicas_f4)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_f4, bins=bin_edges, rwidth= 0.8, color= 'orange')
plt.ylabel('Frecuencia')
plt.xlabel('Tiempo (s)')
plt.grid(axis='y', alpha= 0.5)
plt.savefig("Histograma4.eps")
plt.show()

p = nx.spring_layout(GNDA)

nx.draw(GNDA, node_size = 5500, node_color = 'y', node_shape = 's', with_labels=True)
plt.draw()
plt.savefig("GNDA.eps")
plt.show(4)
```

En la figura 4 se puede apreciar el histograma referido al algoritmo que se analiza en esta sección:

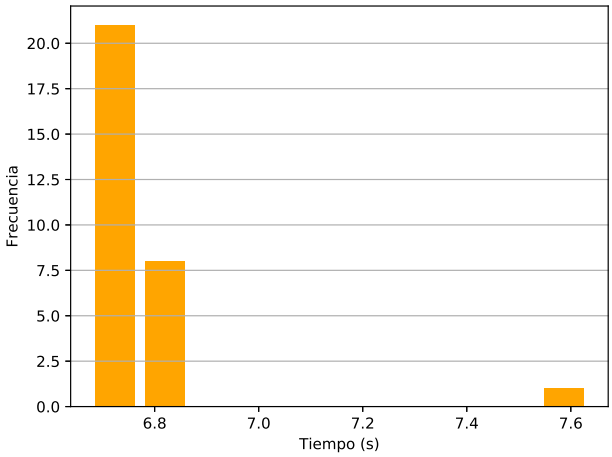


Figura 4: Histograma

5. Algoritmo camarilla máxima

El objetivo que persigue este algoritmo, es encontrar el mayor grupo de nodos en un grafo, de forma tal que todos estén conectados entre sí [2]. Una camarilla en un grafo no dirigido $G = (V, E)$ es un subconjunto del conjunto de vértices C, de manera que por cada dos vértices en C, existe una arista que conecta los dos. La camarilla máxima es aquella que posee el mayor tamaño posible en un grafo dado [5].

El código es:

```
#Grafo 5

MNDR = nx.MultiGraph()

MNDR.add_nodes_from(["1","5"])
MNDR.add_edges_from([("1","2"),("2","3"),("2","4"),("2","5"),("3","5"),("4","5"),("3","5"),
```

```
                    ("4","5"),("5","5")])

color_map = []
for node in MNDR:
    if (node == "5"):
        color_map.append('yellow')
    else:
        color_map.append('red')

replicas_5 = []
for j in range(30):
    start_time = time()
    for i in range (100000):
        cliques = nx.cliques_containing_node(MNDR)
    elapsed_time = time() - start_time
    replicas_5.append(elapsed_time)
print(replicas_5)
print(len(replicas_5))

normality_test=stats.shapiro(replicas_5)
print(normality_test)

hist, bin_edges=np.histogram(replicas_f5,density=True)
first_edge, last_edge = np.min(replicas_f5),np.max(replicas_f5)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_f5, bins=bin_edges, rwidth= 0.8, color= 'orange')
plt.ylabel('Frecuencia')
plt.xlabel('Tiempo (s)')
plt.grid(axis='y', alpha= 0.5)
plt.savefig("Histograma5.eps")
plt.show()

p = nx.spectral_layout(MNDR)

