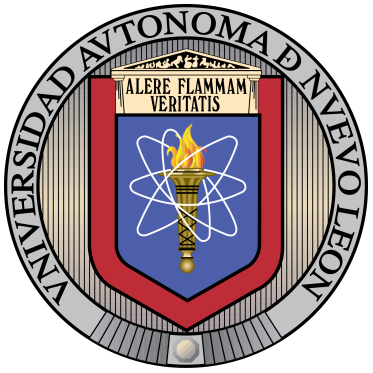


Universidad Autonoma de Nuevo León

Facultad de Ingenieria Mecánica y Eléctrica

Optimización de flujo en redes

Portafolio de Tareas

Alumno:
Jesús Angel Patlán Castillo

Matricula:
1595261

Enero - Junio 2019

Fecha - 04/06/19

1. Introducción

Este documento recopila las tareas de la materia Optimización de flujo en redes realizadas durante el periodo Enero-Junio 2019. Cada una de las tareas esta calificadas por la Dra. Satu Elisa Schaeffer, con anotaciones de aspectos a mejorar; los cuales fueron considerados para ser corregidos en esta recopilación. Se muestra cada tarea con una lista con los puntos corregidos, posteriormente se muestra la tarea con las anotaciones y finalmente, si se considera necesario, la tarea con las nuevas correcciones. Al final del documento, se incluye la conclusión de aprendizaje del curso. Estas tareas se encuentran dentro de mi repositorio personal [1].

2. Tarea 1

Para esta tarea:

- Se realizaron pequeñas correcciones al documento de diseño y errores ortográficos.

Dado que las correcciones no muestran un cambio significativo al documento, se decide no mostrar la nueva versión.

Tarea 1

Jesus Angel Patlán Castillo (5621)

12 de febrero de 2019

En esta tarea se recopila información de grafos con distintas propiedades. Se utiliza código de Python [1], con la librería NetworkX [3] para la generación de grafos y la librería Matplotlib [4] para guardar el grafo en el formato ".eps". El código empleado se obtuvo consultando la documentación oficial de la librería NetworkX [2] y guías suplementarias [6] [10]. Las imágenes y el código se encuentran disponibles directamente en mi repositorio [5].

1. Grafo simple no dirigido acíclico

Una red de ciudades en un estado puede ser un ejemplo de un grafo simple no dirigido acíclico, ya que podemos representar cada ciudad como un nodo del grafo, y tomar en cuenta que cada arista representan las carreteras que conectan cada ciudad, considerando que sería un grafo no dirigido dado que una carretera puede ir de ida y vuelta entre ciudades, y es acíclico puesto que no se consideran carreteras que van de una ciudad a sí misma [8]. La figura 1 representa un ejemplo de este tipo de grafo.

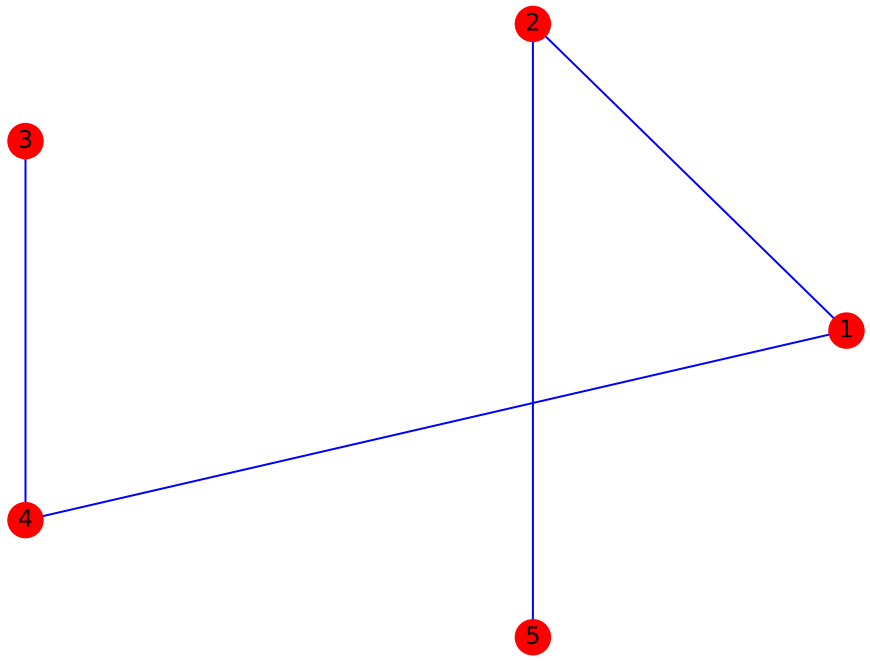


Figura 1: Grafo simple no dirigido acíclico

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


#Creamos una instancia de un grafo
G = nx.Graph()

#Agregamos las aristas en el nodo (junto con los nodos)
G.add_edge(1,2)
G.add_edge(1,4)
G.add_edge(3,4)
G.add_edge(2,5)

#Dibujamos el grafo
nx.draw(G, pos=nx.circular_layout(G), node_color='r', edge_color='b
    ', with_labels=True)
#Mostramos el grafo en pantalla
plt.show()
```

2. Grafo simple no dirigido cíclico

Una ruta de autobús puede ser representada por este tipo de grafos, en donde los autobuses pasan a través de las calles y avenidas (representadas con

las aristas), y cada nodo del grafo se puede representar una parada donde se recogen personas. La figura 2 representa un ejemplo de este tipo de grafo.

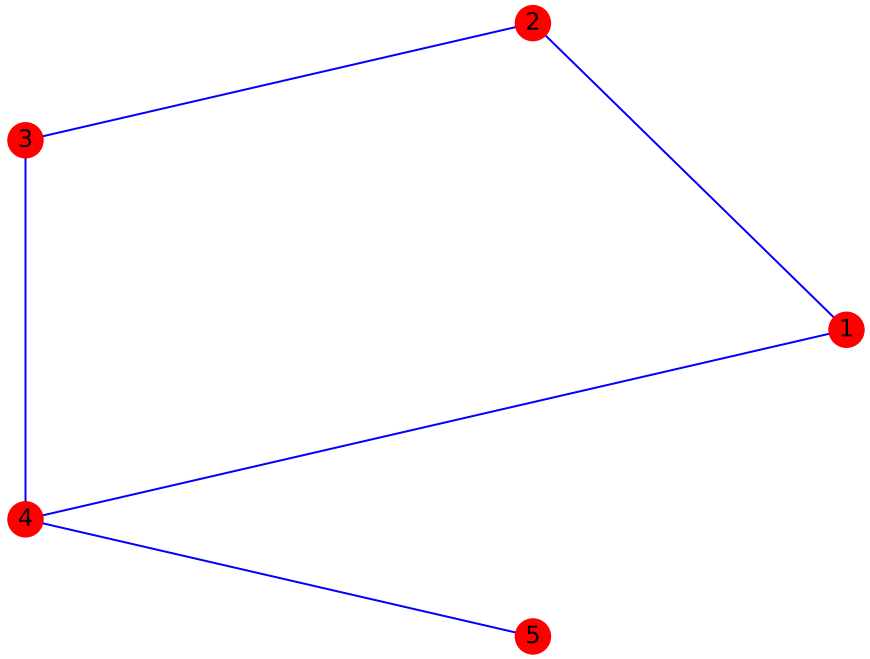


Figura 2: Grafo simple no dirigido cíclico

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


#Creamos una instancia de un grafo
G = nx.Graph()

#Agregamos las aristas en el nodo (junto con los nodos)
G.add_edge(1,2)
G.add_edge(2,3)
G.add_edge(3,4)
G.add_edge(4,1)
G.add_edge(4,5)

#Dibujamos el grafo
nx.draw(G, pos=nx.circular_layout(G), node_color='r', edge_color='b
    ', with_labels=True)
#Mostramos el grafo en pantalla
plt.show()
```

3. Grafo simple no dirigido reflexivo

Podemos representar una red de sistemas informáticos por medio de un grafo reflexivo, donde cada computadora es representada por medio de un nodo, y la conexión hacia las demás computadoras son representadas por las aristas. La arista reflexiva representaría la conexión de una computadora consigo misma [7]. La figura 3 representa un ejemplo de este tipo de grafo.

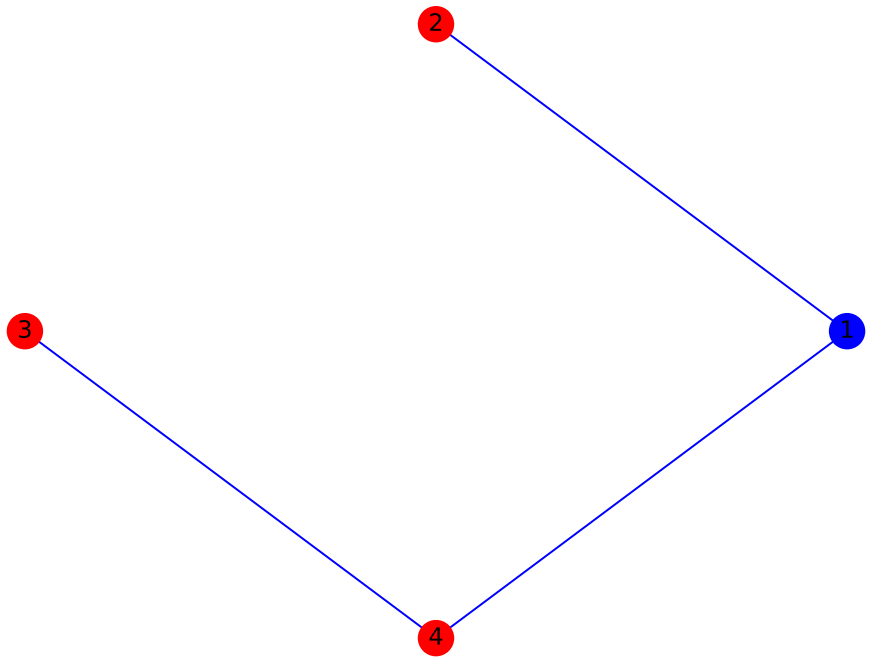


Figura 3: Grafo simple no dirigido reflexivo (El nodo 1 tiene una arista reflexiva)

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


#Creamos una instancia de un grafo
G = nx.Graph()

#Agregamos las aristas en el nodo (junto con los nodos)
G.add_edge(1,2)
G.add_edge(1,1)
G.add_edge(1,4)
G.add_edge(1,5)
G.add_edge(3,4)

#Dibujamos el grafo
```

```
16 nx.draw(G, pos=nx.circular_layout(G), node_color=['b','r','r','r','
       r'], edge_color='b', with_labels=True)
17 #Mostramos el grafo en pantalla
18 plt.show()
```

4. Grafo simple dirigido acíclico

En la Programación Orientada a Objetos es común realizar herencia entre clases, y esta puede ser representada por medio de un grafo dirigido acíclico, teniendo a cada nodo como una clase distinta, y señalando la herencia por medio de una arista dirigida a las clases hijas [9]. La figura 4 representa un ejemplo de este tipo de grafo.

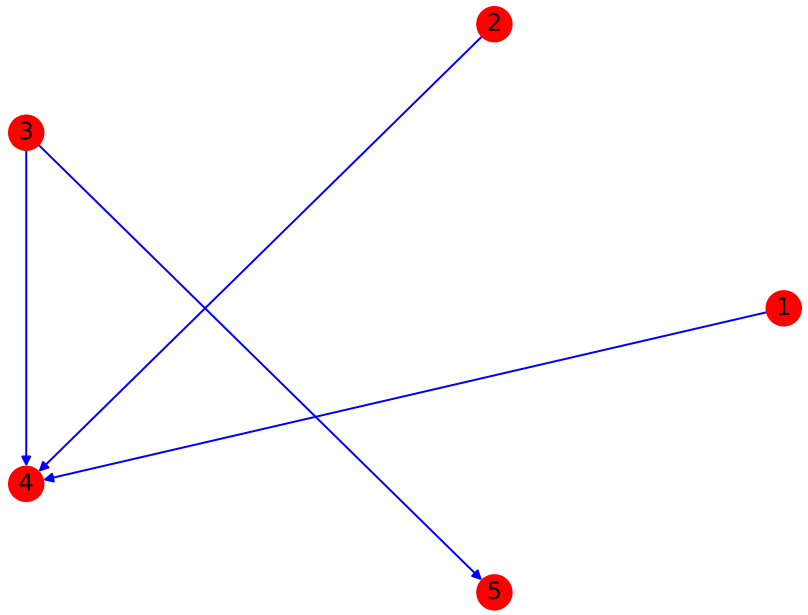


Figura 4: Grafo simple dirigido acíclico

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4
5 #Creamos una instancia de un grafo
6 G = nx.DiGraph()
7
8 #Agregamos las aristas en el nodo (junto con los nodos)
9 G.add_edge(1,4)
```

```
G.add_edge(2,4)
G.add_edge(3,4)
G.add_edge(3,5)

#Dibujamos el grafo
nx.draw(G, pos=nx.circular_layout(G), node_color='r', edge_color='b
    ', with_labels=True)
#Mostramos el grafo en pantalla
plt.show()
```

5. Grafo simple dirigido cíclico

En algunos juegos se tienen distintos estados en los que el jugador se puede encontrar, y esto puede ser representado en un grafo dirigido cíclico, dado que cada estado se puede visualizar por medio de un nodo, y las aristas representarían las maneras en las que un estado puede cambiar a otro [9]. La figura 5 representa un ejemplo de este tipo de grafo.

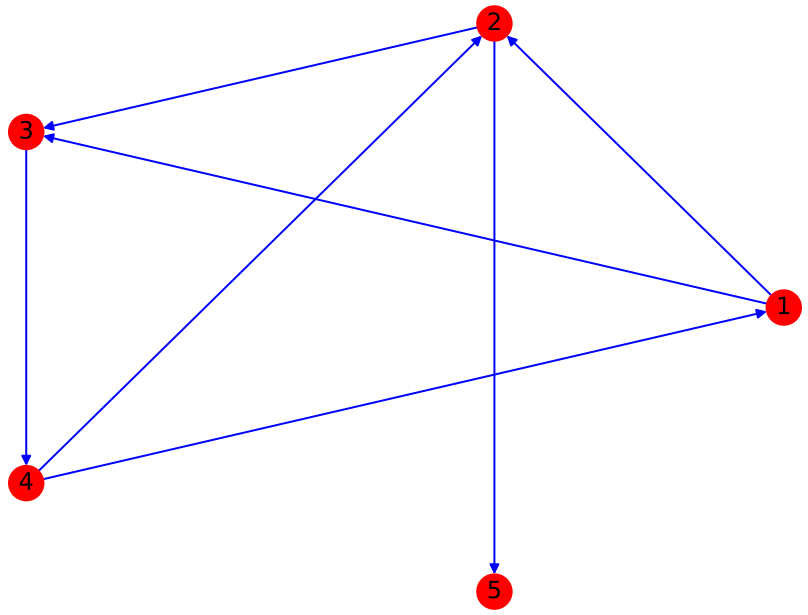


Figura 5: Grafo simple dirigido cíclico

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


```

```
#Creamos una instancia de un grafo
G = nx.DiGraph()

#Agregamos las aristas en el nodo (junto con los nodos)
G.add_edge(1,2)
G.add_edge(2,3)
G.add_edge(2,5)
G.add_edge(1,3)
G.add_edge(3,4)
G.add_edge(4,2)
G.add_edge(4,1)

#Dibujamos el grafo
nx.draw(G, pos=nx.circular_layout(G), node_color='r', edge_color='b
    ', with_labels=True)
#Mostramos el grafo en pantalla
plt.show()
```

6. Grafo simple dirigido reflexivo

Se puede realizar un grafo dirigido reflexivo para representar todos los hipervínculos (aristas) que dirigen a una página web (nodos) en los que el usuario puede navegar a través de una web, cuando una página web se redirige a sí misma se utiliza una arista reflexiva [9]. La figura 6 representa un ejemplo de este tipo de grafo.

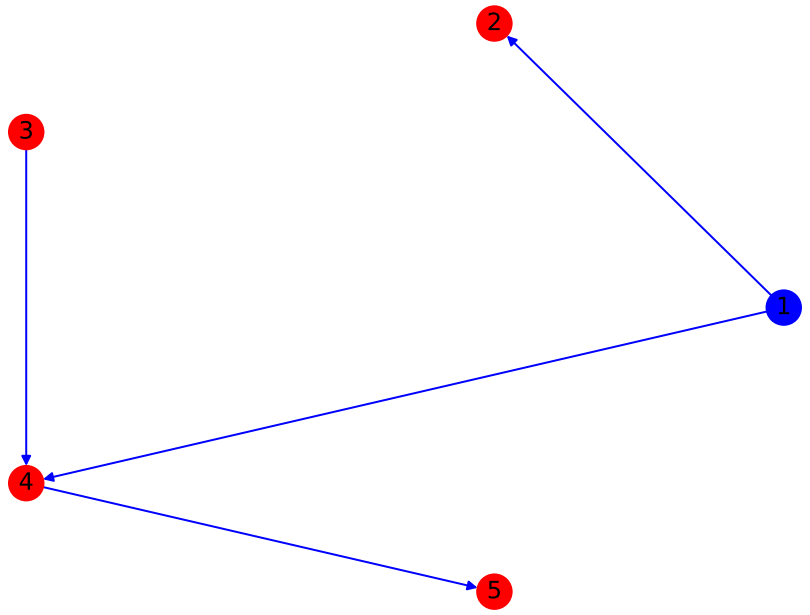


Figura 6: Grafo simple dirigido reflexivo (El nodo 1 tiene una arista reflexiva)

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


#Creamos una instancia de un grafo
G = nx.DiGraph()

#Agregamos las aristas en el nodo (junto con los nodos)
G.add_edge(1,2)
G.add_edge(1,1)
G.add_edge(1,4)
G.add_edge(3,4)
G.add_edge(4,5)

#Dibujamos el grafo
nx.draw(G, pos=nx.circular_layout(G), node_color=['b','r','r','r','r'], edge_color='b', with_labels=True)
#Mostramos el grafo en pantalla
plt.show()
```

7. Multigrafo no dirigido acíclico

De manera similar al ejemplo de un grafo simple no dirigido acíclico, una red de ciudades puede ser representado por un multigrafo, el cual puede pro-

porcionar más información que un grafo simple al añadir diversos caminos por el que se puede trasladar de un nodo a otro. La figura 7 representa un ejemplo de este tipo de grafo.