nx.draw(MNDR, node_size = 1000, node_color = color_map, node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MNDR.eps")
```

A continuación, en la figura siguiente se muestra el histograma:

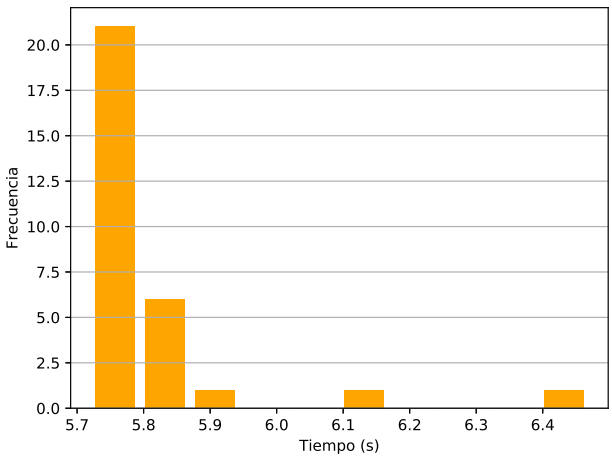


Figura 5: Histograma

Vale la pena recalcar que se realizó la prueba de normalidad Shapiro Wilk, para los datos recolectados del análisis de cada algoritmo por separado, arrojando la misma, que en ninguno de los casos, los datos poseían una distribución normal, por lo cual no se pudo tomar los valores de la media, ni de la desviación estándar como referencia en el análisis.

6. Discusión de conclusiones

A continuación se muestran las dos gráficas de dispersión necesarias (**ver Figuras 6 y 7**) para efectuar el análisis del comportamiento de las combinaciones algoritmo-grafo:

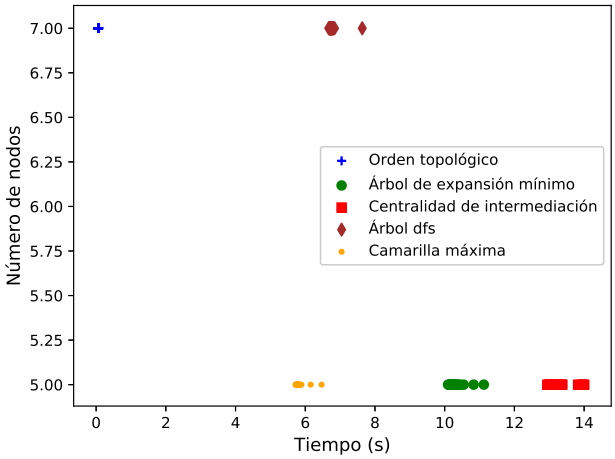


Figura 6

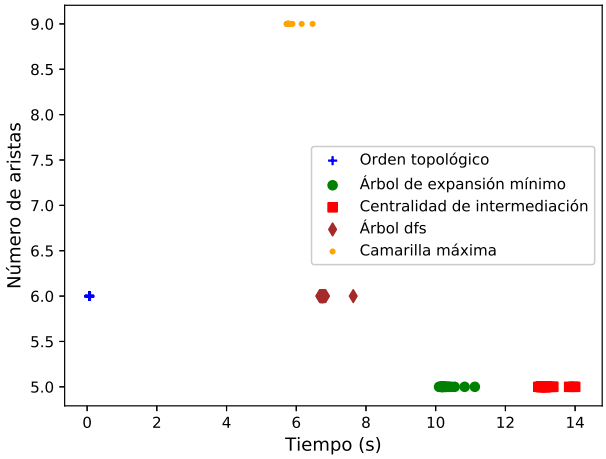


Figura 7

A partir del análisis de las gráficas anteriores, se puede concluir que la combinación algoritmo-grafo que mostró tiempos de ejecución más pequeños, para el número de réplicas realizadas (100000), fue la de orden topológico y la que arrojó mayores tiempos, fue la correspondiente a centralidad de intermediación, en el resto de las combinaciones los tiempos toman valores mayores que 5 y menores a 12 segundos. También se puede apreciar que no existe mucha dispersión entre los tiempos obtenidos para las diferentes combinaciones ya que se encuentran en un rango de valores cercanos.

Referencias

[1] Sunil Kumar Raghavan Unnithan. Balakrishnan Kannan and Madambi Jathavedan. Betweenness centrality in some classes of graphs. Last access in 03-17-2019 to http://www.hindawi.com/journals/ijcom/2014/241723/, 2014.

[2] James MacCaffrey. Algoritmos tabú y el clique máximo. Last access in 03-14-2019 to http://msdn.microsoft.com/es-es/magazine/hh580741.aspx, 2011.

[3] Brad Miller and David Ranum. Problem solving with algorithms and data structures using python. Last access in 03-17-2019 to http://interactivepython.org/runestone/static/pythoned/Graphs/OrdenamientoTopologico.html, 2013.

[4] Networkx. Topological sort. Last access in 03-13-2019 to http://networkx.github.io/documentation/networkx-1.9/reference/generated/networkx.algorithms.dag.topologicalsort.html, 2014.

[5] Networkx. Max clique. Last access in 03-17-2019 to http://networkx.github.io/documentation/networkx-1.9/reference/generated/networkx.algorithms.approximation.clique.maxclique.html, 2014.

[6] Networkx. Minimum spanning tree. Last access in 03-17-2019 to http://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.mst.minimumspanningtree.html, 2015.

[7] Networkx. Dfs tree. Last access in 03-14-2019 to http://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.traversal.depthfirstsearch.dfstree.html, 2015.

[8] Ace Technosoft Team. Graph traversals. Last access in 03-17-2019 to http://acetechnosoft.com/ds/graphtraversal.php, 2018.

[9] GraphStream Team. Betweenness centrality. Last access in 03-14-2019 to http://graphstream-project.org/doc/Algorithms/Betweenness-Centrality/, 2018.

4. Conclusión Tarea 3

Con la realización de este ejercicio y con el empleo por supuesto de técnicas estadísticas como la prueba de normalidad de los datos, se pudo apreciar de manera más clara y concisa la influencia que tiene la combinación algoritmo - grafo en los tiempos de ejecución de los mismos para un número de replicas determinado.

Tarea 4 Optimización de Flujo en Redes

Orlando Lázaro Ruiloba Torres 5270

2 de abril de 2019

1. Breve explicación del objetivo que se persigue

Para la realización de este trabajo se seleccionaron tres métodos generadores de grafos, de los cuales en las secciones siguientes se expone una breve explicación, a partir de estos se generaron 16 grafos de cuatro órdenes distintos en escala logarítmica, a los cuales se les asignaron pesos no negativos y normalmente distribuidos con el fin de poder implementar tres de los algoritmos de networkx para flujo máximo seleccionados, los cuales se ejecutaron cinco veces para los cinco pares fuente - sumidero elegidos, con el fín de determinar mediante el uso de técnicas estadísticas el efecto o interacción existente por parte los métodos generadores, los algoritmos usados, el orden y la densidad de los grafos en el tiempo de ejecución.

A continuación se ofrece una breve explicación de cada uno de los métodos y algoritmos seleccionados:

2. Método generador de grafos: Árbol lleno r-ario

Este método crea un árbol r-ario completo de n vértices. Todos los vértices que no son hojas tienen exactamente r hijos y todos los niveles están llenos excepto por alguna posición más a la derecha del nivel inferior (si falta una hoja en el nivel inferior, entonces también están todas las hojas a su derecha [2].

3. Método generador de grafos: Grafo completo

Este método devuelve un grafo completo K, con n vértices, en el caso de que n sea un número entero, los vértices son de rango n, por otro lado si n es un contenedor de vértices, estos aparecen en el grafo [3]. Un grafo completo, es un grafo en el que cada vértice comparte una arista con cada uno de los otros vértices. Si se trata de un grafo dirigido, las aristas deben existir siempre en ambas direcciones [7].

4. Método generador de grafos: Grafo rueda

Este método devuelve un grafo rueda. El grafo de la rueda consiste en un vértice central conectado a un ciclo de $n - 1$ vértices [4] y para el cual cada vértice del grafo en el ciclo está conectado a otro [8].

5. Algoritmo: Valor de flujo máximo

Este algoritmo consiste en determinar cuál es la tasa mayor a la cual un determinado material puede ser transportado de la fuente al sumidero sin violar ninguna restricción de capacidad [1].

6. Algoritmo: Corte mínimo

Este algoritmo calcula el valor del corte mínimo y de la partición de vértice. Utiliza el teorema de corte mínimo de flujo máximo, es decir, la capacidad de un corte de capacidad mínima es igual al valor de flujo de un flujo máximo [5].

7. Algoritmo: Valor de corte mínimo

Este algoritmo es muy similar al anterior, pero tiene la tipicidad de que solo calcula el valor del corte mínimo. Utiliza el teorema de corte mínimo de flujo máximo, es decir, la capacidad de un corte de capacidad mínima es igual al valor de flujo de un flujo máximo [6].

8. Discusión de conclusiones

A continuación se muestran las cuatro gráficas de caja necesarias para efectuar el análisis del comportamiento del efecto que tienen sobre el tiempo de ejecución, los tres métodos generadores de grafos, los tres algoritmos seleccionados, el orden de cada grafo y la densidad de los mismos:

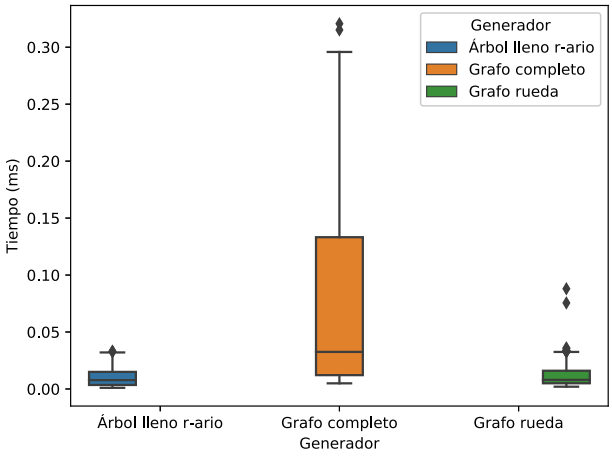


Figura 1: Efecto de los métodos generadores de grafos seleccionados sobre el tiempo de ejecución

Como se puede apreciar en la Figura 1, el método generador de grafos que más tardó en ejecutarse fue el de Grafo completo, siendo este el que mayor efecto tuvo sobre el tiempo de ejecución, en los otros dos casos los tiempos fueron muy similares.

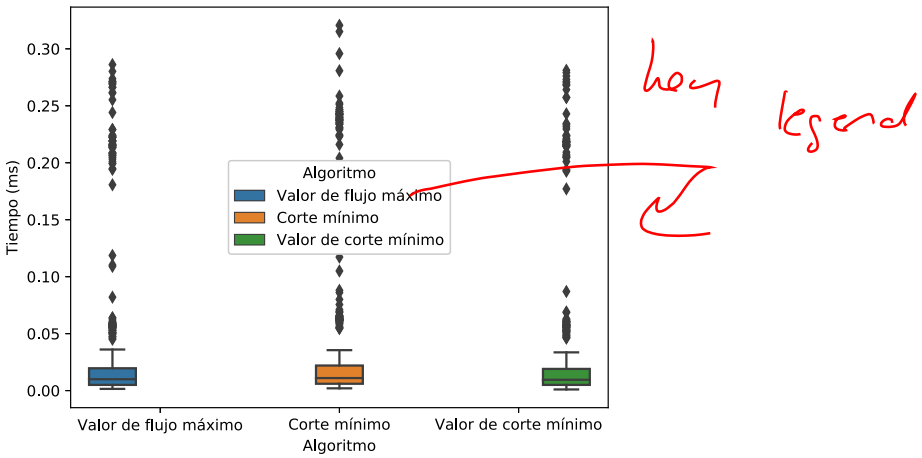


Figura 2: Efecto de los algoritmos seleccionados sobre el tiempo de ejecución

A partir de la Figura 2, se puede concluir que todos los algoritmos seleccionados mostraron tiempos de ejecución similares, por lo que a través de la realización del análisis estadístico (ANOVA) que se realiza con posterioridad se podra concluir de manera más certera el posible efecto.

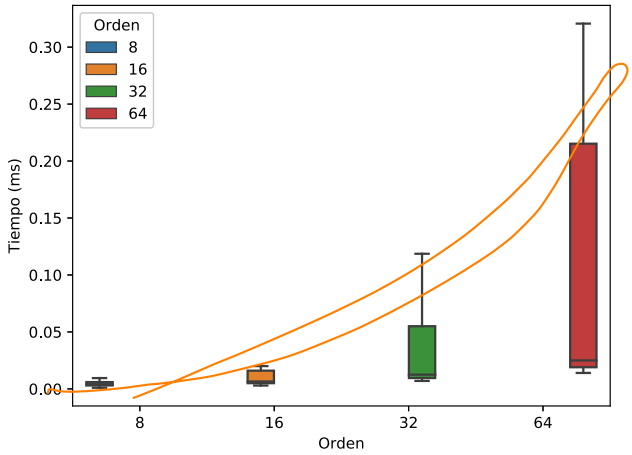


Figura 3: Efecto del orden del grafo sobre el tiempo de ejecución

Como se puede observar en la Figura 3, a medida que aumenta el orden del grafo aumenta también el tiempo de ejecución, por lo que se puede concluir que son directamente proporcionales y que definitivamente el orden seleccionado va a tener un efecto considerable en el tiempo de ejecución.

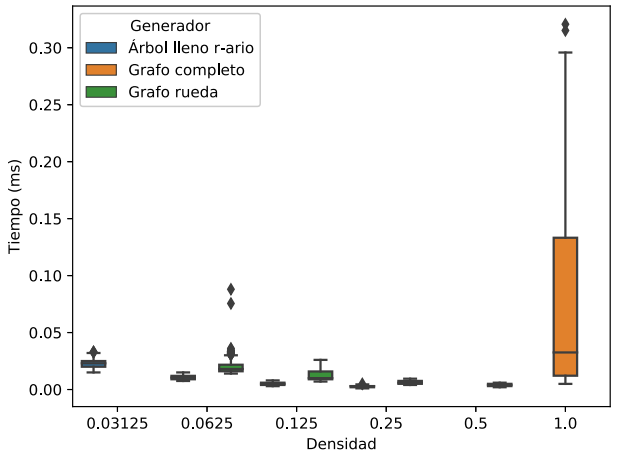


Figura 4: Efecto de la densidad del grafo sobre el tiempo de ejecución

Como se puede apreciar a partir de la figura anterior, la variación de los tiempos de ejecución es muy pequeña, sin embargo cuando la densidad llega a 1, el valor de este se incrementa de forma considerable, lo que permite afirmar que la densidad del grafo sí va a repercutir en el tiempo de ejecución.

Seguidamente se muestra un cuadro en el que se exponen los resultados del análisis de varianza (ANOVA):

	sum sq	**df**	**F**	**PR(>F)**
Generador	1.394225e-10	2.0	3.931140e-07	1.000000
Algoritmo	4.165406e-03	2.0	1.174473e+01	0.000009
Orden	-1.289626e-11	1.0	-7.272428e-08	1.000000
Densidad	1.511910e-10	1.0	8.525926e-07	0.999263
Generador:Algoritmo	2.385267e-04	4.0	3.362736e-01	0.853637
Generador:Orden	9.396538e-07	2.0	2.649437e-03	0.997354
Generador:Densidad	1.879308e-06	2.0	5.298874e-03	0.941979
Algoritmo:Densidad	6.478017e-04	2.0	1.826534e+00	0.161272
Orden:Densidad	9.396538e-07	1.0	5.298874e-03	0.941979
Residual	3.160036e-01	1782.0		

Cuadro 1: Resultados obtenidos ANOVA

A partir de los resultados expuestos anteriormente se puede concluir que para el caso de el método generador, el orden del grafo y la densidad, como su $p$valor > 0.05, el tiempo de ejecución va a sufrir variaciones en dependencia del método elegido, el orden asignado y la densidad que posea el grafo en cuestión, lo cual no sucede para el caso del algoritmo seleccionado, el cual no va a tener ninguna injerencia o efecto sobre el tiempo de ejecución.

Referencias

[1] Agustín J. González. Estructura de datos y algoritmos. Last access in 04-01-2019 to http://profesores.elo.utfsm.cl/ agv/elo320/01and02/redesDeFlujo/maximumFlow.pdf, 2002.

[2] Networkx. Networkx generators classic full_rary_tree. Last access in 04-01-2019 to http://networkx.github.io/documentation/stable/reference /generated/networkx.generators.classic.full_rary_tree.html#networkx.generators.classic.full_rary_tree, 2018.

[3] Networkx. Networkx generators classic complete_graph. Last access in 04-01-2019 to http://networkx.github.io/documentation/stable/reference /generated/networkx.generators.classic.complete_graph.html#networkx.generators.classic.complete_graph, 2018.

[4] Networkx. Networkx generators classic wheel_graph. Last access in 04-01-2019 to http://networkx.github.io/documentation/stable/reference /generated/networkx.generators.classic.wheel_graph.html#networkx.generators.classic.wheel_graph, 2018.

[5] Networkx. Networkx algorithms flow minimum_cut. Last access in 04-01-2019 to http://networkx.github.io/documentation/stable/reference /algorithms/generated/networkx.algorithms.flow.minimum_cut.html#networkx.algorithms.flow.minimum_cut, 2018.

[6] Networkx. Networkx algorithms flow minimum_cut_value. Last access in 04-01-2019 to http://networkx.github.io/documentation/stable/reference /algorithms/generated/networkx.algorithms.flow.minimum_cut_value.html#networkx.algorithms.flow.minimum_cut_value, 2018.

[7] Tim Shearouse. Class completegraphgenerator. Last access in 04-01-2019 to http://jgrapht.org/javadoc/org/jgrapht/generate/CompleteGraphGenerator.html, 2018.

[8] MathWorld Team. Wheel graph. Last access in 04-01-2019 to http://mathworld.wolfram.com/WheelGraph.html, 2019.