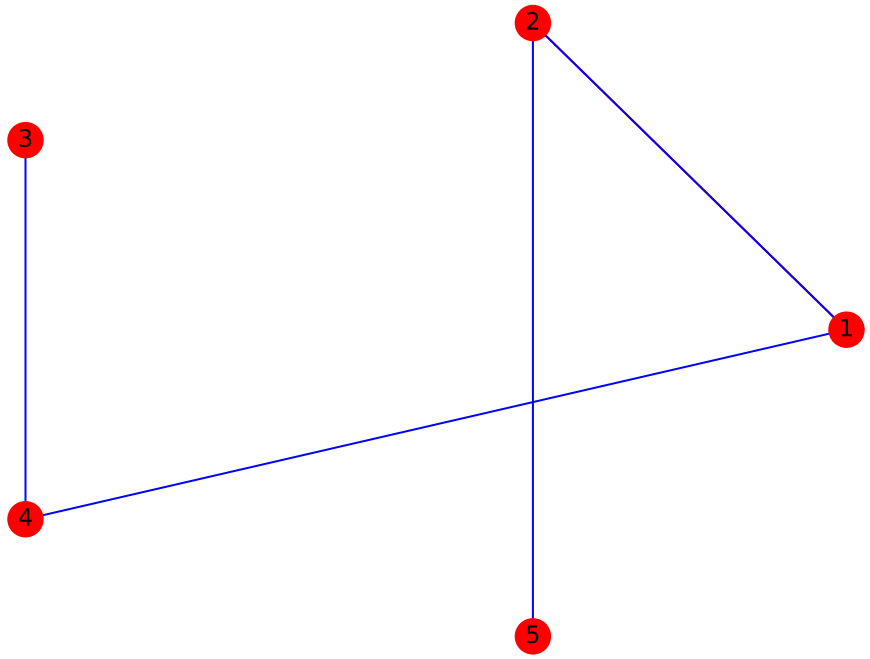


Figura 7: Multigrafo no dirigido acíclico (Los nodos 1 y 2 tienen múltiples aristas)

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


#Creamos una instancia de un multigrafo
G = nx.MultiGraph()

#Agregamos las aristas en el nodo (junto con los nodos)
G.add_edge(1,2)
G.add_edge(1,2)
G.add_edge(1,4)
G.add_edge(3,4)
G.add_edge(2,5)
#Dibujamos el grafo
nx.draw(G, pos=nx.circular_layout(G), node_color=['r','r','r','r','r'], edge_color=['r','b','b','b','b'], with_labels=True)
#Mostramos el grafo en pantalla
plt.show()
```

8. Multigrafo no dirigido cíclico

Considerando el ejemplo de la ruta de autobuses, podemos tener entre dos paradas (nodos) múltiples rutas para llegar a la parada siguiente. La figura 8 representa un ejemplo de este tipo de grafo.

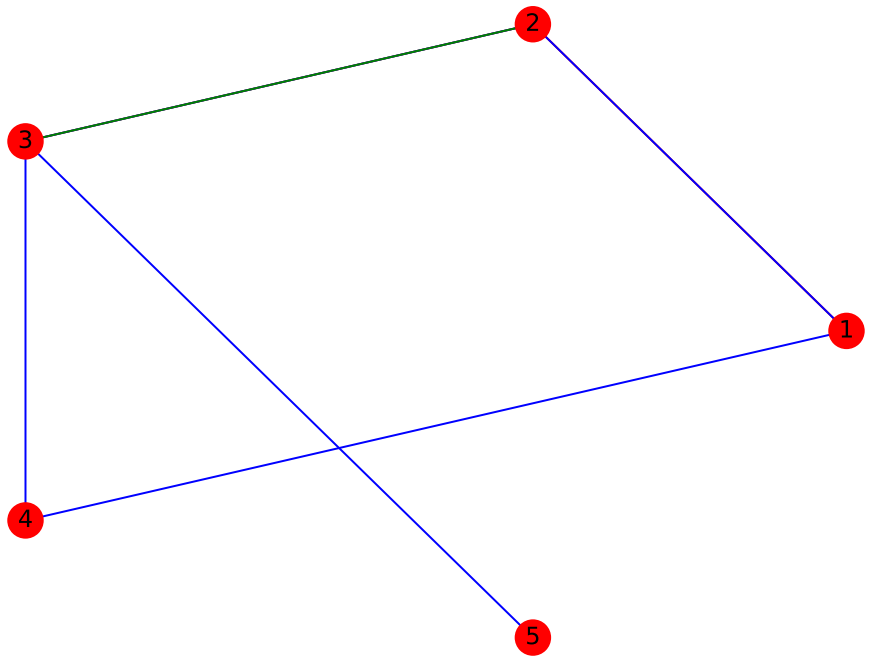


Figura 8: Multigrafo no dirigido cíclico (Los nodos 1, 2 y 3 tienen múltiples aristas)

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


#Creamos una instancia de un multigrafo
G = nx.MultiGraph()

#Agregamos las aristas en el nodo (junto con los nodos)
G.add_edge(1,2)
G.add_edge(1,2)
G.add_edge(2,3)
G.add_edge(2,3)
G.add_edge(2,3)
G.add_edge(3,4)
G.add_edge(3,5)
G.add_edge(4,1)

```

```
#Dibujamos el grafo
nx.draw(G, pos=nx.circular_layout(G), nodecolor='r', edge_color=['r
    ','b','b','r','b','g','b','b'], with_labels=True)
#Mostramos el grafo en pantalla
plt.show()
```

9. Multigrafo no dirigido reflexivo

En las redes sociales, se puede utilizar un multigrafo reflexivo para representar las menciones que se hacen entre usuarios por medio de publicaciones, donde cada nodo representa un usuario y cada publicación representa la arista con la que conecta el usuario que realizo la publicación con el que es mencionado. La figura 9 representa un ejemplo de este tipo de grafo.

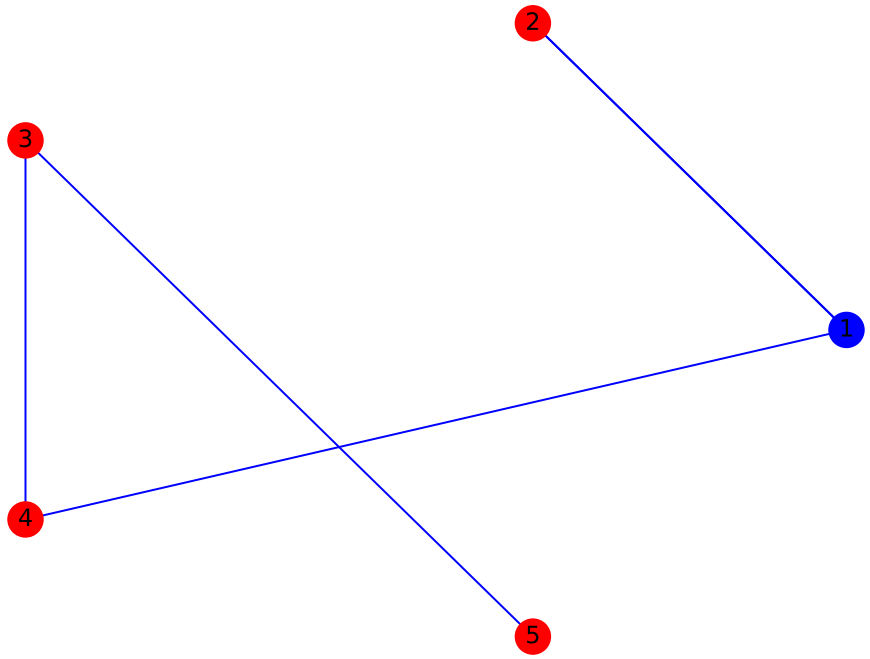


Figura 9: Multigrafo no dirigido reflexivo (Los nodos 1 y 2 tienen múltiples aristas, el nodo 1 tiene una arista reflexiva)

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


#Creamos una instancia de un grafo
G = nx.MultiGraph()

```

```
#Agregamos las aristas en el nodo (junto con los nodos)
G.add_edge(1,2)
G.add_edge(1,2)
G.add_edge(1,1)
G.add_edge(1,1)
G.add_edge(1,4)
G.add_edge(3,4)
G.add_edge(3,5)

#Dibujamos el grafo
nx.draw(G, pos=nx.circular_layout(G), node_color=['b','r','r','r','r'], edge_color=['r','b','b','b','b','b','b'], with_labels=True)
#Mostramos el grafo en pantalla
plt.show()
```

10. Multigrafo dirigido acíclico

Con un multigrafo dirigido acíclico es posible representar el flujo de dinero entre cuentas bancarias, tomando las cuentas bancarias como los nodos del grafo y las aristas el método de traspaso de una cuenta a otra. La figura 10 representa un ejemplo de este tipo de grafo.

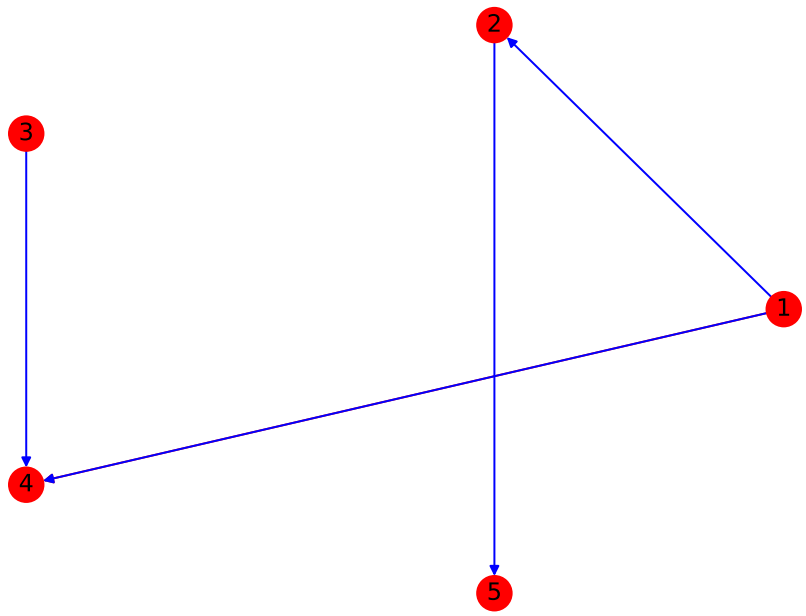


Figura 10: Multigrafo dirigido acíclico (Los nodos 1 y 4 tienen múltiples aristas)

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


#Creamos una instancia de un multigrafo dirigido
G = nx.MultiDiGraph()

#Agregamos las aristas en el nodo (junto con los nodos)
G.add_edge(1,2)
G.add_edge(1,4)
G.add_edge(1,4)
G.add_edge(3,4)
G.add_edge(2,5)

#Dibujamos el grafo
nx.draw(G, pos=nx.circular_layout(G), nodecolor='r', edge_color=['b
    ','r','b','b','b'], with_labels=True)
#Mostramos el grafo en pantalla
plt.show()
```

11. Multigrafo dirigido cíclico

Tomando el ejemplo utilizado en los grafos simples dirigidos cíclicos, en los juegos es posible que haya diferentes maneras de pasar de un estado a otro, lo cual es representado por aristas múltiples entre los nodos. La figura 11 representa un ejemplo de este tipo de grafo.

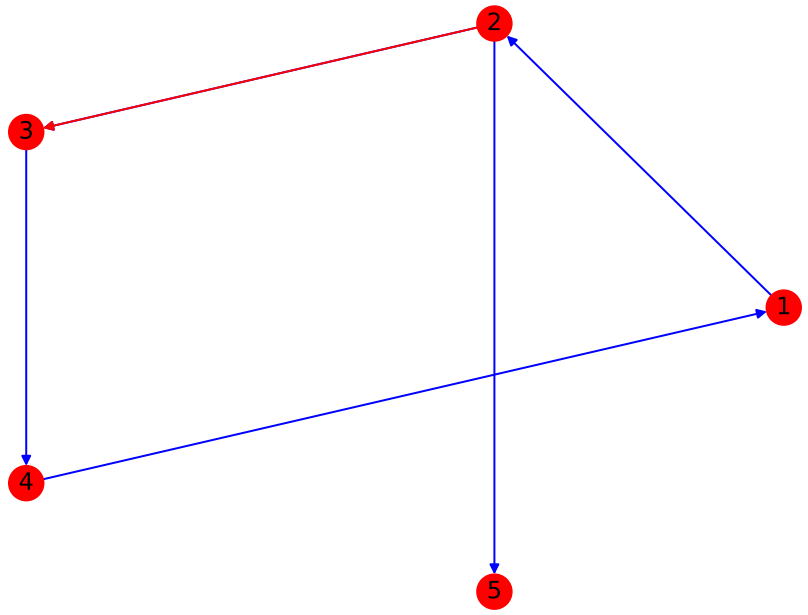


Figura 11: Multigrafo dirigido cíclico (Los nodos 2 y 3 tienen múltiples aristas)

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


#Creamos una instancia de un multigrafo
G = nx.MultiDiGraph()

#Agregamos las aristas en el nodo (junto con los nodos)
G.add_edge(1,2)
G.add_edge(2,3)
G.add_edge(2,5)
G.add_edge(2,3)
G.add_edge(3,4)
G.add_edge(4,1)

#Dibujamos el grafo
nx.draw(G, pos=nx.circular_layout(G), nodecolor='r', edge_color=['b
    ','b','r','b','b','b'], with_labels=True)
#Mostramos el grafo en pantalla
plt.show()
```

12. Multigrafo dirigido reflexivo

Como en el grafo simple dirigido reflexivo, en una página web se pueden tener múltiples hipervínculos que te redirigen a una misma página web. La figura 12 representa un ejemplo de este tipo de grafo.

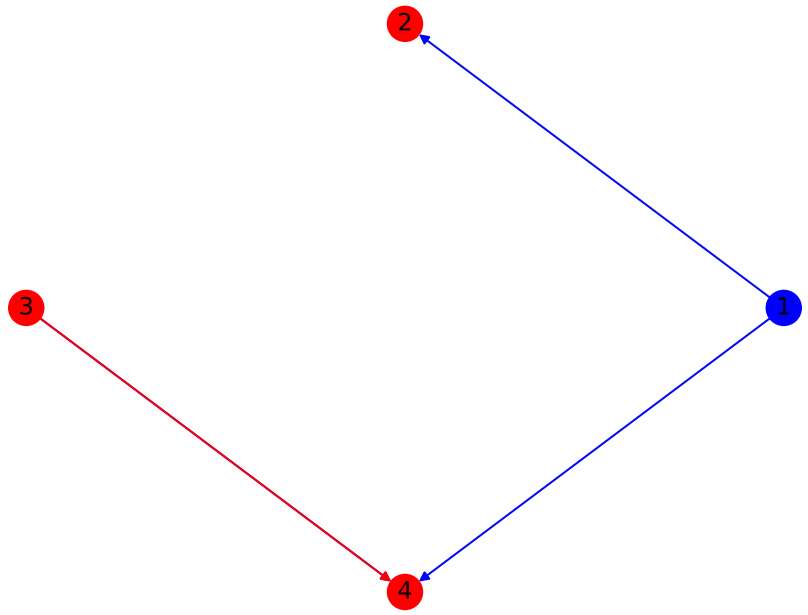


Figura 12: Multigrafo dirigido reflexivo (Los nodos 3 y 4 tienen múltiples aristas, el nodo 1 tiene una arista reflexiva)

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


#Creamos una instancia de un multigrafo
G = nx.MultiDiGraph()

#Agregamos las aristas en el nodo (junto con los nodos)
G.add_edge(1,2)
G.add_edge(1,1)
G.add_edge(1,4)
G.add_edge(3,4)
G.add_edge(3,4)

#Dibujamos el grafo
nx.draw(G, pos=nx.circular_layout(G), node_color=['b','r','r','r'],
     edge_color=['b','b','b','b','r'], with_labels=True)
```

```
#Mostramos el grafo en pantalla
plt.show()
```

Referencias

[1] Python Software Foundation Versión 3.7.2. https://www.python.org/. Copyright ©2001-2019.

[2] NetworkX developers con última actualización el 19 de Septiembre 2018. https://networkx.github.io/documentation/stable/index.html. Copyright ©2004-2018.

[3] NetworkX developers Versión 2.0. https://networkx.github.io/. Copyright ©2004-2018.

[4] The Matplotlib development team Versión 3.0.2. https://matplotlib.org/. Copyright @2002-2012.

[5] Repositorio Optimización Flujo en Redes de Jesús Angel Patlán Castillo. https://github.com/JAPatlanC/Flujo-Redes.

[6] Plotting NetworkX graph in Python Pregunta en Stackoverflow: https://stackoverflow.com/questions/44692644/plotting-networkx-graph-in-python. Copyright ©2019.

[7] What is the application of reflexive graph Pregunta en Quora. https://www.quora.com/What-is-the-application-of-reflexive-graph.

[8] E. Novo and A. Méndez Alonso. Aplicaciones de la teoría de grafos a algunos juegos de estrategia. *Suma*, 46:31–35, 2004.

[9] Example of Digraphs Applications Oxford Math Center. http://www.oxfordmathcenter.com/drupal7/node/678.

[10] How to set colors for nodes in NetworkX Pregunta en Stackoverflow. https://stackoverflow.com/questions/27030473/how-to-set-colors-for-nodes-in-networkx-python.

3. Tarea 2

Para esta tarea:

- Se realizaron pequeñas correcciones al documento de diseño y errores ortográficos.
- Se tradujeron las palabras que venían en inglés a español para mantener el mismo idioma en el documento.
- Se hizo un reacomodo en las referencias del documento.