Tarea 4 Optimización de Flujo en Redes

Orlando Lázaro Ruiloba Torres 5270

3 de junio de 2019

1. Breve explicación del objetivo que se persigue

Para la realización de este trabajo se seleccionaron tres métodos generadores de grafos, de los cuales en las secciones siguientes se expone una breve explicación, a partir de estos se generaron diez grafos de cuatro órdenes distintos en escala logarítmica, a los cuales se les asignaron pesos no negativos y normalmente distribuidos con el fin de poder implementar tres de los algoritmos de *networkx* para flujo máximo seleccionados, los cuales se ejecutaron cinco veces para los cinco pares fuente − sumidero elegidos, con el fín de determinar mediante el uso de técnicas estadísticas el efecto o interacción existente por parte los métodos generadores, los algoritmos usados, el orden y la densidad de los grafos en el tiempo de ejecución.

A continuación se ofrece una breve explicación de cada uno de los métodos y algoritmos seleccionados:

2. Método generador de grafos: Árbol lleno r-ario

Este método crea un árbol r-ario completo de n vértices. Todos los vértices que no son hojas tienen exactamente r hijos y todos los niveles están llenos excepto por alguna posición más a la derecha del nivel inferior (si falta una hoja en el nivel inferior, entonces también están todas las hojas a su derecha [2].

3. Método generador de grafos: Grafo completo

Este método devuelve un grafo completo K, con n vértices, en el caso de que n sea un número entero, los vértices son de rango n, por otro lado si n es un contenedor de vértices, estos aparecen en el grafo [3]. Un grafo completo, es un grafo en el que cada vértice comparte una arista con cada uno de los otros vértices. Si se trata de un grafo dirigido, las aristas deben existir siempre en ambas direcciones [7].

4. Método generador de grafos: Grafo rueda

Este método devuelve un grafo rueda. El grafo de la rueda consiste en un vértice central conectado a un ciclo de $n - 1$ vértices [4] y para el cual cada vértice del grafo en el ciclo está conectado a otro [8].

5. Algoritmo: Valor de flujo máximo

Este algoritmo consiste en determinar cuál es la tasa mayor a la cual un determinado material puede ser transportado de la fuente al sumidero sin violar ninguna restricción de capacidad [1].

6. Algoritmo: Corte mínimo

Este algoritmo calcula el valor del corte mínimo y de la partición de vértice. Utiliza el teorema de corte mínimo de flujo máximo, es decir, la capacidad de un corte de capacidad mínima es igual al valor de flujo de un flujo máximo [5].

7. Algoritmo: Valor de corte mínimo

Este algoritmo es muy similar al anterior, pero tiene la tipicidad de que solo calcula el valor del corte mínimo. Utiliza el teorema de corte mínimo de flujo máximo, es decir, la capacidad de un corte de capacidad mínima es igual al valor de flujo de un flujo máximo [6].

8. Discusión de conclusiones

A continuación se muestran las cuatro gráficas de caja necesarias para efectuar el análisis del comportamiento del efecto que tienen sobre el tiempo de ejecución, los tres métodos generadores de grafos, los tres algoritmos seleccionados, el orden de cada grafo y la densidad de los mismos:

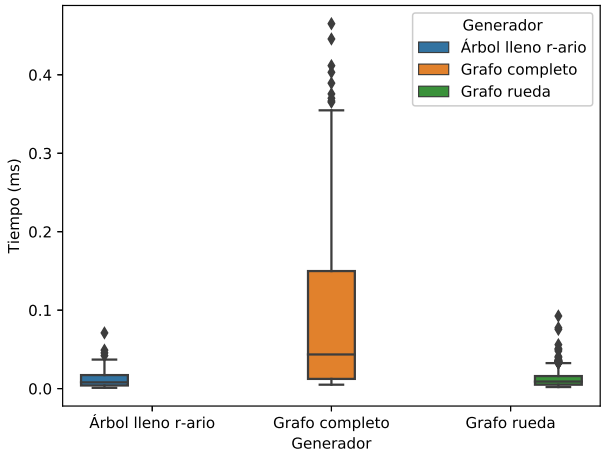


Figura 1: Efecto de los métodos generadores de grafos seleccionados sobre el tiempo de ejecución

Como se puede apreciar en la Figura 1, el método generador de grafos que más tardó en ejecutarse fue el de grafo completo, siendo este el que mayor efecto tuvo sobre el tiempo de ejecución, en los otros dos casos los tiempos fueron muy similares.

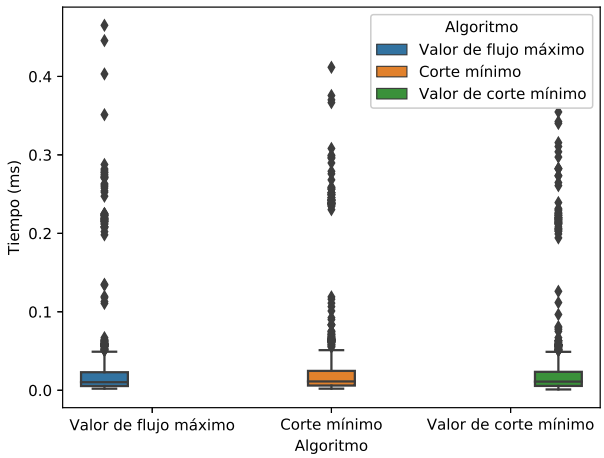


Figura 2: Efecto de los algoritmos seleccionados sobre el tiempo de ejecución

A partir de la Figura 2, se puede concluir que todos los algoritmos seleccionados mostraron tiempos de ejecución similares, por lo que a través de la realización del análisis estadístico (ANOVA) que se realiza con posterioridad se podra concluir de manera más certera el posible efecto.

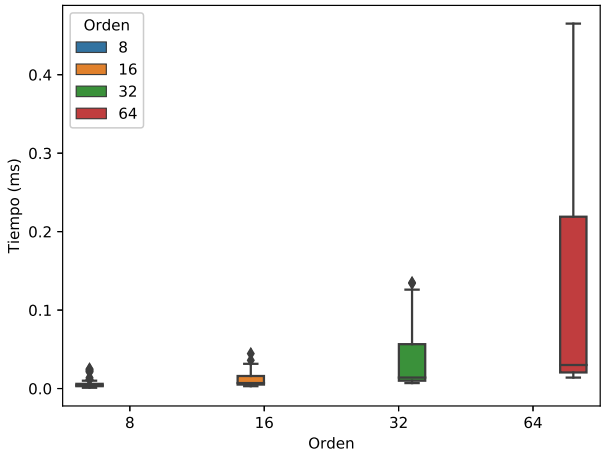


Figura 3: Efecto del orden del grafo sobre el tiempo de ejecución

Como se puede observar en la Figura 3, a medida que aumenta el orden del grafo aumenta también el tiempo de ejecución, por lo que se puede concluir que son directamente proporcionales y que definitivamente el orden seleccionado va a tener un efecto considerable en el tiempo de ejecución.

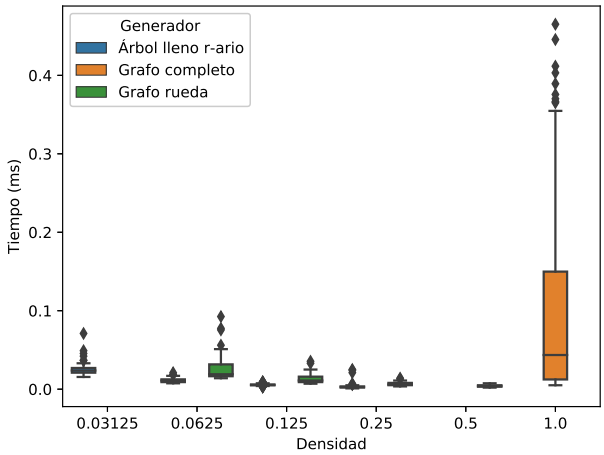


Figura 4: Efecto de la densidad del grafo sobre el tiempo de ejecución

Como se puede apreciar a partir de la figura anterior, la variación de los tiempos de ejecución es muy pequeña, sin embargo cuando la densidad llega a uno, el valor de este se incrementa de forma considerable, lo que permite afirmar que la densidad del grafo si va a repercutir en el tiempo de ejecución.

Seguidamente se muestra un cuadro en el que se exponen los resultados del análisis de varianza (ANOVA):

Cuadro 1: Resultados obtenidos ANOVA

	sum sq	**df**	**F**	**PR(>F)**
Generador	$1.394\,225 \times 10^{-10}$	2	$3.931\,140 \times 10^{-7}$	1.00
Algoritmo	$4.165\,406 \times 10^{-3}$	2	$1.174\,473 \times 10^{1}$	0.09
Orden	$-1.289\,626 \times 10^{-11}$	1	$-7.272\,428 \times 10^{-8}$	1.00
Densidad	$1.511\,910 \times 10^{-10}$	1	$8.525\,926 \times 10^{-7}$	0.99
Generador:Algoritmo	$2.385\,267 \times 10^{-4}$	4	$3.362\,736 \times 10^{-1}$	0.85
Generador:Orden	$9.396\,538 \times 10^{-7}$	2	$2.649\,437 \times 10^{-3}$	0.99
Generador:Densidad	$1.879\,308 \times 10^{-6}$	2	$5.298\,874 \times 10^{-3}$	0.94