Tarea 2

Jesus Angel Patlán Castillo (5621)

26 de febrero de 2019

En esta tarea se analizan distintos tipos de acomodo visual grafos generados a partir de la librería NetworkX [4] de Python [1]. Se estudian particularmente los grafos utilizados para la tarea 1, la librería Matplotlib [5] para generar el grafo en los distintos algoritmos estudiados y para guardar el grafo en el formato ".eps". El código empleado se obtuvo consultando la documentación oficial de la librería NetworkX [3] y guías suplementarias [7, 13]. Las imágenes y el código se encuentran disponibles directamente en mi repositorio [12]. Las distintas formas de trazar los grafos son:

```
#Distintos algoritmos para trazar los grafos
nx.draw(G, pos=nx.spectral_layout(G), node_color='r', edge_color='b', with_labels=True)
nx.draw(G, pos=nx.circular_layout(G), node_color='r', edge_color='b', with_labels=True)
nx.draw(G, pos=nx.random_layout(G), node_color='r', edge_color='b', with_labels=True)
nx.draw(G, pos=nx.shell_layout(G), node_color='r', edge_color='b', with_labels=True)
nx.draw(G, pos=nx.spring_layout(G), node_color='r', edge_color='b', with_labels=True)
```

1. Algoritmo Circular

Este algoritmo cuenta con las siguientes propiedades [15]:

- Los nodos del grafo se encuentran sobre una misma circunferencia.
- Las aristas se representa con lineas rectas.
- Requieren a lo más $O(n)$ tiempo para su trazado, siendo n el número de aristas.
- Se utilizan particularmente para grafos en las que se desea mostrar claramente la biconectividad entre nodos.

1.1. Grafo simple no dirigido acíclico

Una red de ciudades en un estado puede ser un ejemplo de un grafo simple no dirigido acíclico, ya que podemos representar cada ciudad como un nodo

del grafo, y tomar en cuenta que cada arista representan las carreteras que conectan cada ciudad, considerando que sería un grafo no dirigido dado que una carretera puede ir de ida y vuelta entre ciudades, y es acíclico puesto que no se consideran carreteras que van de una ciudad a sí misma [8]. La figura 1 representa un ejemplo de este tipo de grafo.

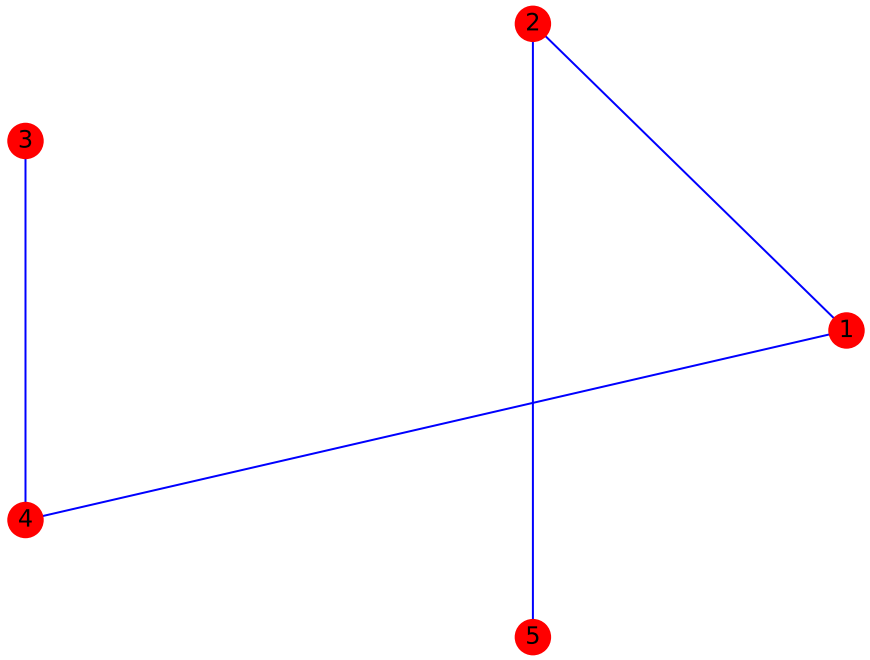


Figura 1: Grafo simple no dirigido acíclico.

1.2. Grafo simple no dirigido cíclico

Una ruta de autobús puede ser representada por este tipo de grafos, en donde los autobuses pasan a través de las calles y avenidas (representadas con las aristas), y cada nodo del grafo se puede representar una parada donde se recogen personas. La figura 2 representa un ejemplo de este tipo de grafo.

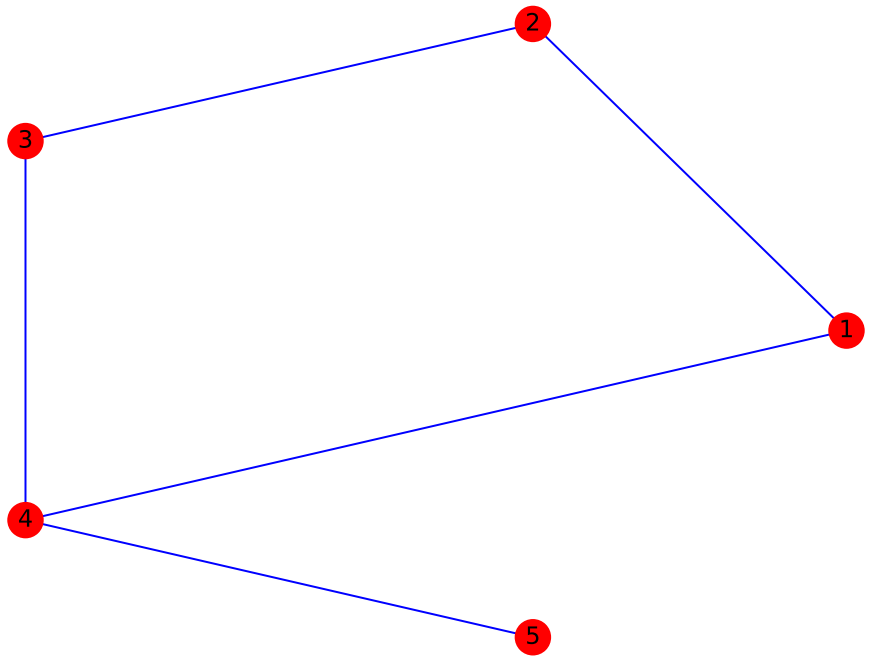


Figura 2: Grafo simple no dirigido cíclico.

1.3. Grafo simple dirigido cíclico

En algunos juegos se tienen distintos estados en los que el jugador se puede encontrar, y esto puede ser representado en un grafo dirigido cíclico, dado que cada estado se puede visualizar por medio de un nodo, y las aristas representarían las maneras en las que un estado puede cambiar a otro [9]. La figura 3 representa un ejemplo de este tipo de grafo.

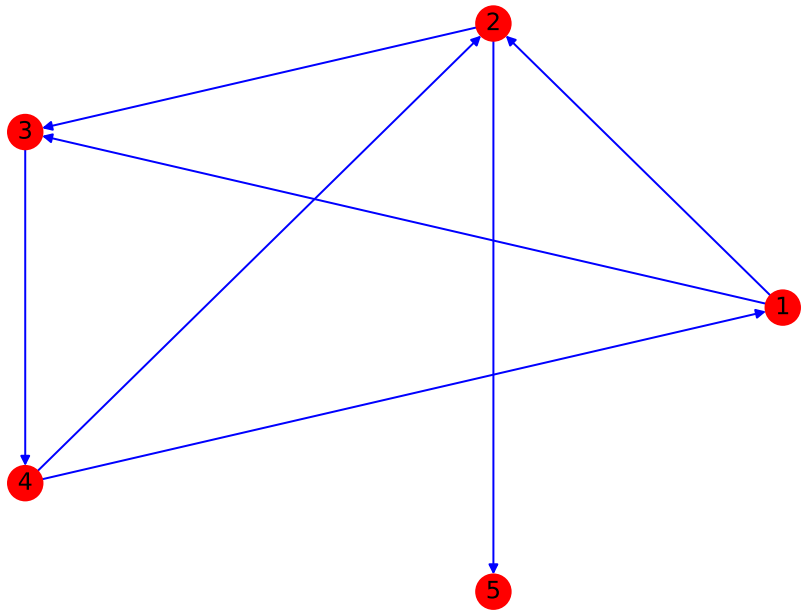


Figura 3: Grafo simple dirigido cíclico.

1.4. Multigrafo dirigido acíclico

Como en el grafo simple dirigido reflexivo, en una página web se pueden tener múltiples hipervínculos que te redirigen a una misma página web. La figura 9 representa un ejemplo de este tipo de grafo.

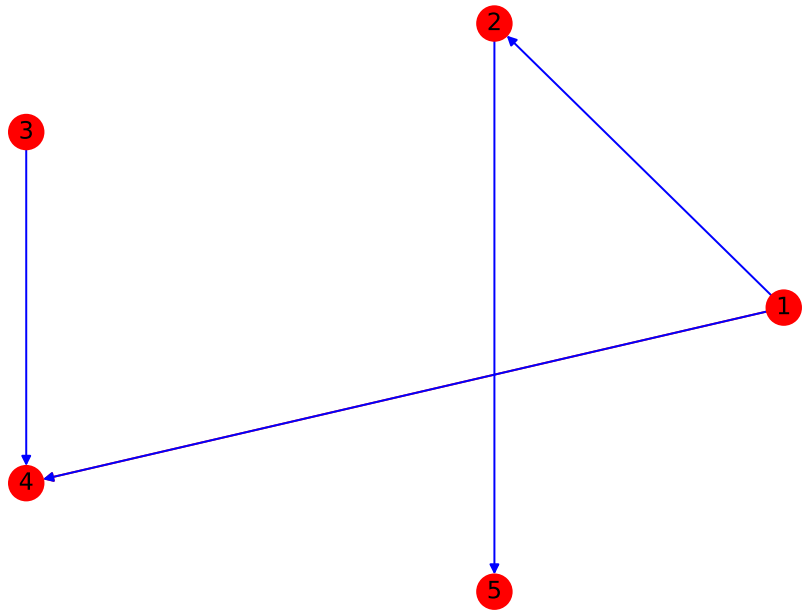


Figura 4: Multigrafo dirigido acíclico (los nodos 1 y 4 tienen múltiples aristas).

2. Algoritmo Random

Realiza el posicionamiento de los nodos de manera aleatoria, teniendo únicamente de restricción la ubicación de los nodos para no acomodar nodos encima de otros. Dada esta única restricción, su complejidad es nula [11].

2.1. Multigrafo no dirigido cíclico

Considerando el ejemplo de la ruta de autobuses, podemos tener entre dos paradas (nodos) múltiples rutas para llegar a la parada siguiente. La figura 5 representa un ejemplo de este tipo de grafo.

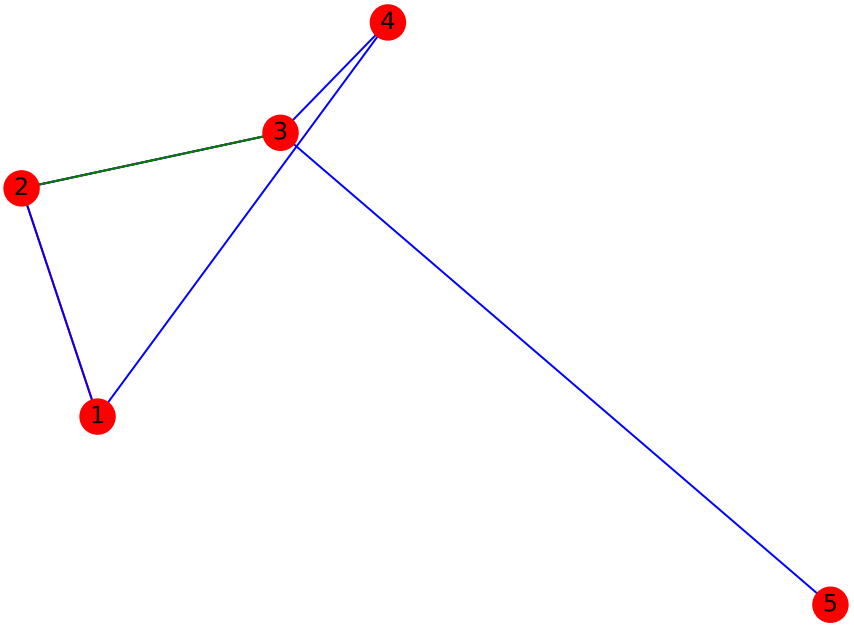


Figura 5: Multigrafo no dirigido cíclico (los nodos 1, 2 y 3 tienen múltiples aristas).

3. Algoritmo *Spectral*

El algoritmo *spectral* utiliza eigenvectores de la matriz del grafo (también conocido como el laplaciano). Dado un grafo ponderado, el algoritmo tiene como fin dar una relación entre los pesos de las aristas y la longitud de las aristas entre los nodos, esto es, entre mayor sea el peso de las aristas, más corta será la distancia entre los nodos [6].

3.1. Grafo simple dirigido reflexivo

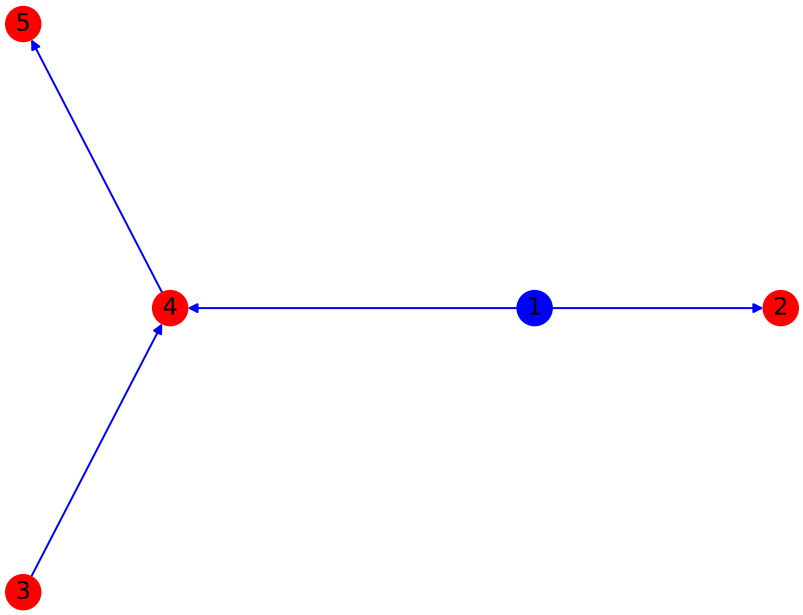


Figura 6: Grafo simple dirigido reflexivo (el nodo 1 tiene una arista reflexiva).

3.2. Multigrafo no dirigido acíclico

De manera similar al ejemplo de un grafo simple no dirigido acíclico, una red de ciudades puede ser representado por un multigrafo, el cual puede proporcionar más información que un grafo simple al añadir diversos caminos por el que se puede trasladar de un nodo a otro. La figura 7 representa un ejemplo de este tipo de grafo.

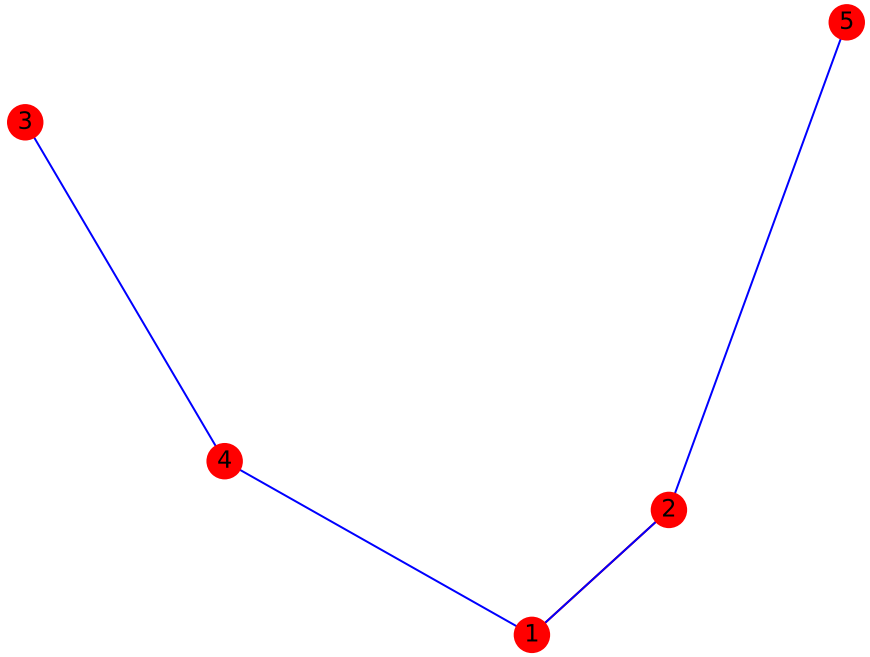


Figura 7: Multigrafo no dirigido acíclico (Los nodos 1 y 2 tienen múltiples aristas)

3.3. Multigrafo dirigido cíclico

Tomando el ejemplo utilizado en los grafos simples dirigidos cíclicos, en los juegos es posible que haya diferentes maneras de pasar de un estado a otro, lo cual es representado por aristas múltiples entre los nodos. La figura 8 representa un ejemplo de este tipo de grafo.

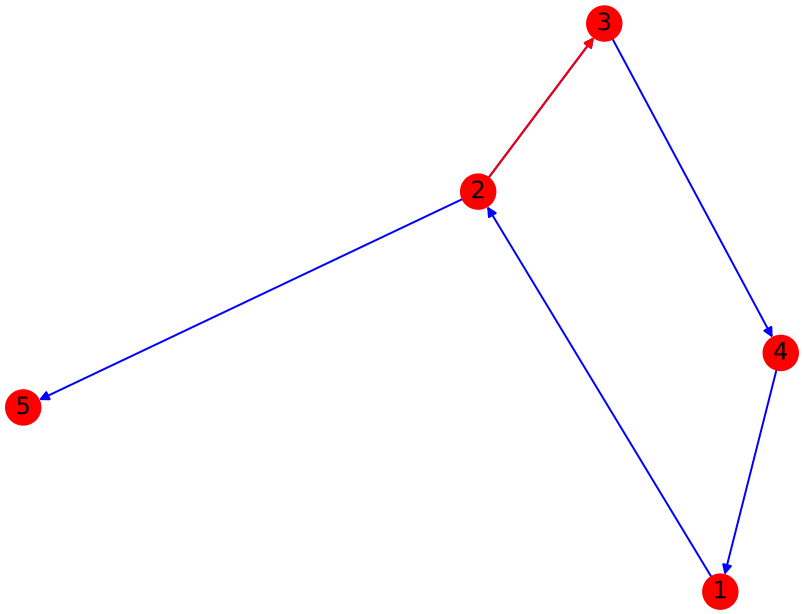


Figura 8: Multigrafo dirigido cíclico (los nodos 2 y 3 tienen múltiples aristas).

3.4. Multigrafo dirigido reflexivo

Como en el grafo simple dirigido reflexivo, en una página web se pueden tener múltiples hipervínculos que te redirigen a una misma página web. La figura 9 representa un ejemplo de este tipo de grafo.

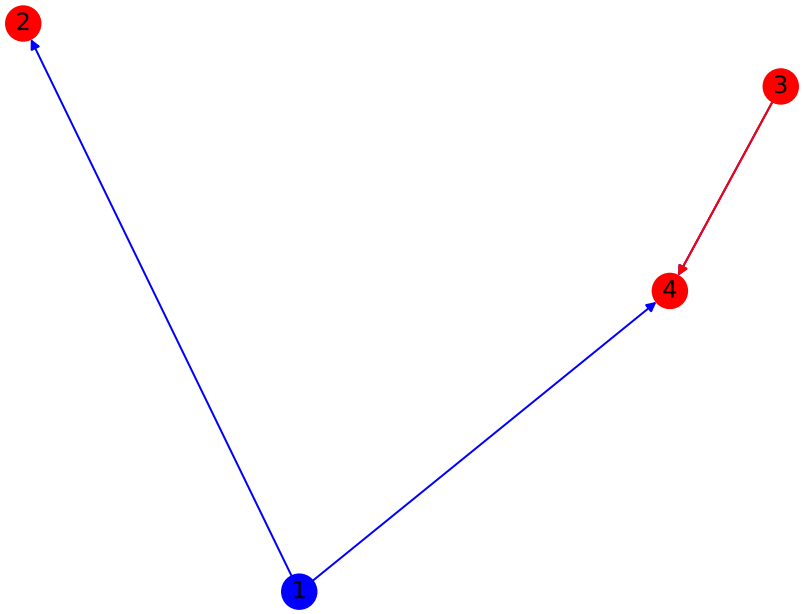


Figura 9: Multigrafo dirigido reflexivo (los nodos 3 y 4 tienen múltiples aristas, el nodo 1 tiene una arista reflexiva).

4. Algoritmo *Spring*

Basado en la física, el algoritmo *spring* trata de mantener que la suma de las fuerzas entre los nodos (los pesos de las aristas) sea de 0. Se basa esencialmente en la ley de Coulomb y en la métrica euclidiana para obtener estos valores, teniendo una complejidad computacional de $O(n^2)$ [2].

4.1. Grafo simple no dirigido reflexivo

Podemos representar una red de sistemas informáticos por medio de un grafo reflexivo, donde cada computadora es representada por medio de un nodo, y la conexión hacia las demás computadoras son representadas por las aristas. La arista reflexiva representaría la conexión de una computadora consigo misma [14]. La figura 10 representa un ejemplo de este tipo de grafo.

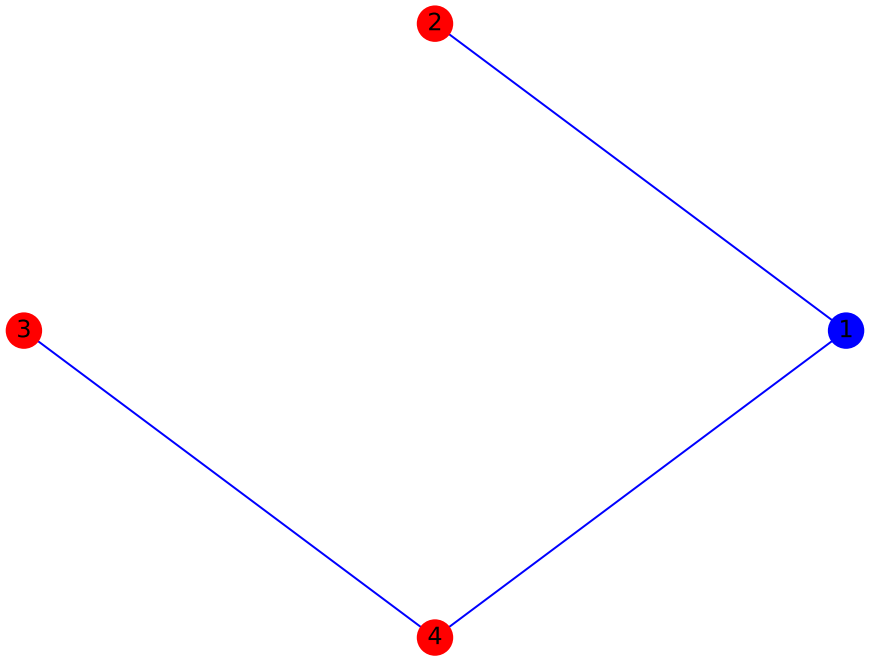


Figura 10: Grafo simple no dirigido reflexivo (el nodo 1 tiene una arista reflexiva).

4.2. Grafo simple dirigido acíclico

En la Programación Orientada a Objetos es común realizar herencia entre clases, y esta puede ser representada por medio de un grafo dirigido acíclico, teniendo a cada nodo como una clase distinta, y señalando la herencia por medio de una arista dirigida a las clases hijas [9]. La figura 11 representa un ejemplo de este tipo de grafo.

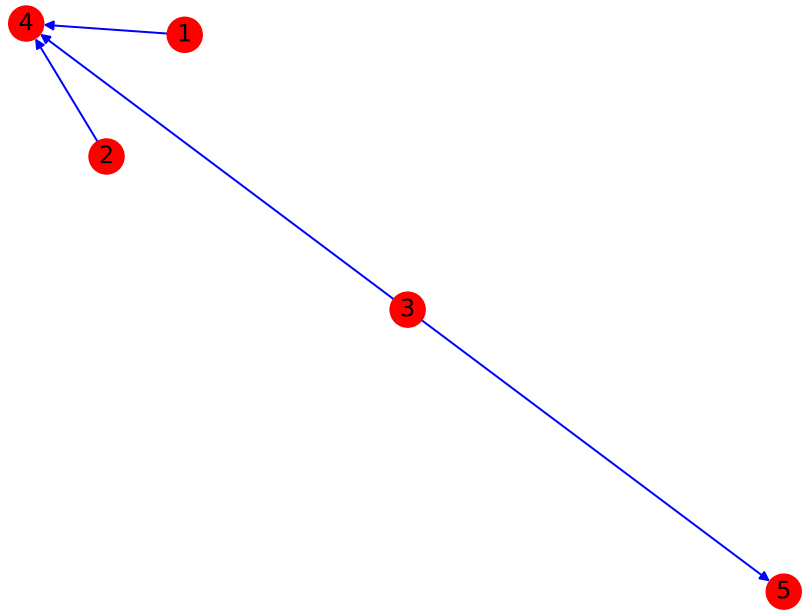


Figura 11: Grafo simple dirigido acíclico

5. Algoritmo *Shell*

Similar al algoritmo circular, el algoritmo *shell* realiza agrupaciones de nodos en círculos, con la diferencia que los nodos se pueden separar en distintos círculos a lo largo del grafo, por lo que la complejidad de este algoritmo es igual al algoritmo circular de $O(n)$ siendo n el número de aristas [10].

5.1. Multigrafo no dirigido reflexivo

En las redes sociales, se puede utilizar un multigrafo reflexivo para representar las menciones que se hacen entre usuarios por medio de publicaciones, donde cada nodo representa un usuario y cada publicación representa la arista con la que conecta el usuario que realizo la publicación con el que es mencionado. La figura 12 representa un ejemplo de este tipo de grafo.

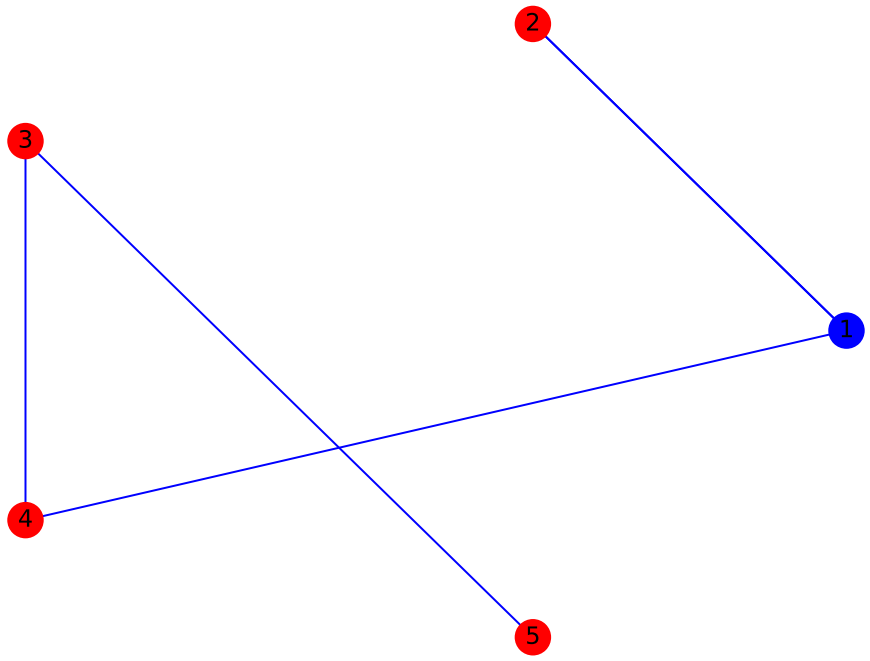


Figura 12: Multigrafo no dirigido reflexivo (los nodos 1 y 2 tienen múltiples aristas, el nodo 1 tiene una arista reflexiva).

6. Conclusiones entre algoritmos

Los algoritmos estudiados tienen distintos propósitos, siendo el tipo de visualización que se desea del grafo para determinar el algoritmo más adecuado. Por ejemplo, para simples pruebas de visualización se utilizaría el algoritmo *random* por su complejidad más simple entre los algoritmos, mientras que para otros tipos de grafos se utilizarían los otros tipos de algoritmos según el caso.

Referencias

[1] Python Software Foundation Versión 3.7.2. https://www.python.org/. Copyright ©2001-2019.

[2] Articulo. "Praktikum Algorithmen-Entwurf". http://wwwmayr.informatik.tu-muenchen.de/lehre/2012WS/algoprak/uebung/tutorial11.english.pdf.

[3] NetworkX developers con última actualización el 19 de Septiembre 2018. https://networkx.github.io/documentation/stable/index.html. Copyright ©2004-2018.

[4] NetworkX developers Versión 2.0. https://networkx.github.io/. Copyright ©2004-2018.

[5] The Matplotlib development team Versión 3.0.2. https://matplotlib.org/. Copyright ©2002-2012.

[6] Artículo: "Spectral Graph Drawing". http://www.cis.upenn.edu/ cis515/ -15-graph-drawing.pdf.

[7] Plotting NetworkX graph in Python Pregunta en Stackoverflow. https://stackoverflow.com/questions/44692644/plotting-networkx-graph-in-python. Copyright ©2019.

[8] E. Novo and A. Méndez Alonso. Aplicaciones de la teoría de grafos a algunos juegos de estrategia. *Suma*, 46:31–35, 2004.

[9] Example of Digraphs Applications Oxford Math Center. http://www.oxfordmathcenter.com/drupal7/node/678.

[10] Documentación oficial Networkx: Shell Layout. https://networkx.github.io/documentation/networkx-1.10/$_m$*odules/networkx/drawing/layout.html*$_s$*hell*$_l$*ayout*.

[11] Documentación oficial Rogue Wave Software. https://docs.roguewave.com/visualization/views/6.1/views.htmlpage/ Options/layouts.51.19.html.

[12] Jesús Angel Patlán Castillo. Repositorio Optimización Flujo en Redes. https://github.com/JAPatlanC/Flujo-Redes.

[13] "How to set colors for nodes in NetworkX" Respuesta de Abdallah Sobehy. https://stackoverflow.com/questions/27030473/how-to-set-colors-for-nodes-in-networkx-python.

[14] "What is the application of reflexive graph?Respuesta de Prenoy Kumar. https://www.quora.com/What-is-the-application-of-reflexive-graph.

[15] Janet M Six and Ioannis G Tollis. A framework for circular drawings of networks. In *International Symposium on Graph Drawing*, pages 107–116. Springer, 1999.

Tarea 2

Jesus Angel Patlán Castillo (5261)

26 de febrero de 2019

En esta tarea se analizan distintos tipos de acomodo visual grafos generados a partir de la librería NetworkX [4] de Python [1]. Se estudian particularmente los grafos utilizados para la tarea 1, la librería Matplotlib [5] para generar el grafo en los distintos algoritmos estudiados y para guardar el grafo en el formato ".eps". El código empleado se obtuvo consultando la documentación oficial de la librería NetworkX [3] y guías suplementarias [7, 15]. Las imágenes y el código se encuentran disponibles directamente en mi repositorio [13]. Las distintas formas de trazar los grafos son:

```
#Distintos algoritmos para trazar los grafos
nx.draw(G, pos=nx.spectral_layout(G), node_color='r', edge_color='b', with_labels=True)
nx.draw(G, pos=nx.circular_layout(G), node_color='r', edge_color='b', with_labels=True)
nx.draw(G, pos=nx.random_layout(G), node_color='r', edge_color='b', with_labels=True)
nx.draw(G, pos=nx.shell_layout(G), node_color='r', edge_color='b', with_labels=True)
nx.draw(G, pos=nx.spring_layout(G), node_color='r', edge_color='b', with_labels=True)
```

1. Algoritmo Circular

Este algoritmo cuenta con las siguientes propiedades [14]:

- Los nodos del grafo se encuentran sobre una misma circunferencia.
- Las aristas se representa con lineas rectas.
- Requieren a lo más $O(n)$ tiempo para su trazado, siendo n el número de aristas.
- Se utilizan particularmente para grafos en las que se desea mostrar claramente la biconectividad entre nodos.

1.1. Grafo simple no dirigido acíclico

Una red de ciudades en un estado puede ser un ejemplo de un grafo simple no dirigido acíclico, ya que cada ciudad se representa como un nodo del grafo,

y se toma en cuenta que cada arista representan las carreteras que conectan cada ciudad, considerando que sería un grafo no dirigido dado que una carretera puede ir de ida y vuelta entre ciudades, y es acíclico puesto que no se consideran carreteras que van de una ciudad a sí misma [9]. La figura 1 representa un ejemplo de este tipo de grafo.

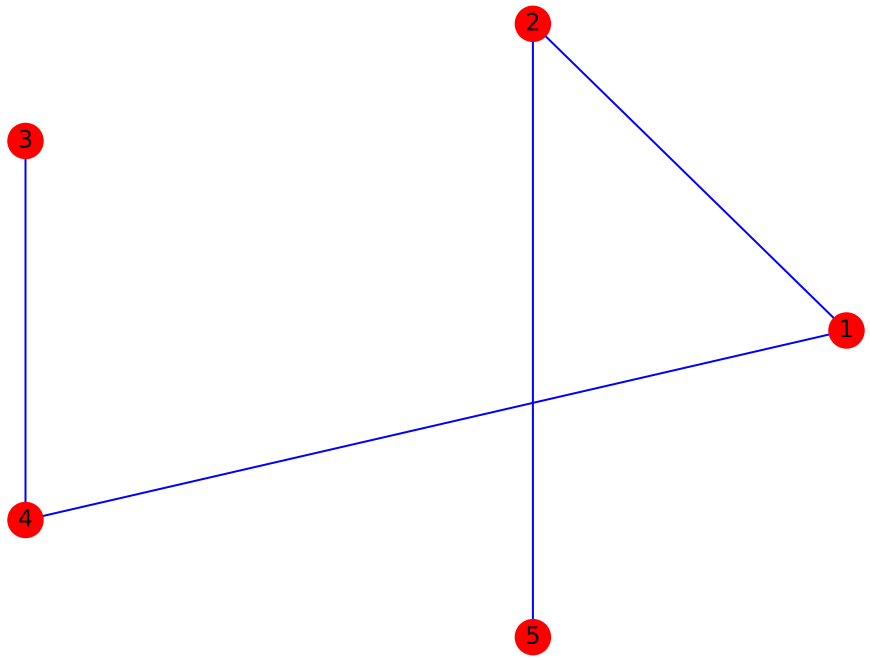


Figura 1: Grafo simple no dirigido acíclico.

1.2. Grafo simple no dirigido cíclico

Una ruta de autobús puede ser representada por este tipo de grafos, en donde los autobuses pasan a través de las calles y avenidas (representadas con las aristas), y cada nodo del grafo se puede representar una parada donde se recogen personas. La figura 2 representa un ejemplo de este tipo de grafo.

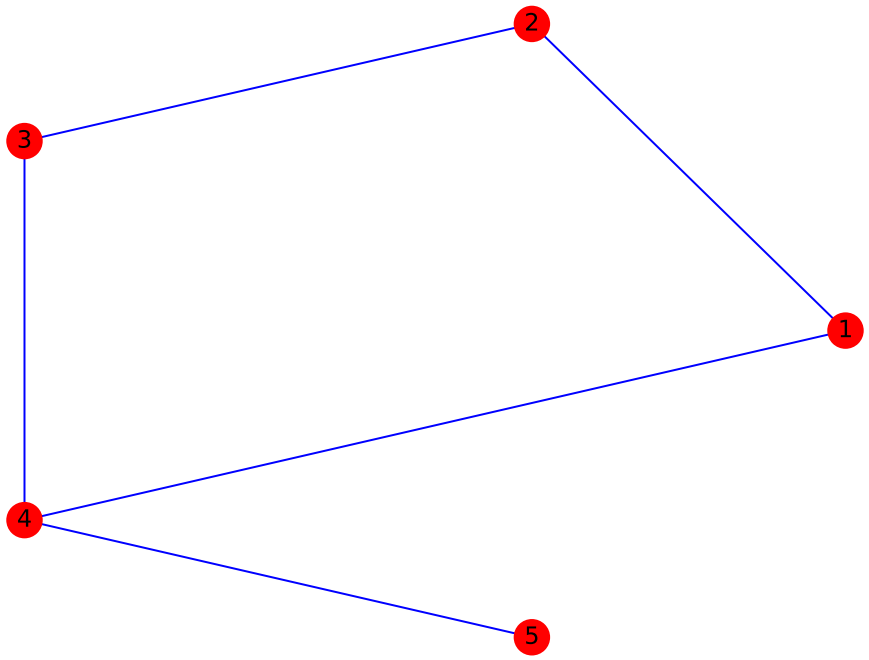


Figura 2: Grafo simple no dirigido cíclico.