Algoritmo:Densidad	$6.478\,017 \times 10^{-4}$	2	1.826 534	0.16
Orden:Densidad	$9.396\,538 \times 10^{-7}$	1	$5.298\,874 \times 10^{-3}$	0.94
Residual	$3.160\,036 \times 10^{-1}$	1782		

A partir de los resultados expuestos anteriormente se puede concluir que para el caso de el método generador, el orden del grafo y la densidad, como su p valor > 0.05, el tiempo de ejecución va a sufrir variaciones en dependencia del método elegido, el orden asignado y la densidad que posea el grafo en cuestión, lo cual no sucede para el caso del algoritmo seleccionado, el cual no va a tener ninguna injerencia o efecto sobre el tiempo de ejecución.

Referencias

[1] Agustín J. González. Estructura de datos y algoritmos. Last access in 04-01-2019 to http://profesores.elo.utfsm.cl/ agv/elo320/01and02/redesDeFlujo/maximumFlow.pdf, 2002.

[2] Networkx. Networkx generators classic full_rary_tree. Last access in 04-01-2019 to http://networkx.github.io/documentation/stable/reference /generated/networkx.generators.classic.full_rary_tree.html#networkx.generators.classic.full_rary_tree, 2018.

[3] Networkx. Networkx generators classic complete_graph. Last access in 04-01-2019 to http://networkx.github.io/documentation/stable/reference /generated/networkx.generators.classic.complete_graph.html#networkx.generators.classic.complete_graph, 2018.

[4] Networkx. Networkx generators classic wheel_graph. Last access in 04-01-2019 to http://networkx.github.io/documentation/stable/reference /generated/networkx.generators.classic.wheel_graph.html#networkx.generators.classic.wheel_graph, 2018.

[5] Networkx. Networkx algorithms flow minimum_cut. Last access in 04-01-2019 to http://networkx.github.io/documentation/stable/reference /algorithms/generated/networkx.algorithms.flow.minimum_cut.html#networkx.algorithms.flow.minimum_cut, 2018.

[6] Networkx. Networkx algorithms flow minimum_cut_value. Last access in 04-01-2019 to http://networkx.github.io/documentation/stable/reference /algorithms/generated/networkx.algorithms.flow.minimum_cut_value.html#networkx.algorithms.flow.minimum_cut_value, 2018.

[7] Tim Shearouse. Class completegraphgenerator. Last access in 04-01-2019 to http://jgrapht.org/javadoc/org/jgrapht/generate/CompleteGraphGenerator.html, 2018.

[8] MathWorld Team. Wheel graph. Last access in 04-01-2019 to http://mathworld.wolfram.com/WheelGraph.html, 2019.

5. Conclusión Tarea 4

Con la realización de este ejercicio y haciendo uso de métodos estadísticos como el ANOVA, se tuvo la oportunidad de observar el efecto que producían variables como: el generador, el orden y la densidad del grafo, así como el algoritmo sobre el tiempo de ejecución.

6. Tarea 5

Profe en el caso de la tarea número cinco no se la pude entregar en tiempo y forma, debido a que tuve para esa fecha problemas familiares, y por ende falta de tiempo, pero a continuación aparece el reporte de la misma.

Tarea 5 Optimización de Flujo en Redes

Orlando Lázaro Ruiloba Torres 5270

4 de junio de 2019

1. Introducción

El objetivo de esta tarea es el análisis de 5 instancias (grafos) en las cuales se desea maximizar el flujo entre sus nodos, para proceder a esto se selecciona un método generador, un algoritmo de acomodo y posteriormente se analiza la influencia del tiempo de ejecución sobre el algoritmo de flujo máximo basado en ciertas características como: la distribución de grado, la centralidad de cercanía, el coeficiente de agrupamiento, la excentricidad, entre otras.

El algoritmo de flujo máximo lo que hace básicamente es, dado que se distinguen dos nodos especiales en G, el nodo fuente s y el nodo sumidero t, determinar la mayor cantidad de flujo que se puede enviar entre ambos a través de la red [1].

2. Características estructurales

- **Distribución de grado:** El grado de un nodo en una red, es el número de conexiones que tiene con otros nodos y la distribución de grados se refiere a la distribución de probabilidad de estos grados en toda la red.
- **Coeficiente de agrupamiento:** El coeficiente de agrupamiento de un vértice en un grafo cuantifica qué tanto está de agrupado (o interconectado) con sus vecinos. Se suele representar formalmente con C_i.
- **Centralidad de cercanía:** La centralidad en un grafo se refiere a una medida posible de un vértice en dicho grafo, que determina su importancia relativa dentro de éste. La centralidad mide según cierto criterio la contribución de un nodo según su ubicación en la red, independientemente de si se está evaluando su importancia, influencia, relevancia o prominencia.
- **Centralidad de carga:** La centralidad de carga de un nodo es la fracción de todas las rutas más cortas que pasan a través de dicho nodo.
- **Excentricidad:** La excentricidad de un nodo v es la distancia máxima de v a todos los demás nodos en G.
- **PageRank:** Este algoritmo calcula una clasificación de los nodos en el grafo G en función de la estructura de los enlaces entrantes. Originalmente fue diseñado como un algoritmo para clasificar páginas web.

3. Generación de instancias

Se generaron 5 grafos, para la generación de los mismos se utilizó el método generador *watts_strogatz_graph*. Las instancias generadas constan de diferente tamaño (20, 22, 24, 26 y 28 nodos respectivamente) en las mismas se denotan con color verde los nodos que son fuente y con color rojo los nodos que son sumideros, en cuanto a las aristas su grosor es proporcional al flujo que pasa a través de ellas.

El código en Python para generar los grafos se muestra a continuación:

```
import networkx as nx
import matplotlib.pyplot as plt
from numpy import array
import random

#Generar grafos

random.seed(a=1)
nodos=array([20,22,24,26,28])
grafos={}
aux=0
k=.3
p=0.4

for i in nodos:
    grafos[aux] = nx.watts_strogatz_graph(i, int (k * i.item()), p, seed=aux)
    aux=aux+1

#Dibujar grafos

arreglo=[0,1,2,3,4]
Pesos={}

for i in arreglo:
    plt.clf()
    pos = nx.spring_layout(grafos[i])

    no_edges=grafos[i].number_of_edges()
    no_nodes=grafos[i].number_of_nodes()

    Pesos=[round((max(0.1,random.gauss(10,4)))/3,2) for _ in range(no_edges)]
    arr_edges=[_ for _ in grafos[i].edges()]


    e=[]
    for j in range(no_edges):
        a=(arr_edges[j][0],arr_edges[j][1],Pesos[j])
        e.append(a)

    grafos[i].add_weighted_edges_from(e)
```

A continuación se muestran los grafos generados (**ver Figuras 1, 2, 3, 4 y 5**).

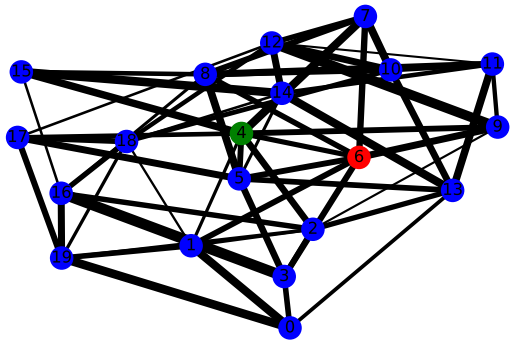


Figura 1: Instancia de 20 nodos

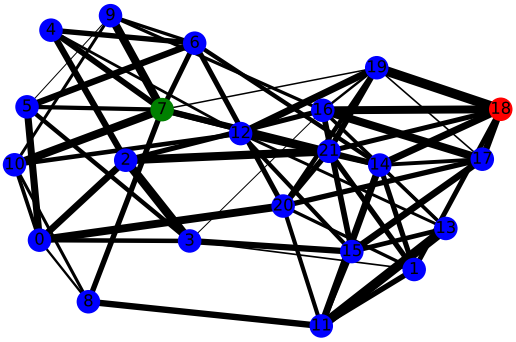


Figura 2: Instancia de 22 nodos

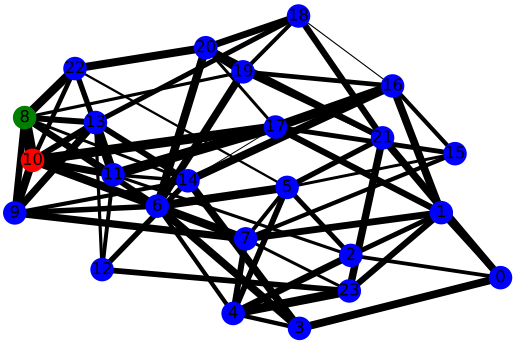


Figura 3: Instancia de 24 nodos

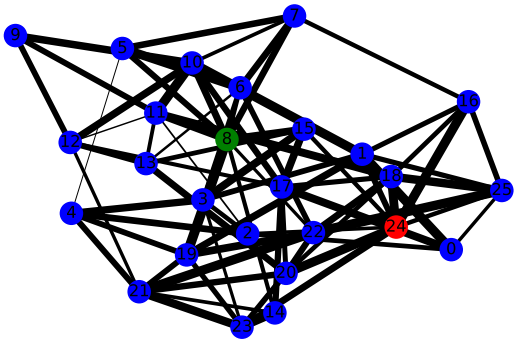


Figura 4: Instancia de 26 nodos

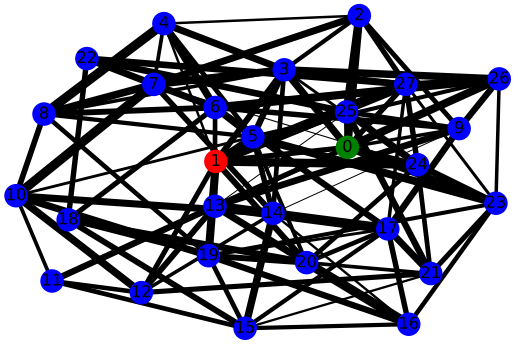
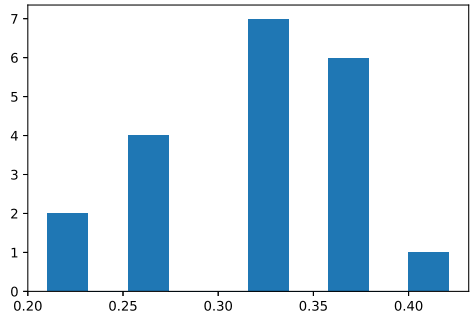
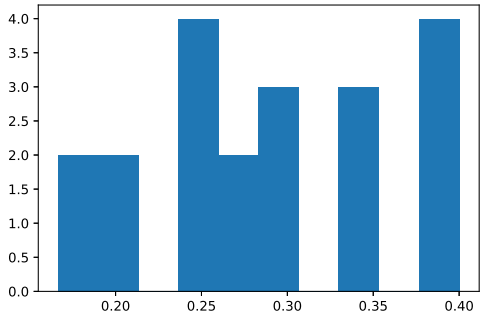


Figura 5: Instancia de 28 nodos

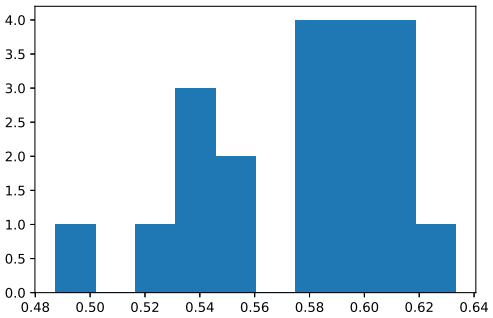
A continuación como parte del análisis realizado se muestran los histogramas a través de los cuales se puede observar la distribución que poseen los datos de cada instancia con referencia a las características estructurales:



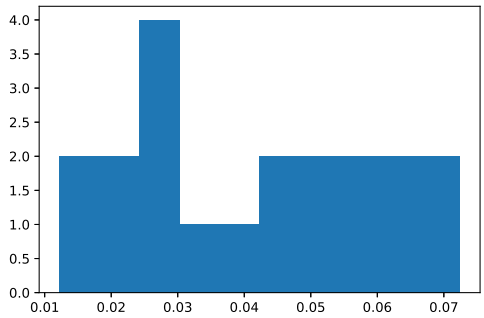
(a) Distribución de grado



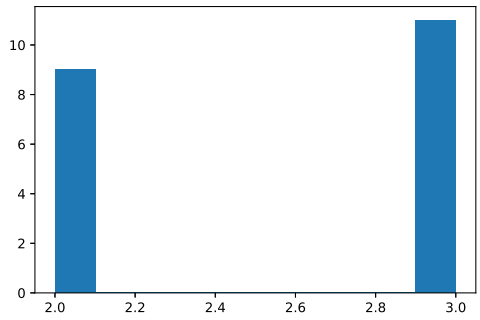
(b) Coeficiente de agrupamiento



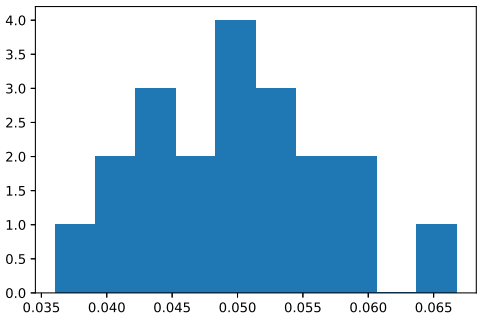
(c) Centralidad de cercanía



(d) Centralidad de carga

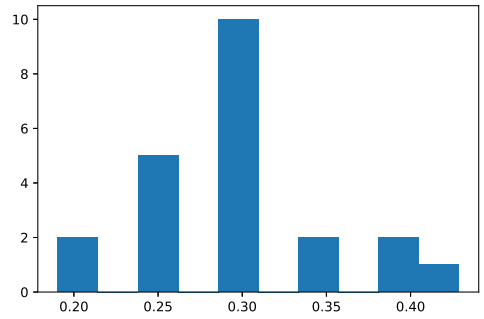


(e) Excentricidad

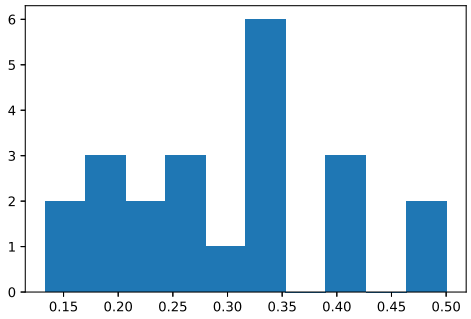


(f) PageRank

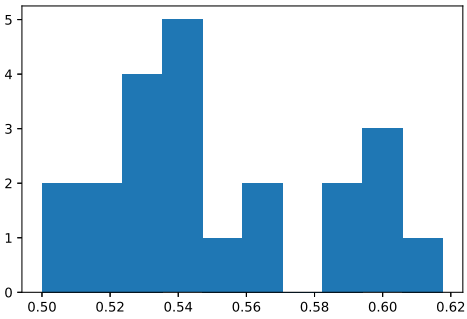
Figura 6: Visualización de los histogramas de la instancia de 20 nodos para las diferentes características estructurales.



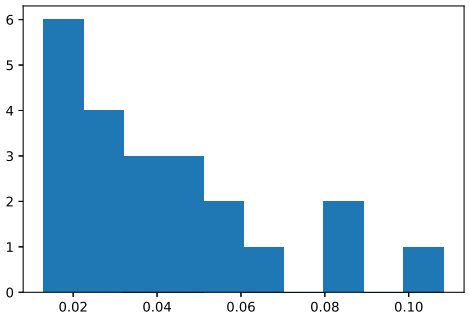
(a) Distribución de grado



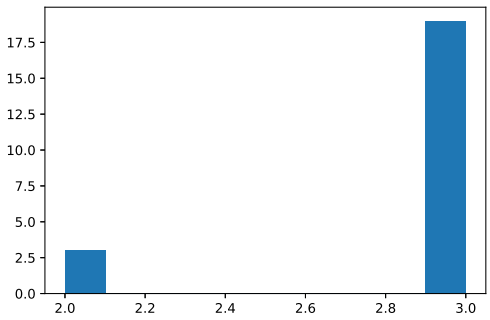
(b) Coeficiente de agrupamiento



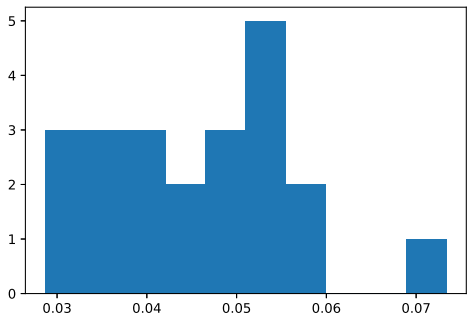
(c) Centralidad de cercanía



(d) Centralidad de carga