1.3. Grafo simple dirigido cíclico

En algunos juegos se tienen distintos estados en los que el jugador se puede encontrar, y esto puede ser representado en un grafo dirigido cíclico, dado que cada estado se puede visualizar por medio de un nodo, y las aristas representarían las maneras en las que un estado puede cambiar a otro [10]. La figura 3 representa un ejemplo de este tipo de grafo.

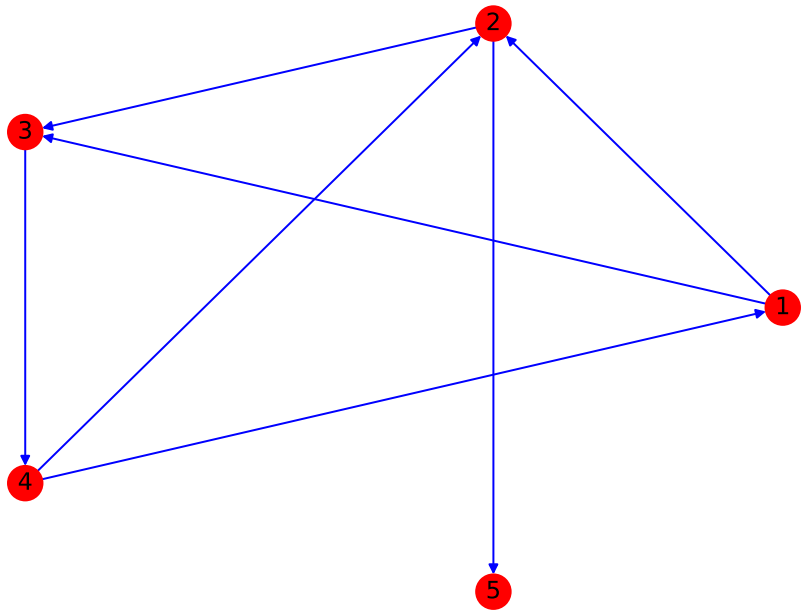


Figura 3: Grafo simple dirigido cíclico.

1.4. Multigrafo dirigido acíclico

Como en el grafo simple dirigido reflexivo, en una página web se pueden tener múltiples hipervínculos que te redirigen a una misma página web. La figura 9 representa un ejemplo de este tipo de grafo.

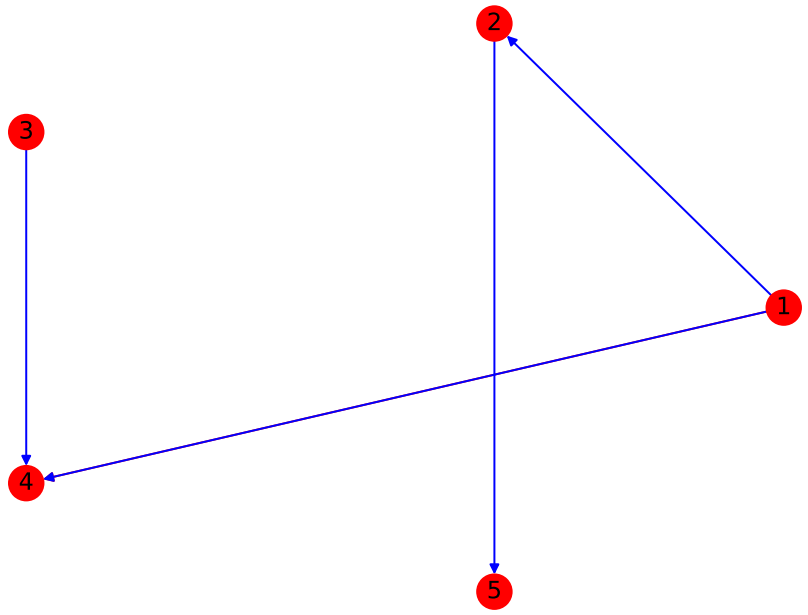


Figura 4: Multigrafo dirigido acíclico (los nodos 1 y 4 tienen múltiples aristas).

2. Algoritmo Aleatorizado

Realiza el posicionamiento de los nodos de manera aleatoria, teniendo únicamente de restricción la ubicación de los nodos para no acomodar nodos encima de otros. Dada esta única restricción, su complejidad es nula [12].

2.1. Multigrafo no dirigido cíclico

Considerando el ejemplo de la ruta de autobuses, podemos tener entre dos paradas (nodos) múltiples rutas para llegar a la parada siguiente. La figura 5 representa un ejemplo de este tipo de grafo.

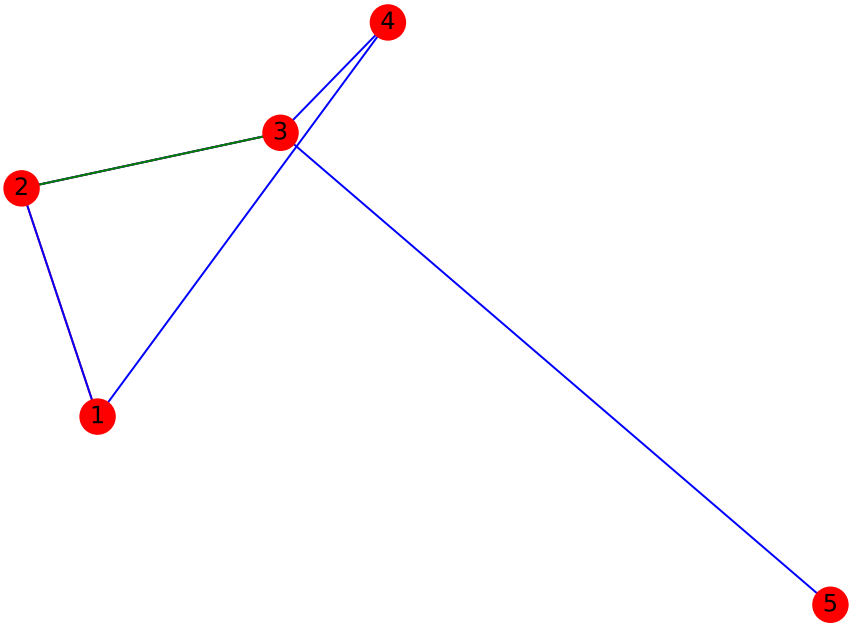


Figura 5: Multigrafo no dirigido cíclico (los nodos 1, 2 y 3 tienen múltiples aristas).

3. Algoritmo Espectral

El algoritmo espectral utiliza eigenvectores de la matriz del grafo (también conocido como el laplaciano). Dado un grafo ponderado, el algoritmo tiene como fin dar una relación entre los pesos de las aristas y la longitud de las aristas entre los nodos, esto es, entre mayor sea el peso de las aristas, más corta será la distancia entre los nodos [6].

3.1. Grafo simple dirigido reflexivo

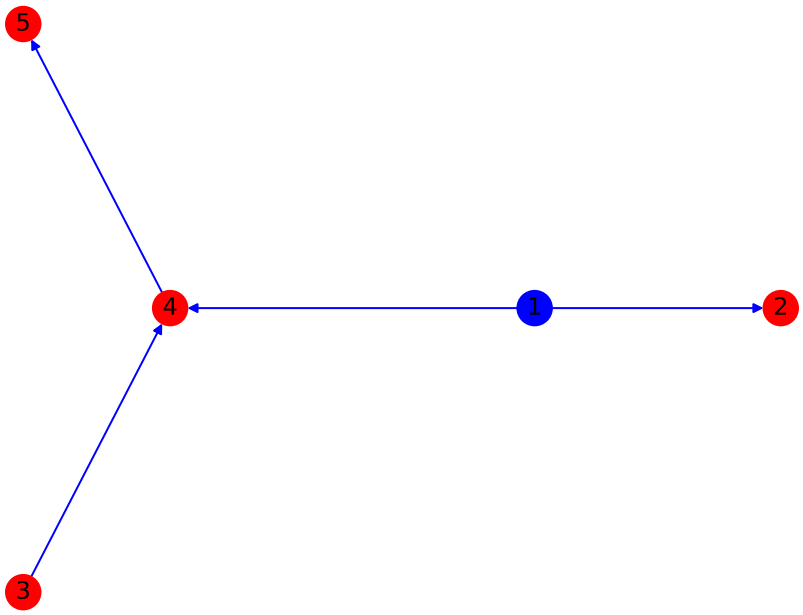


Figura 6: Grafo simple dirigido reflexivo (el nodo 1 tiene una arista reflexiva).

3.2. Multigrafo no dirigido acíclico

De manera similar al ejemplo de un grafo simple no dirigido acíclico, una red de ciudades puede ser representado por un multigrafo, el cual puede proporcionar más información que un grafo simple al añadir diversos caminos por el que se puede trasladar de un nodo a otro. La figura 7 representa un ejemplo de este tipo de grafo.

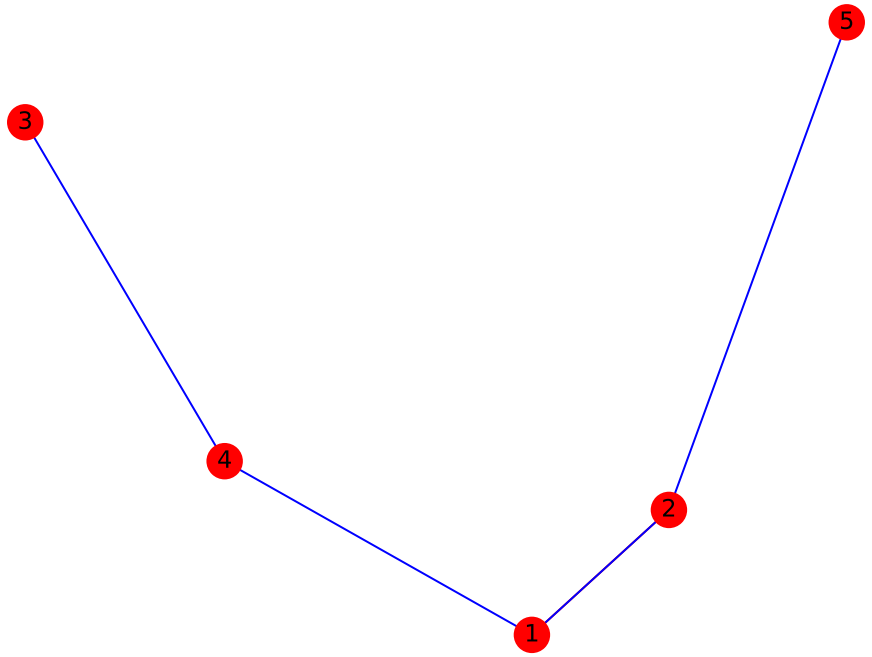


Figura 7: Multigrafo no dirigido acíclico (los nodos 1 y 2 tienen múltiples aristas)

3.3. Multigrafo dirigido cíclico

Tomando el ejemplo utilizado en los grafos simples dirigidos cíclicos, en los juegos es posible que haya diferentes maneras de pasar de un estado a otro, lo cual es representado por aristas múltiples entre los nodos. La figura 8 representa un ejemplo de este tipo de grafo.

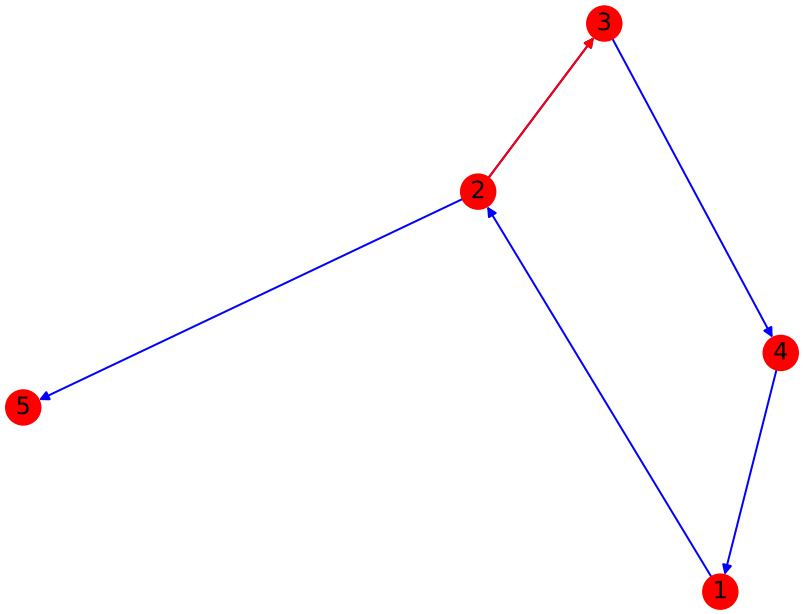


Figura 8: Multigrafo dirigido cíclico (los nodos 2 y 3 tienen múltiples aristas).

3.4. Multigrafo dirigido reflexivo

Como en el grafo simple dirigido reflexivo, en una página web se pueden tener múltiples hipervínculos que redirigen a una misma página web. La figura 9 representa un ejemplo de este tipo de grafo.

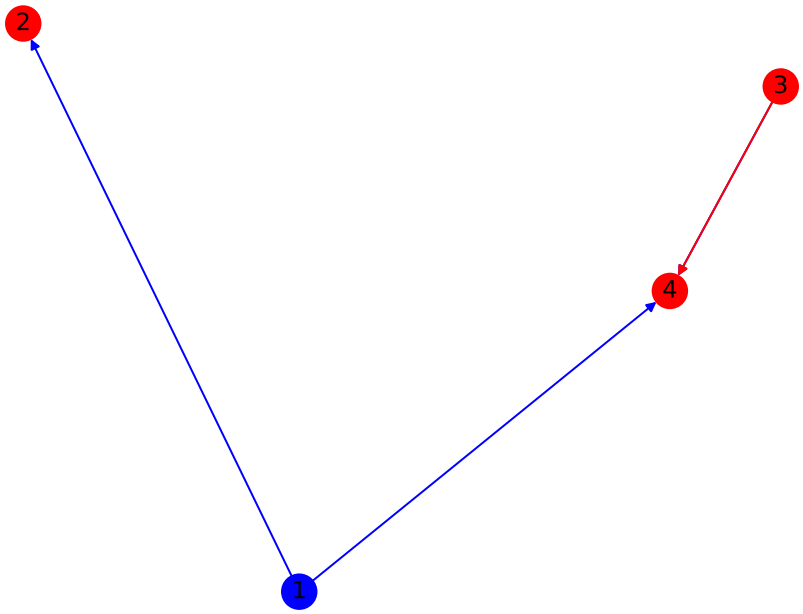


Figura 9: Multigrafo dirigido reflexivo (los nodos 3 y 4 tienen múltiples aristas, el nodo 1 tiene una arista reflexiva).

4. Algoritmo Resorte

Basado en la física, el algoritmo resorte trata de mantener que la suma de las fuerzas entre los nodos (los pesos de las aristas) sea de 0. Se basa esencialmente en la ley de Coulomb y en la métrica euclidiana para obtener estos valores, teniendo una complejidad computacional de $O(n^2)$ [2].

4.1. Grafo simple no dirigido reflexivo

Podemos representar una red de sistemas informáticos por medio de un grafo reflexivo, donde cada computadora es representada por medio de un nodo, y la conexión hacia las demás computadoras son representadas por las aristas. La arista reflexiva representa la conexión de una computadora consigo misma [8]. La figura 10 representa un ejemplo de este tipo de grafo.

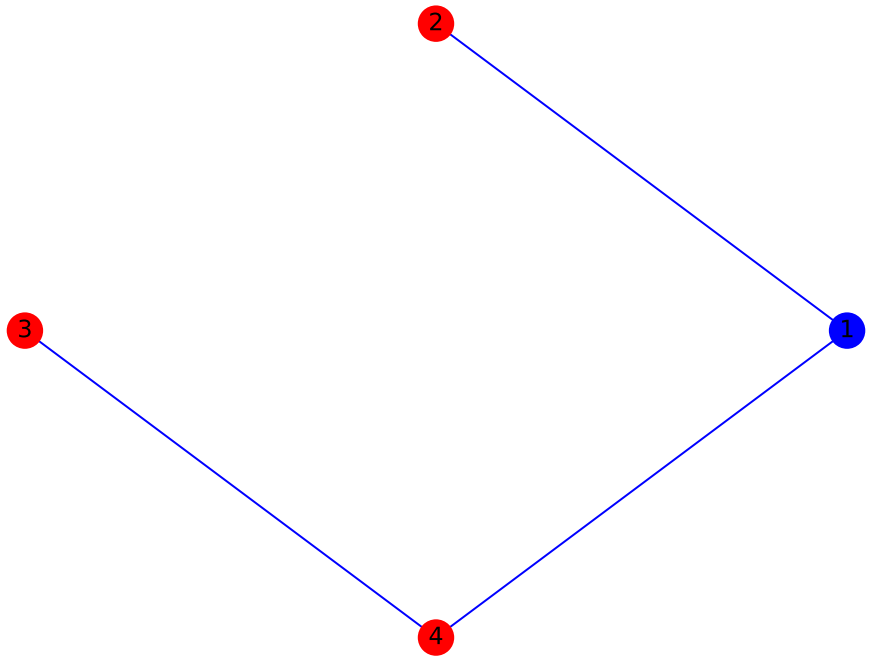


Figura 10: Grafo simple no dirigido reflexivo (el nodo 1 tiene una arista reflexiva).

4.2. Grafo simple dirigido acíclico

En la Programación Orientada a Objetos es común realizar herencia entre clases, y esta puede ser representada por medio de un grafo dirigido acíclico, teniendo a cada nodo como una clase distinta, y señalando la herencia por medio de una arista dirigida a las clases hijas [10]. La figura 11 representa un ejemplo de este tipo de grafo.

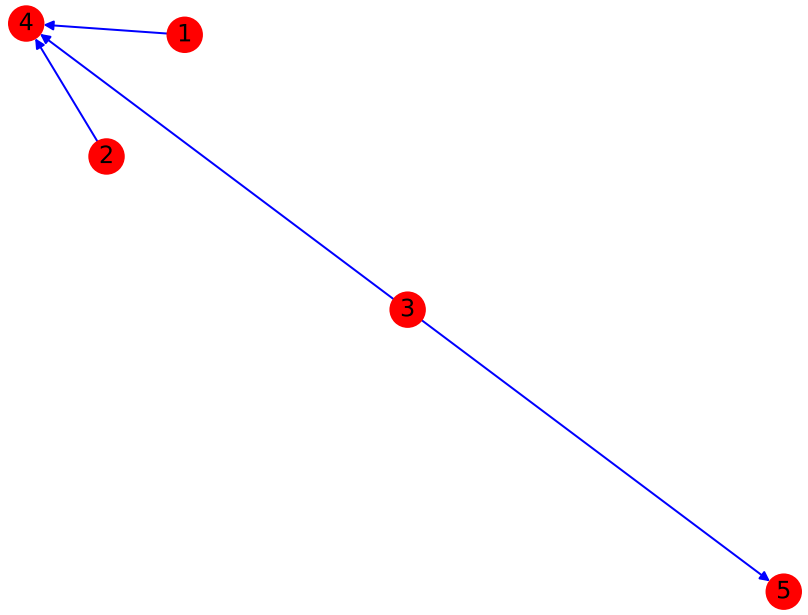


Figura 11: Grafo simple dirigido acíclico

5. Algoritmo Cascaron

Similar al algoritmo circular, el algoritmo cascaron realiza agrupaciones de nodos en círculos, con la diferencia que los nodos se pueden separar en distintos círculos a lo largo del grafo, por lo que la complejidad de este algoritmo es igual al algoritmo circular de $O(n)$ siendo n el número de aristas [11].

5.1. Multigrafo no dirigido reflexivo

En las redes sociales, se puede utilizar un multigrafo reflexivo para representar las menciones que se hacen entre usuarios por medio de publicaciones, donde cada nodo representa un usuario y cada publicación representa la arista con la que conecta el usuario que realizo la publicación con el que es mencionado. La figura 12 representa un ejemplo de este tipo de grafo.

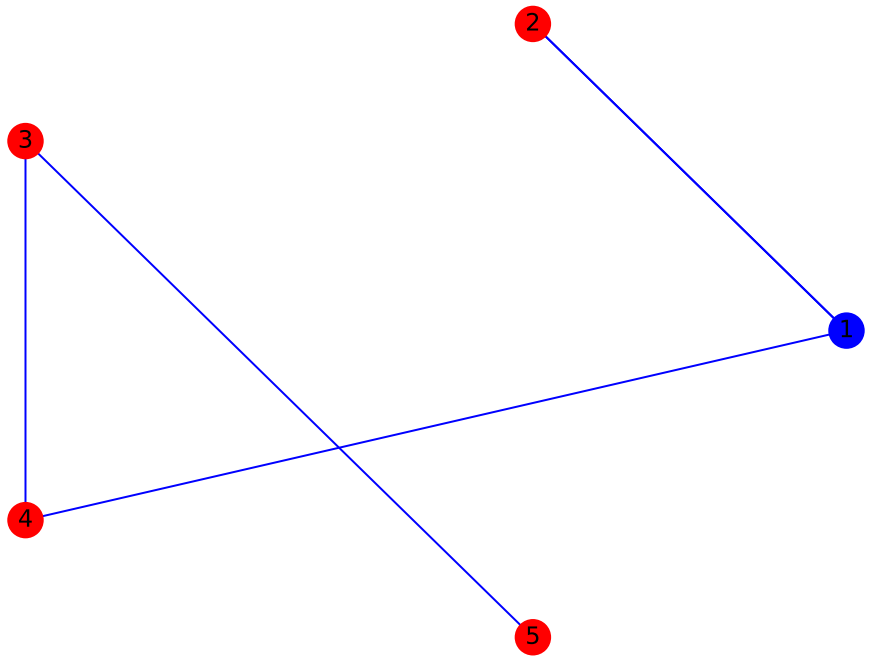


Figura 12: Multigrafo no dirigido reflexivo (los nodos 1 y 2 tienen múltiples aristas, el nodo 1 tiene una arista reflexiva).

6. Conclusiones entre algoritmos

Los algoritmos estudiados tienen distintos propósitos, siendo el tipo de visualización que se desea del grafo para determinar el algoritmo más adecuado. Por ejemplo, para simples pruebas de visualización se utilizaría el algoritmo aleatorio por su complejidad más simple entre los algoritmos.

Referencias

[1] Python Software Foundation Versión 3.7.2. `https://www.python.org/`.

[2] "Praktikum Algorithmen-Entwurf". `http://wwwmayr.informatik.tu-muenchen.de/lehre/2012WS/algoprak/uebung/tutorial11.english.pdf`.

[3] NetworkX developers con última actualización el 19 de Septiembre 2018. `https://networkx.github.io/documentation/stable/index.html`.

[4] NetworkX developers Versión 2.0. `https://networkx.github.io/`.

[5] The Matplotlib development team Versión 3.0.2. https://matplotlib.org/.

[6] "Spectral Graph Drawing". http://www.cis.upenn.edu/~cis515/-15-graph-drawing.pdf.

[7] Plotting NetworkX graph in Python Pregunta en Stackoverflow. https://stackoverflow.com/questions/44692644/plotting-networkx-graph-in-python.

[8] What is the application of reflexive graph? https://www.quora.com/What-is-the-application-of-reflexive-graph.

[9] E. Novo and A. Méndez Alonso. *Aplicaciones de la teoría de grafos a algunos juegos de estrategia*, volume 46. 2004.

[10] Example of Digraphs Applications Oxford Math Center. http://www.oxfordmathcenter.com/drupal7/node/678.

[11] Documentación oficial Networkx: Shell Layout. https://networkx.github.io/documentation/networkx-1.10/_modules/networkx/drawing/layout.html#shell_layout.

[12] Documentación oficial Rogue Wave Software. https://docs.roguewave.com/visualization/views/6.1/views.html#page/Options/layouts.51.19.html.

[13] J. A. Patlán Castillo. Repositorio optimización flujo en redes. https://github.com/JAPatlanC/Flujo-Redes.

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

[15] How to set colors for nodes in NetworkX. https://stackoverflow.com/questions/27030473/how-to-set-colors-for-nodes-in-networkx-python.

4. Tarea 3

Para esta tarea:

- Se realizaron pequeñas correcciones al documento de diseño y errores ortográficos.

Dado que las correcciones no muestran un cambio significativo al documento, se decide no mostrar la nueva versión.

Tarea 3

Jesus Angel Patlán Castillo (5261)

19 de marzo de 2019

En esta tarea se analizan 5 distintos algoritmos en grafos midiendo su tiempo de ejecución en base a grafos realizados tareas anteriores. Los algoritmos se obtuvieron por medio de la librería NetworkX [3] de Python [1], la librería Matplotlib [4] es utilizada para generar las gráficas de comparación entre los distintos algoritmos estudiados. El código empleado se obtuvo consultando la documentación oficial de la librería NetworkX [2]. Las imágenes y el código se encuentran disponibles directamente en mi repositorio [5].

1. Metodología

La librería NetworkX tiene a la mano diversos algoritmos para resolver distintos problemas que se pueden presentar en los grafos. Particularmente para esta investigación, se realiza el análisis de los algoritmos:

- Ruta más corta: El algoritmo de la ruta más corta, como lo dice el nombre del algoritmo, trata de buscar la ruta más corta entre dos nodos dentro de un grafo. En la librería de NetworkX esta disponible por medio de la función "all shortest paths(G,N1,N2)", donde el parámetro G es el grafo, N1 el nodo origen y N2 el nodo destino:

```
1        nx.all_shortest_paths(G1,1,5)
```

- Árbol búsqueda a profundidad: El algoritmo de árbol búsqueda a profundidad recorre el grafo dado para generar un árbol por medio de una búsqueda a profundidad. En la librería de NetworkX esta disponible por medio de la función "dfs tree(G)", donde el parámetro G es el grafo:

```
1        nx.dfs_tree(G6)
```

- Problema de la liga de amigos: El algoritmo para el problema de la liga de amigos trata de encontrar el nodo con mayor clique", es decir, el que posee un mayor de número de nodos conectados. En la librería de NetworkX esta disponible por medio de la función "max clique(G)", donde el parámetro G es el grafo:

```
1        nx.make_max_clique_graph(G11)
```

- Árbol de mínimo grado: El algoritmo de árbol de mínimo grado recorre entre los nodos del grafo para determinar el nodo con grado mínimo. En la librería de NetworkX esta disponible por medio de la función "treewidth min degree(G)", donde el parámetro G es el grafo:

```
tree.treewidth_min_degree(G11)
```

- Árbol de mínima expansión: El algoritmo de árbol de expansión mínima recorre el grafo proporcionado para obtener el árbol con mínima expansión que se pueda generar en él. En la librería de NetworkX esta disponible por medio de la función "minimum spanning tree(G)", donde el parámetro G es el grafo:

```
nx.minimum_spanning_tree(G11)
```

Cada uno de estos algoritmos fueron ejecutados sobre 5 grafos diferentes, siendo estos 5 grafos simples con ciclos no dirigidos. Para el caso de la ruta más corta, se usaron grafos con la misma cantidad de nodos y aristas, pero cambiando el peso de aristas. Para los demás algoritmos, cada uno de los grafos, enumerados del 1 al 5, contiene un número mayor de aristas y nodos conforme incrementa su numeración, es decir, el grafo 1 es el de menor número de aristas y nodos, mientras que el grafo 5 es el de mayor número de aristas y nodos. Para cada grafo cuales se realizo la cantidad de repeticiones posibles para que el tiempo mínimo de ejecución fueran de 5 segundos, y posteriormente a este conjunto de repeticiones por algoritmos se realizaron 30 réplicas distintas. Con estos datos, se obtuvo la media y la desviación estándar de cada una de las muestras obtenidas, además de realizar un histograma y un diagrama caja bigote para el análisis de cada uno de los algoritmos. Finalmente, se analiza el resultado final por medio de una gráfica de dispersión entre los datos de las 5 muestras, comparando el tiempo de ejecución contra la cantidad de nodos y contra la cantidad de aristas de cada grafo. A continuación se muestra el código para la generación del histograma, diagrama caja y bigote, y el histograma con la librería NetworkX, Matplot y Sciplot:

```
n, bins, patches = plt.hist(a1_times, 'auto', density=True,
    facecolor='blue', alpha=0.75)
y = scipy.stats.norm.pdf(bins, a1_mean,a1_standard)
plt.plot(bins, y, 'r--')
plt.xlabel('Tiempo (segundos)')
plt.ylabel('Frecuencia')
plt.title('Ruta m s corta (Media='+str(round(a1_mean,2))+' ,STD='+
    str(round(a1_standard,2))+' )',size=18, color='green')
plt.savefig('H1.eps', format='eps', dpi=1000)
plt.show()

#  Boxplots
to_plot=[a1_t1_times,a1_t2_times,a1_t3_times,a1_t4_times,
    a1_t5_times]
fig=plt.figure(1,figsize=(9,6))
ax=fig.add_subplot(111)
bp=ax.boxplot(to_plot, showfliers=False)
```

```
plt.xlabel('Grafo')
plt.ylabel('Tiempo (segundos)')
plt.title('Ruta m s corta')
plt.savefig('BP1.eps', format='eps', dpi=1000)
```

```
to_plot=[a1_t1_times,a1_t2_times,a1_t3_times,a1_t4_times,
    a1_t5_times]
fig=plt.figure(1,figsize=(9,6))
ax=fig.add_subplot(111)
bp=ax.boxplot(to_plot, showfliers=False)
plt.xlabel('Grafo')
plt.ylabel('Tiempo (segundos)')
plt.title('Ruta m s corta')
plt.savefig('BP1.eps', format='eps', dpi=1000)
plt.show()
```

```
plt.errorbar(y_1,x_1_1, xerr=z_1, fmt='o',color='blue',alpha=0.5)
plt.errorbar(y_2,x_1_2, xerr=z_2, fmt='s',color='yellow',alpha=0.5)
plt.errorbar(y_3,x_1_3, xerr=z_3, fmt='*',color='green',alpha=0.5)
plt.errorbar(y_4,x_1_4, xerr=z_4, fmt='h',color='red',alpha=0.5)
plt.errorbar(y_5,x_1_5, xerr=z_5, fmt='D',color='orange',alpha=0.5)
plt.xlabel('Tiempo (segundos)', size=14)
plt.ylabel('Nodos', size=14)
plt.title('Nodos vs tiempo',size=18)
plt.savefig('S1.eps', format='eps', dpi=1000)
plt.show()
```

Este proceso se realizó en una laptop con las siguientes características:

- Procesador: Intel Core i7-7500U 2.7GHz
- Memoria RAM: 16GB
- Sistema Operativo: Windows 10 64 bits

2. Resultados

2.1. Ruta más corta

A continuación se muestra el histograma en la figura 1 y el diagrama caja y bigote en la figura 2 con los resultados obtenidos:

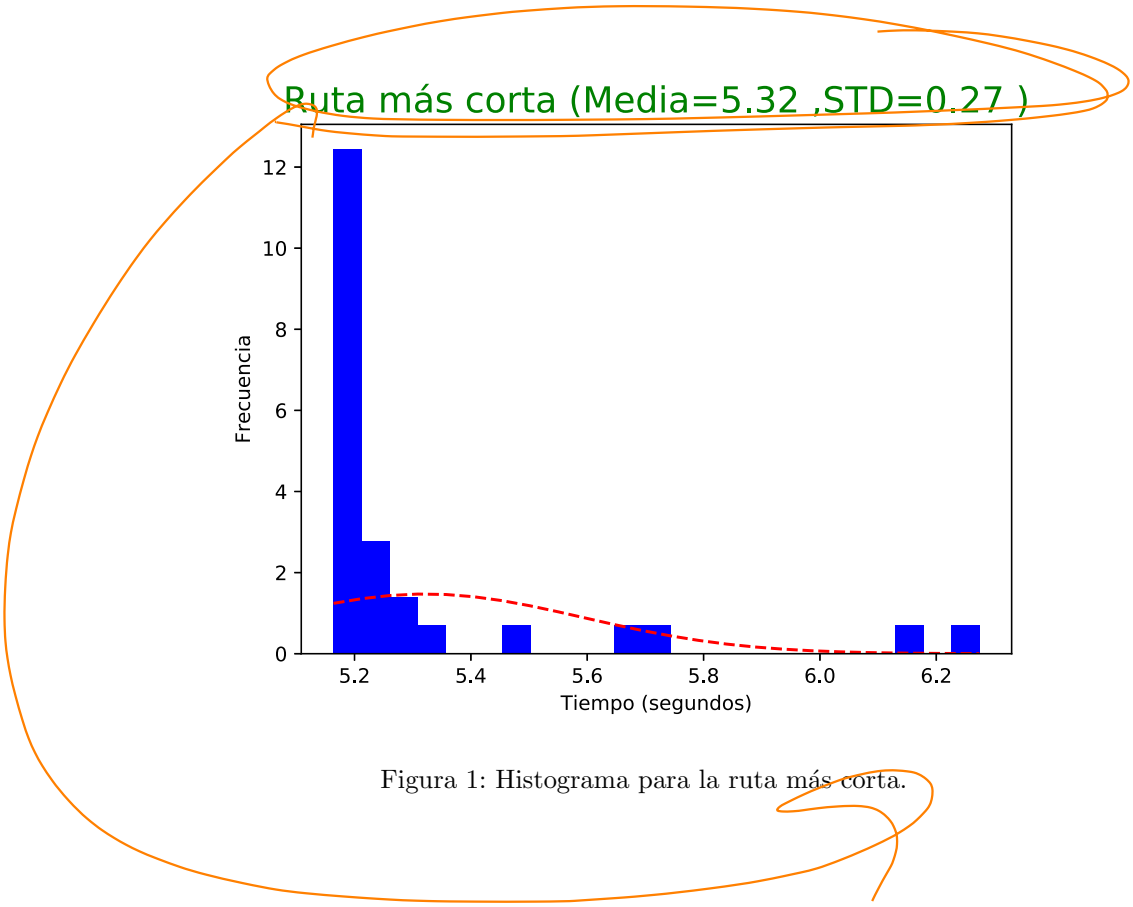


Figura 1: Histograma para la ruta más corta.

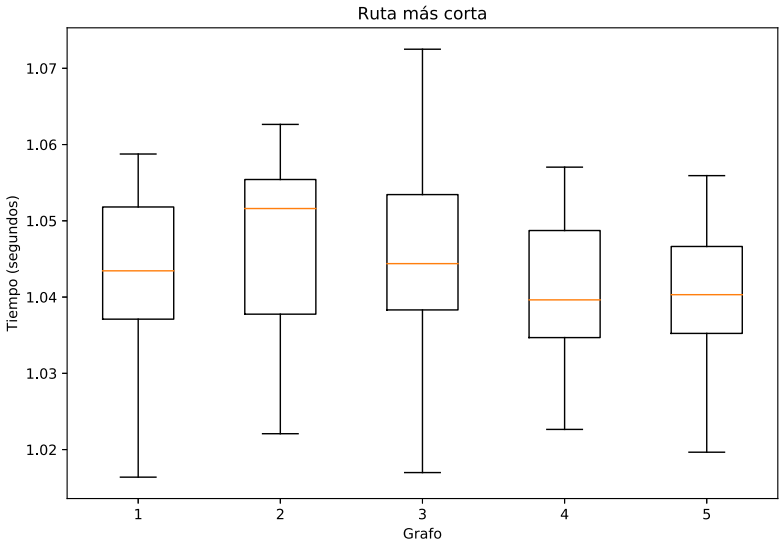


Figura 2: Diagrama caja y bigote para la ruta más corta.

En este caso, las gráficas reflejan que no hay mucha diferencia entre los 5 grafos al momento de aplicar el algoritmo de la ruta más corta, por lo que el cambio de pesos entre los grafos no afecta el tiempo de ejecución.