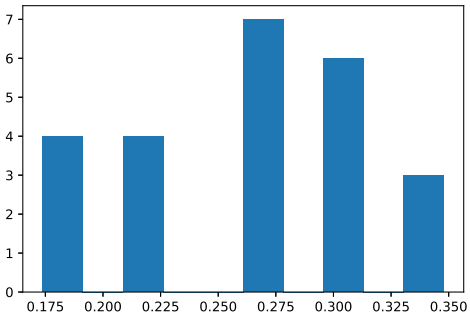


(e) Excentricidad

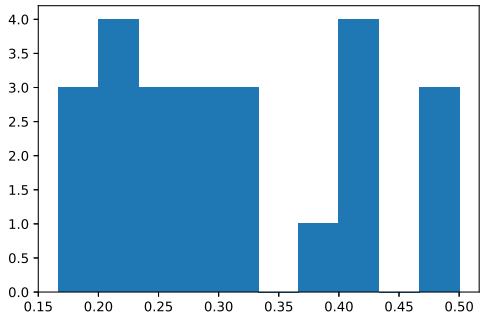


(f) PageRank

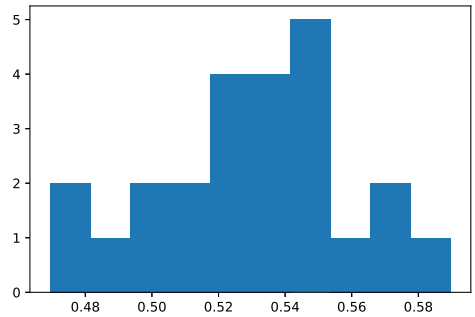
Figura 7: Visualización de los histogramas de la instancia de 22 nodos para las diferentes características estructurales.



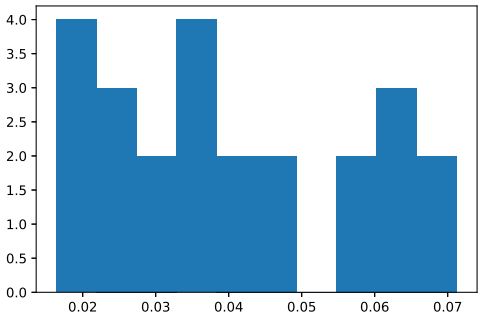
(a) Distribución de grado



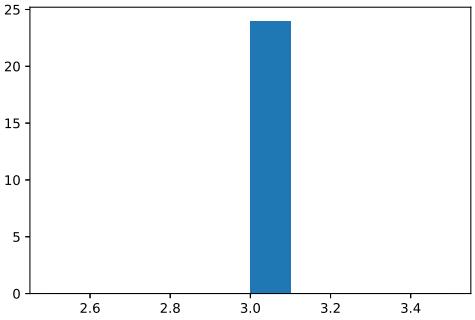
(b) Coeficiente de agrupamiento



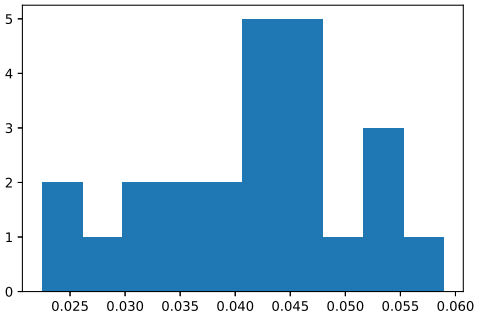
(c) Centralidad de cercanía



(d) Centralidad de carga

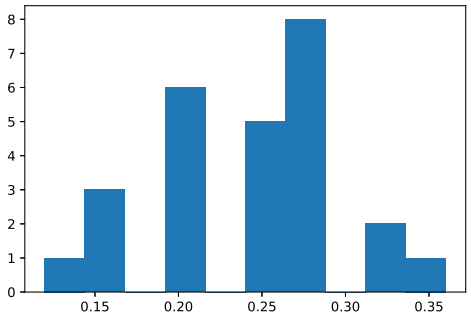


(e) Excentricidad

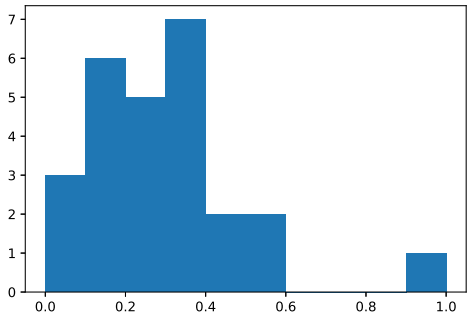


(f) PageRank

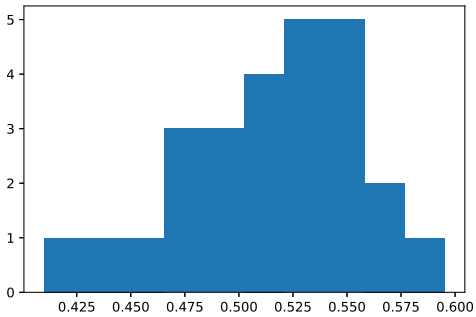
Figura 8: Visualización de los histogramas de la instancia de 24 nodos para las diferentes características estructurales.



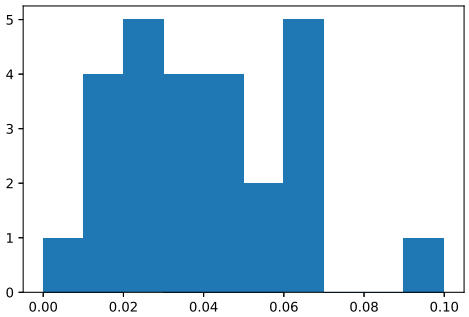
(a) Distribución de grado



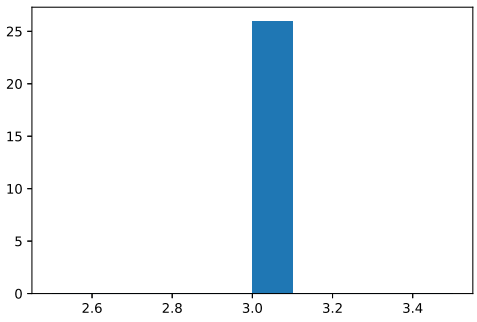
(b) Coeficiente de agrupamiento



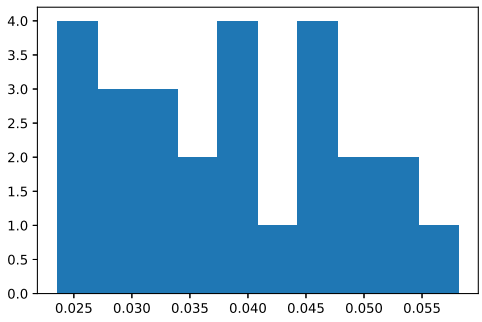
(c) Centralidad de cercanía



(d) Centralidad de carga

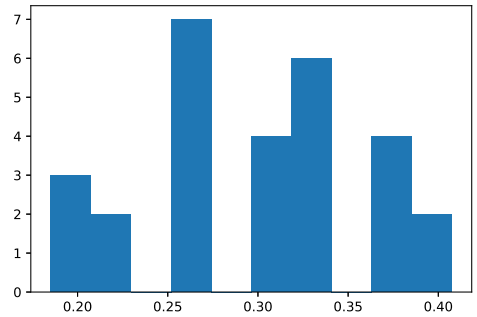


(e) Excentricidad

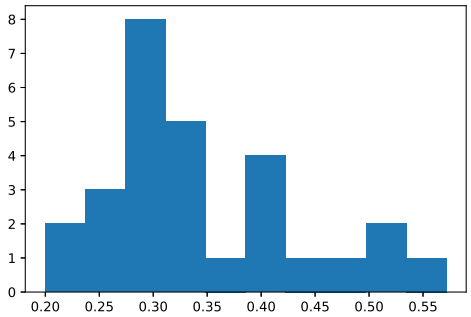


(f) PageRank

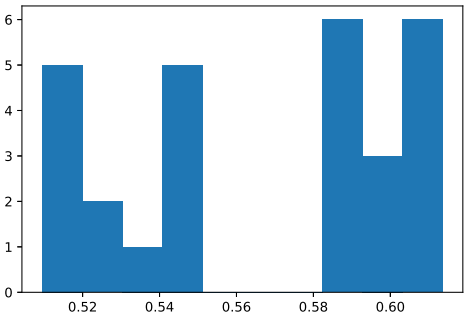
Figura 9: Visualización de los histogramas de la instancia de 26 nodos para las diferentes características estructurales.



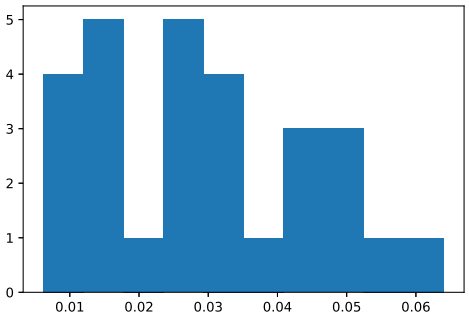
(a) Distribución de grado



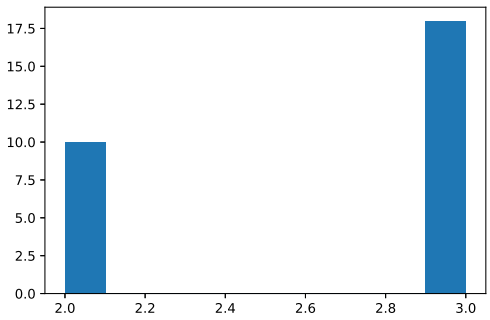
(b) Coeficiente de agrupamiento



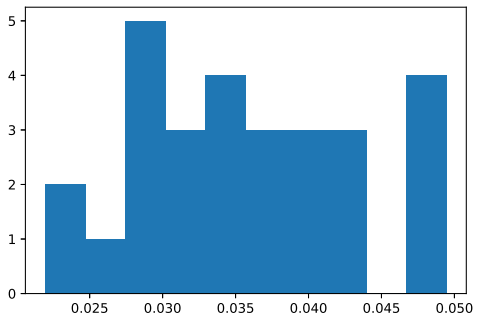
(c) Centralidad de cercanía



(d) Centralidad de carga



(e) Excentricidad



(f) PageRank

Figura 10: Visualización de los histogramas de la instancia de 28 nodos para las diferentes características estructurales.

A partir del análisis de los histogramas anteriores se puede concluir que los datos no siguen una distribución normal, por lo que se puede afirmar que las características estructurales analizadas no van a tener ninguna injerencia sobre el tiempo promedio de ejecución del algoritmo.

A continuación se muestran los diagramas de caja y bigote para cada una de las instancias sujetas a análisis, de igual manera teniendo en cuenta las características estructurales:

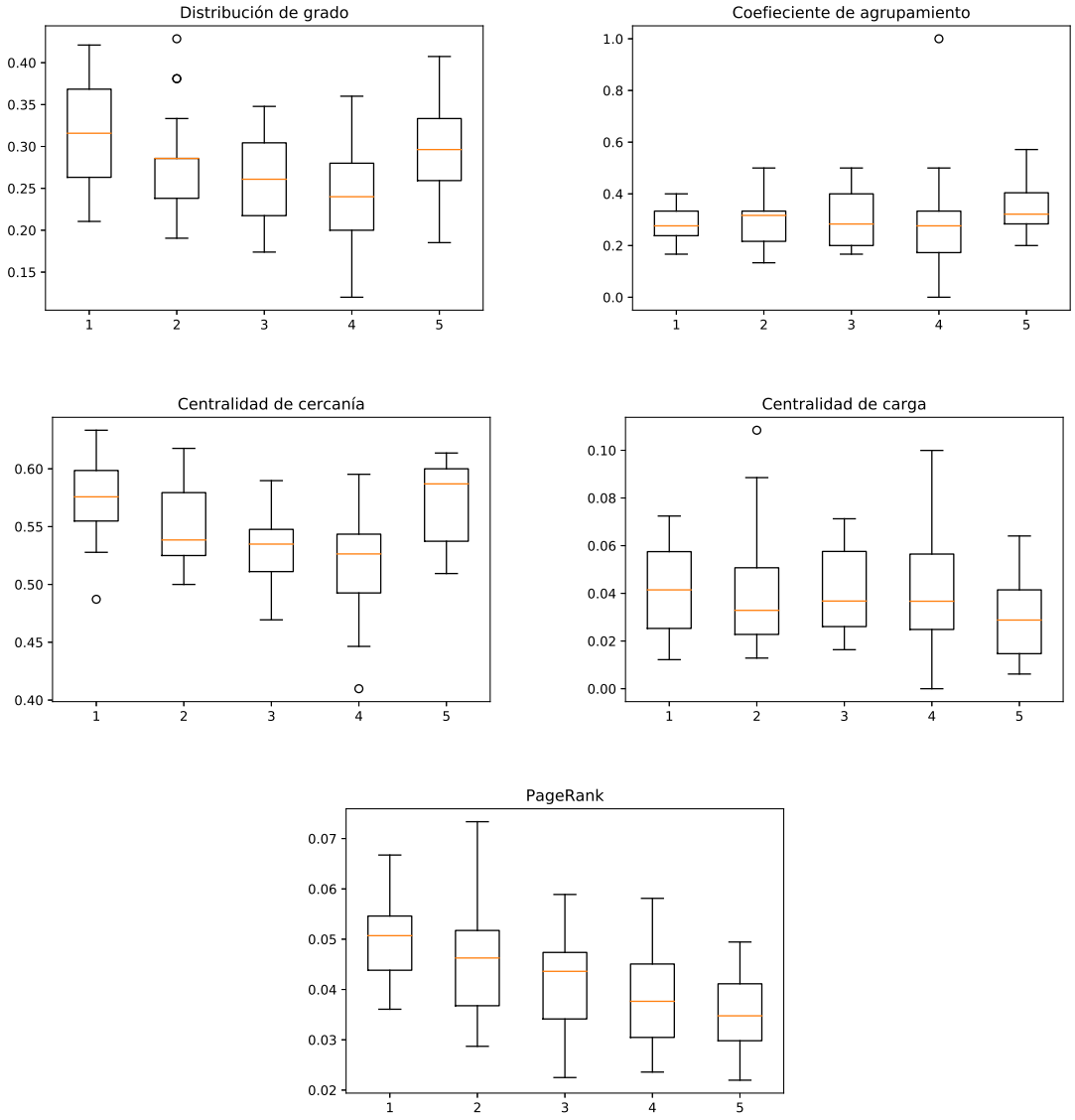


Figura 11: Visualización de los diagramas de caja y bigote para las instancias.