2.2. Árbol búsqueda a profundidad

A continuación se muestra el histograma en la figura 3 y el diagrama caja y bigote en la figura 4 con los resultados obtenidos:

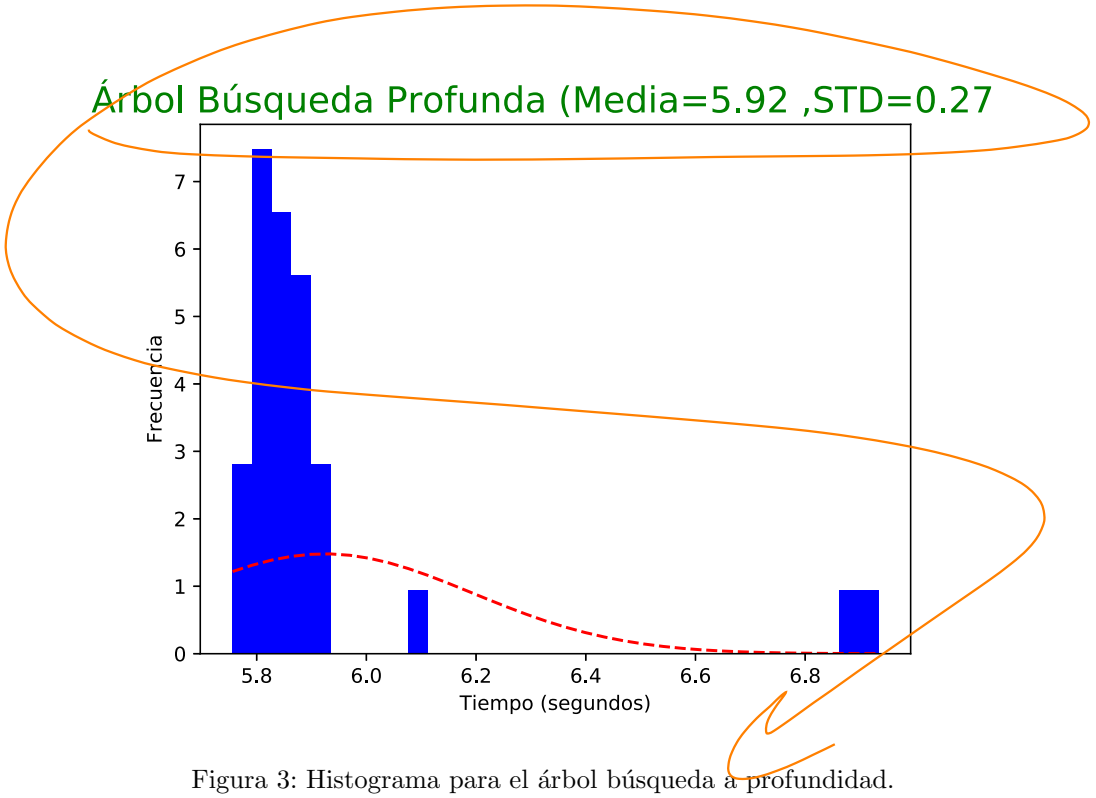


Figura 3: Histograma para el árbol búsqueda a profundidad.

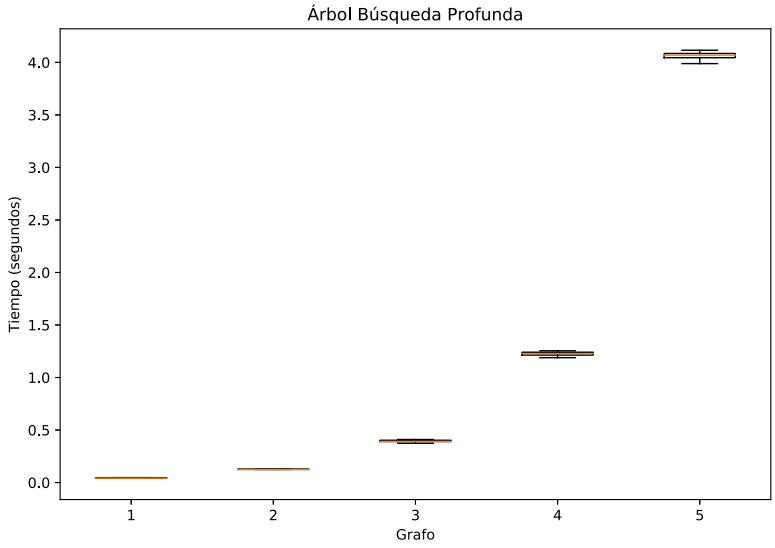


Figura 4: Diagrama caja y bigote para el árbol búsqueda a profundidad.

A diferencia del algoritmo de la ruta más corta, en este tipo de algoritmo si existe una clara diferencia en el tiempo de ejecución entre los grafos, por lo que un aumento en la cantidad de nodos y aristas aumenta el tiempo de ejecución.

2.3. Problema de la liga de amigos

A continuación se muestra el histograma en la figura 5 y el diagrama caja y bigote en la figura 6 con los resultados obtenidos:

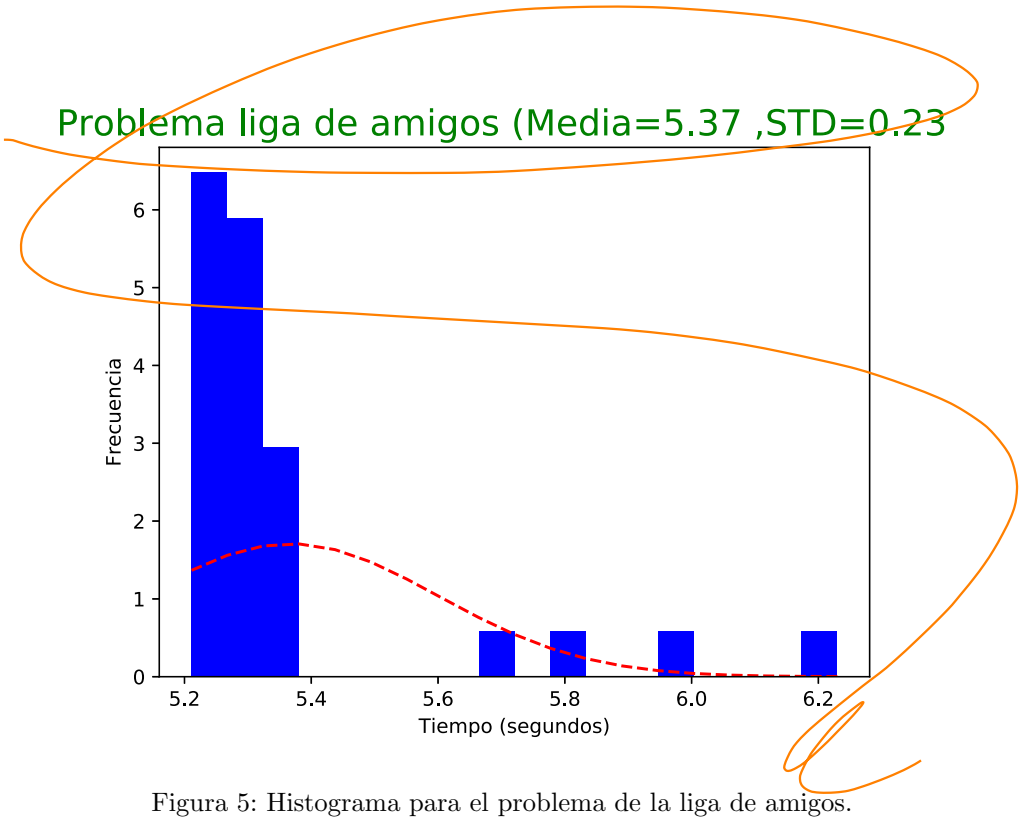


Figura 5: Histograma para el problema de la liga de amigos.

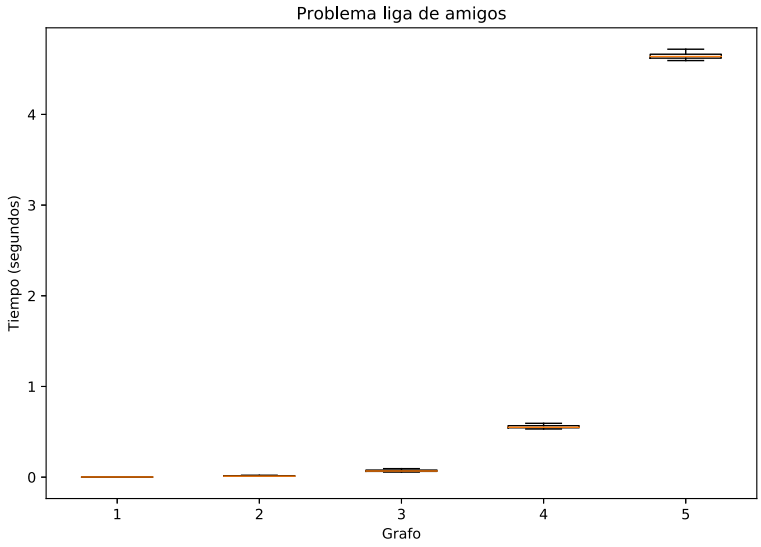


Figura 6: Diagrama caja y bigote para el problema de la liga de amigos.

Las gráficas muestran una clara diferencia en el tiempo de ejecución conforme el número de nodos aumenta en el grafo.

2.4. Árbol de mínimo grado

A continuación se muestra el histograma en la figura 7 y el diagrama caja y bigote en la figura 8 con los resultados obtenidos:

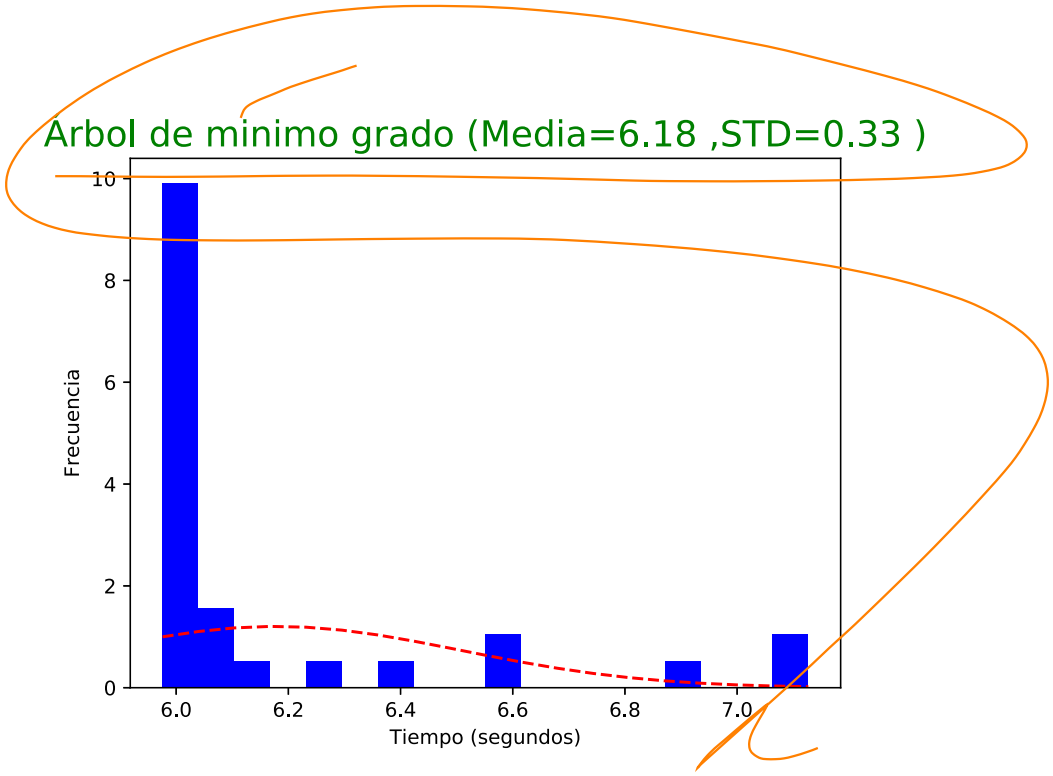


Figura 7: Histograma para el árbol de mínimo grado.

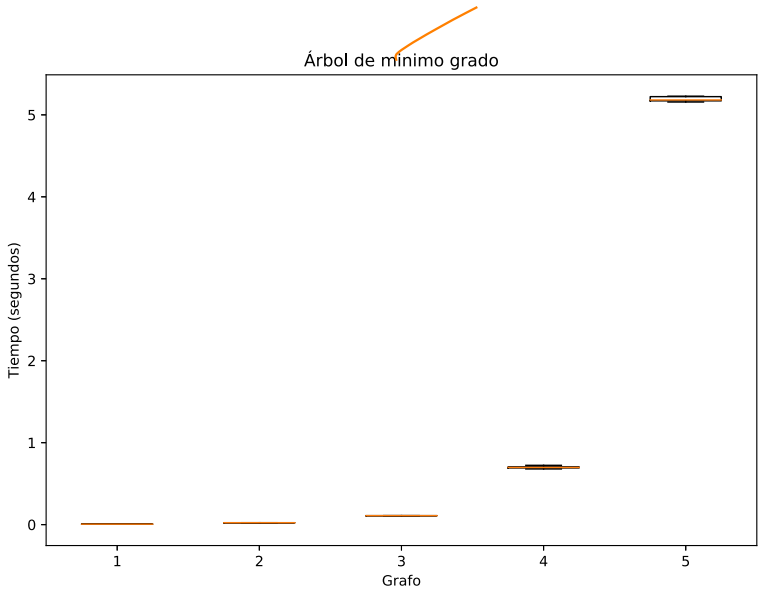


Figura 8: Diagrama caja y bigote para el árbol de mínima grado.

En este algoritmo también se ve una clara diferencia en el tiempo de ejecución mientras más nodos tiene el grafo.

2.5. Árbol de mínima expansión

A continuación se muestra el histograma en la figura 9 y el diagrama caja y bigote en la figura 10 con los resultados obtenidos:

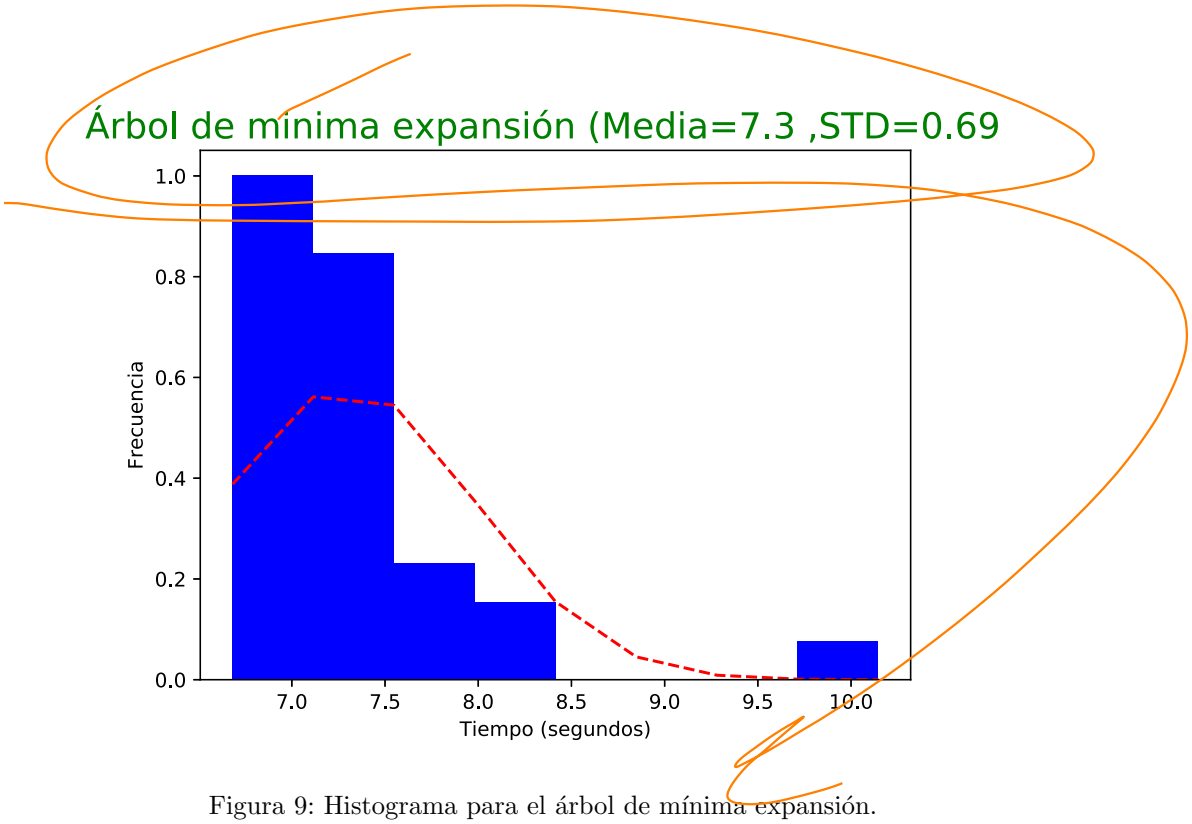


Figura 9: Histograma para el árbol de mínima expansión.

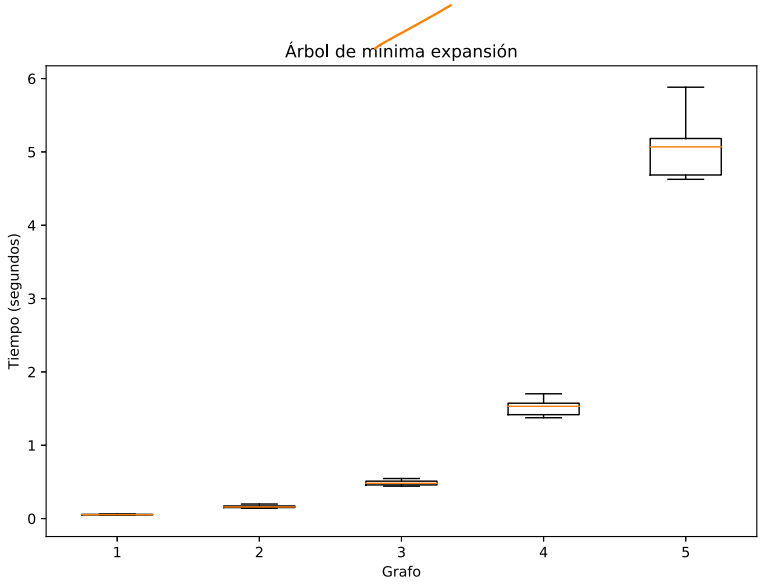


Figura 10: Diagrama caja y bigote para el árbol de mínima expansión.