A partir de la visualización de los diagramas anteriores se puede apreciar que los datos no siguen una distribución simétrica en la mayoría de los casos, pudiendo apreciarse valores atípicos para la instancia de 22 nodos referidos a la característica de distribución de grado, para la instancia de 26 nodos referido al coeficiente de agrupamiento y para las instancias de 20 y 22 nodos referidos a la centralidad de cercanía y centralidad de carga respectivamente.

Como parte también de la experimentación se determinaron los nodos que de acuerdo a cada instancia, constituyen mejor fuente y mejor sumidero, así como el valor del flujo máximo, los valores obtenidos se muestran a continuación:

Para la instancia de 20 nodos:

- Flujo máximo = 25.78
- nodo inicio (4)
- nodo fin (6)
- distribución de grado = 0.42
- coeficiente de agrupamiento = 0.25
- centralidad de cercanía = 0.63
- centralidad de carga = 0.07
- excentricidad = 2
- PageRank = 0.06

Para la instancia de 22 nodos:

- Flujo máximo = 25.18
- nodo inicio (7)
- nodo fin (18)
- distribución de grado = 0.42
- coeficiente de agrupamiento = 0.22
- centralidad de cercanía = 0.61
- centralidad de carga = 0.10
- excentricidad = 3
- PageRank = 0.07

Para la instancia de 24 nodos:

- Flujo máximo = 30.93
- nodo inicio (8)
- nodo fin (10)
- distribución de grado = 0.34
- coeficiente de agrupamiento = 0.42
- centralidad de cercanía = 0.54
- centralidad de carga = 0.04
- excentricidad = 3
- PageRank = 0.05

Para la instancia de 26 nodos:

- Flujo máximo = 32.1
- nodo inicio (8)
- nodo fin (24)
- distribución de grado = 0.36
- coeficiente de agrupamiento = 0.19
- centralidad de cercanía = 0.59
- centralidad de carga = 0.09
- excentricidad = 3
- PageRank = 0.05

Para la instancia de 28 nodos:

- Flujo máximo = 39.01
- nodo inicio (0)
- nodo fin (1)
- distribución de grado = 0.40
- coeficiente de agrupamiento = 0.40
- centralidad de cercanía = 0.61
- centralidad de carga = 0.05
- excentricidad = 3
- PageRank = 0.04

Referencias

[1] Antonio Alberto Sedeño Noda. Nuevos algoritmos y mejoras computacionales para problemas de flujos en redes. 2000.

7. Conclusión Tarea 5

Con la realización de este ejercicio se tuvo la oportunidad de aprender acerca de cómo pueden influir ciertas características estructurales de las instancias sobre el tiempo promedio de ejecución de un algoritmo, y la importancia de elegir una buena combinación fuente - sumidero.