En el algoritmo también hay una clara diferencia en el tiempo de ejecución mientras más nodos y aristas tiene el grafo.

2.6. Comparación entre algoritmos

La gráfica de dispersión siguiente compara los resultados obtenidos entre los 5 algoritmos, en base al tiempo de ejecución vs el número de nodos (figura 11) y el tiempo de ejecución vs el número de aristas (figura 12). Para diferenciar entre cada algoritmo, utilizamos:

- Ruta más corta: Círculo azul.
- Árbol búsqueda profunda: Cuadrado amarillo.
- Problema de liga de amigos: Estrella verde.
- Árbol de mínimo grado: Pentágono rojo.
- Árbol de mínima expansión: Diamante naranja.

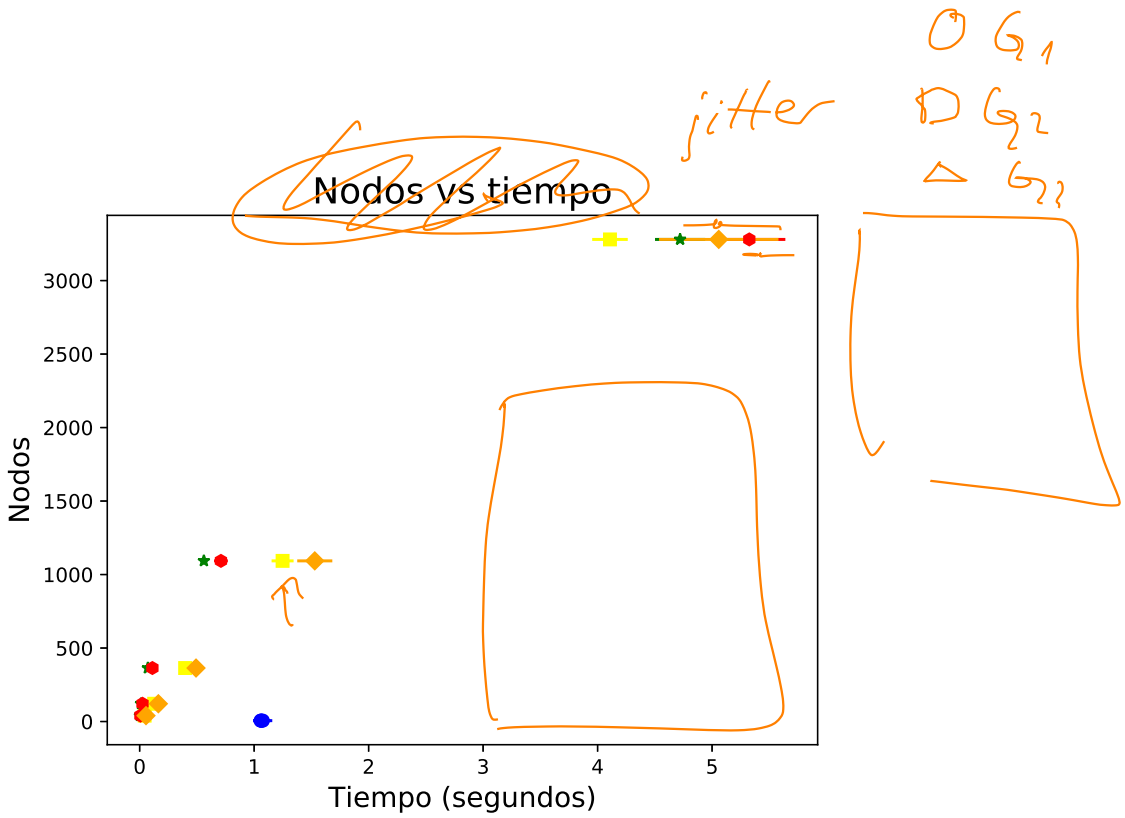


Figura 11: Gráfica de dispersión: tiempo de ejecución vs número de nodos.

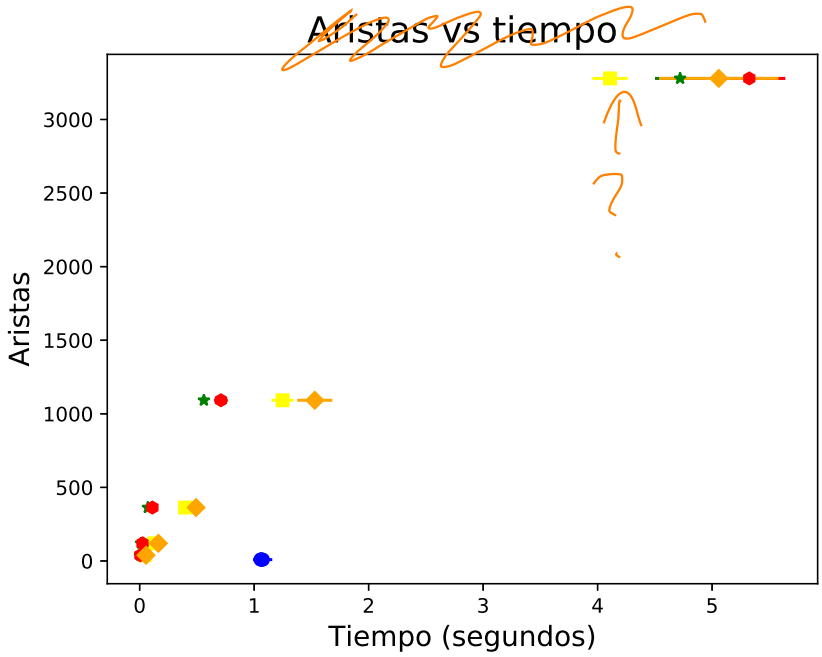


Figura 12: Gráfica de dispersión: tiempo de ejecución vs número de aristas.

3. Conclusiones

Como conclusión de esta investigación, se tiene que el tiempo de ejecución de cualquiera de los algoritmos estudiados es proporcional al número de nodos y de aristas que se tiene en el grafo, por lo que un grafo con un número mayor de nodos incrementa el tiempo de ejecución de los algoritmos.

Referencias

[1] Python Software Foundation Versión 3.7.2. https://www.python.org/.

[2] NetworkX developers con última actualización el 19 de Septiembre 2018. https://networkx.github.io/documentation/stable/index.html.

[3] NetworkX developers Versión 2.0. https://networkx.github.io/.

[4] The Matplotlib development team Versión 3.0.2. https://matplotlib.org/.

[5] Patlán Castillo Jesús Angel. Repositorio Optimización Flujo en Redes. https://github.com/JAPatlanC/Flujo-Redes.

5. Tarea 4

Para esta tarea:

- Se realizaron pequeñas correcciones al documento de diseño y errores ortográficos.
- Se arreglo el código presente en el documento.
- Se ajustaron los valores P mostrados en la tabla ANOVA, mostrando solo hasta 2 decimales.
- Se agregaron gráficas que apoyan los resultados y las conclusiones.

Tarea 4

Jesus Angel Patlán Castillo (5261)

2 de abril de 2019

En esta tarea se analizan la complejidad que se tiene en la utilización de métodos algorítmicos de flujo máximo en distintos tipos de grafos, los cuales son generados por algoritmos de generación de grafos proporcionados por librerías de Python. Los algoritmos se obtuvieron por medio de la librería NetworkX [3] de Python [1], la librería Matplotlib [4] es utilizada para generar las gráfica de correlación entre los efectos estudiados. El código empleado se obtuvo consultando la documentación oficial de la librería NetworkX [2]. Las imágenes y el código se encuentran disponibles directamente en mi repositorio [5].

1. Metodología

La librería NetworkX tiene a la mano diversos algoritmos para resolver distintos algoritmos para poder resolver problemas de flujo máximo. Particularmente para esta investigación, se realiza el análisis de los algoritmos:

- Flujo Máximo: El algoritmo de flujo máximo nos permite encontrar el flujo máximo que se puede dar entre dos nodos del grafo (llamados fuente y sumidero). En la librería de NetworkX está disponible por medio de la función `maximum flow(G,N1,N2)`, donde el parámetro G es el grafo, N1 el nodo fuente y N2 el nodo sumidero:

```
nx.maximum_flow(G1,G1S,G1T)
```

- Valor de flujo máximo: El algoritmo de valor de flujo máximo se utiliza para encontrar el valor máximo de flujo entre un par de nodos. En la librería de NetworkX esta disponible por medio de la función `maximum flow value(G,N1,N2)`, donde el parámetro G es el grafo, $N1$ es el nodo fuente y $N2$ es el nodo sumidero:

```
nx.maximum_flow_value(G1, G1S, G1T)
```

- Valor de corte mínimo: El algoritmo valor de corte mínimo permite encontrar el valor mínimo entre el corte (una partición de nodos del grafo) que permita minimizar los pesos entre los grafos. En la librería de NetworkX esta disponible por medio de la función `minimum cut value(G,N1,N2)`, donde el parámetro G es el grafo, $N1$ el nodo fuente y $N2$ es el nodo sumidero:

```
nx.minimum_cut_value(G1, G1S, G1T)
```

Cada uno de estos algoritmos fueron ejecutados sobre diez grafos diferentes generados por tres métodos de generación de grafos distintos, los cuales son:

- Grafo paleta: Un grafo paleta es un grafo que consiste en la unión entre un grafo completo y un "puente"sobre uno de los nodos del grafo. En la librería de NetworkX está disponible por medio de la función `lollipop graph(N1,N2)`, donde el parámetro $N1$ es la dimensión del grafo completo y $N2$ es la dimensión del "puente":

```
G1 = nx.lollipop_graph(log, 2)
```

- Grafo Turan: El grafo turan es un grafo completo multipartito de que contiene n nodos con r subconjuntos disjuntos. En la librería de NetworkX está disponible por medio de la función `turan graph(n,r)`, donde el parámetro n es la cantidad de nodos del grafo completo y r es la cantidad de subconjuntos disjuntos:

```
G2 = nx.turan_graph(log, 2)
```

- Grafo escalera: El grafo escalera es un grafo que contiene dos caminos de n nodos, sobre el cual cada par esta conectado por una única arista. En la librería de NetworkX está disponible por medio de la función `ladder graph(n)`, donde el parámetro n es la dimensión del grafo:

```
G3 = nx.ladder_graph(log)
```

Para cada combinación de métodos de flujo máximo y de generación de grafo, se realizaron 10 réplicas de grafos distintos, para cada uno de los cuales se realizaron 5 repeticiones distintas para 5 pares de fuente-sumidero distintos. Además, se consideraron 4 distintos órdenes de tamaño para cada grafo, considerando una escala logarítmica en base 2^n con valores de $n \in \{7, 8, 9, 10\}$. Después de realizar las replicas, se obtuvo el tiempo que tomo en ejecutar cada algoritmo en cada uno de los grafos, y se realizaron las pruebas estadísticas ANOVA para determinar si el tiempo de ejecución es afectado por el orden del grafo, el algoritmo de flujo máximo utilizado, el algoritmo generador del grafo y la densidad de cada grafo; y si existe una correlación entre estos factores. A cada arista de cada grafo se le asigno un peso dado por una distribución normal con media 10 y desviación 2.5. Las siguientes líneas de código representan la generación de la recopilación de datos de las réplicas, la matriz de correlación y la tabla de ANOVA:

```
#Orden logaritmico
for logarithmOrder in range(3,5):
    print('Orden: ',logarithmOrder)
    log = 2**logarithmOrder
    ban=True
    # 3 Metodos de generaci n distintos
```

```
    G1 = nx.lollipop_graph(log, 2)
    for e in G1.edges():
        G1[e[0]][e[1]]['capacity'] = np.random.normal(mu, sigma)
    G2 = nx.turan_graph(log, 2)
    for e in G2.edges():
        G2[e[0]][e[1]]['capacity'] = np.random.normal(mu, sigma)
    G3 = nx.ladder_graph(log)
    for e in G3.edges():
        G3[e[0]][e[1]]['capacity'] = np.random.normal(mu, sigma)
    for newPair in range(5):
        print('Pareja: ',newPair)
        #10 Grafos
        for graphRepetition in range(10):
            if ban:
                ban=False
                while G1T==G1S:
                    G1S = choice(list(G1.nodes()))
                    G1T = choice(list(G1.nodes()))
                while G2T==G2S:
                    G2S = choice(list(G2.nodes()))
                    G2T = choice(list(G2.nodes()))
                while G3T==G3S:
                    G3S = choice(list(G3.nodes()))
                    G3T = choice(list(G3.nodes()))
            # GRAFO 1
            g1_start_time = time.time()
            nx.maximum_flow(G1,G1S,G1T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(0, 0, logarithmOrder, nx.density
    (G1), g1_end_time))

            g1_start_time = time.time()
            nx.maximum_flow_value(G1, G1S, G1T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(0, 1, logarithmOrder, nx.density
    (G1), g1_end_time))

            g1_start_time = time.time()
            nx.minimum_cut_value(G1, G1S, G1T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(0, 2, logarithmOrder, nx.density
    (G1), g1_end_time))

            # GRAFO 2
            g1_start_time = time.time()
            nx.maximum_flow(G2, G2S, G2T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(1, 0, logarithmOrder, nx.density
    (G2), g1_end_time))

            g1_start_time = time.time()
            nx.maximum_flow_value(G2, G2S, G2T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(1, 1, logarithmOrder, nx.density
    (G2), g1_end_time))

            g1_start_time = time.time()
```

```
            nx.minimum_cut_value(G2, G2S, G2T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(1, 2, logarithmOrder, nx.density(G2), g1_end_time))

            # GRAFO 3
            g1_start_time = time.time()
            nx.maximum_flow(G3, G3S, G3T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(2, 0, logarithmOrder, nx.density(G3), g1_end_time))

            g1_start_time = time.time()
            nx.maximum_flow_value(G3, G3S, G3T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(2, 1, logarithmOrder, nx.density(G3), g1_end_time))

            g1_start_time = time.time()
            nx.minimum_cut_value(G3, G3S, G3T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(2, 2, logarithmOrder, nx.density(G3), g1_end_time))
print(totalTests.__sizeof__())

data = pd.DataFrame.from_records([s.to_dict() for s in totalTests])

#ANOVA
formula1 = 'time ~ C(generation)'
formula2 = 'time ~ C(algorithm)'
formula3 = 'time ~ C(nodes)'
formula4 = 'time ~ C(density)'
formula5 = 'time ~ C(generation)*C(algorithm)'
formula6 = 'time ~ C(generation)*C(nodes)'
formula7 = 'time ~ C(generation)*C(density)'
formula8 = 'time ~ C(algorithm)*C(nodes)'
formula9 = 'time ~ C(algorithm)*C(density)'
formula10 = 'time ~ C(nodes)*C(density)'
formula11 = 'time ~ C(generation)*C(algorithm)*C(nodes)'
formula12 = 'time ~ C(generation)*C(algorithm)*C(density)'
formula13 = 'time ~ C(algorithm)*C(nodes)*C(density)'
formula14 = 'time ~ C(generation)*C(algorithm)*C(nodes)*C(density)'
model1 = ols(formula1, data).fit()
model2 = ols(formula2, data).fit()
model3 = ols(formula3, data).fit()
model4 = ols(formula4, data).fit()
model5 = ols(formula5, data).fit()
model6 = ols(formula6, data).fit()
model7 = ols(formula7, data).fit()
model8 = ols(formula8, data).fit()
model9 = ols(formula9, data).fit()
model10 = ols(formula10, data).fit()
model11 = ols(formula11, data).fit()
model12 = ols(formula12, data).fit()
model13 = ols(formula13, data).fit()
model14 = ols(formula14, data).fit()
anova_table1 = anova_lm(model1, typ=2)
```

```
anova_table2 = anova_lm(model2, typ=2)
anova_table3 = anova_lm(model3, typ=2)
anova_table4 = anova_lm(model4, typ=2)
anova_table5 = anova_lm(model5, typ=2)
anova_table6 = anova_lm(model6, typ=2)
anova_table7 = anova_lm(model7, typ=2)
anova_table8 = anova_lm(model8, typ=2)
anova_table9 = anova_lm(model9, typ=2)
anova_table10 = anova_lm(model10, typ=2)
anova_table11 = anova_lm(model11, typ=2)
anova_table12 = anova_lm(model12, typ=2)
anova_table13 = anova_lm(model13, typ=2)
anova_table14 = anova_lm(model14, typ=2)
#anova_table1.to_csv('anova1.xls',sep='\t')
#anova_table2.to_csv('anova2.xls',sep='\t')
#anova_table3.to_csv('anova3.xls',sep='\t')
#anova_table4.to_csv('anova4.xls',sep='\t')
#anova_table5.to_csv('anova5.xls',sep='\t')
#anova_table6.to_csv('anova6.xls',sep='\t')
#anova_table7.to_csv('anova7.xls',sep='\t')
#anova_table8.to_csv('anova8.xls',sep='\t')
#anova_table9.to_csv('anova9.xls',sep='\t')
#anova_table10.to_csv('anova10.xls',sep='\t')
#anova_table11.to_csv('anova11.xls',sep='\t')
#anova_table12.to_csv('anova12.xls',sep='\t')
#anova_table13.to_csv('anova13.xls',sep='\t')
#anova_table14.to_csv('anova14.xls',sep='\t')
#data.to_csv('data.xls', sep='\t')

#MATRIZ CORRELACION
data = data.iloc[:, :-1]
data.columns = ['algoritmo','densidad','generacion',
                'orden']
corr = data.corr()
print(corr.columns)
fig, ax = plt.subplots(figsize=(6, 6))
ax.matshow(corr)
plt.xticks(range(len(corr.columns)), corr.columns)
plt.yticks(range(len(corr.columns)), corr.columns)
cax = ax.matshow(corr,cmap='OrRd', vmin=-1, vmax=1, aspect='equal',
    origin='lower')
fig.colorbar(cax)
plt.show()
```

Este proceso se realizó en una laptop con las siguientes características:

- Procesador: Intel Core i7-7500U 2.7GHz
- Memoria RAM: 16GB
- Sistema Operativo: Windows 10 64 bits

2. Resultados

Se obtuvieron los siguientes resultados de las replicas, una matriz de correlación en la figura 1 y las pruebas estadísticas de ANOVA para determinar los

efectos de los factores:

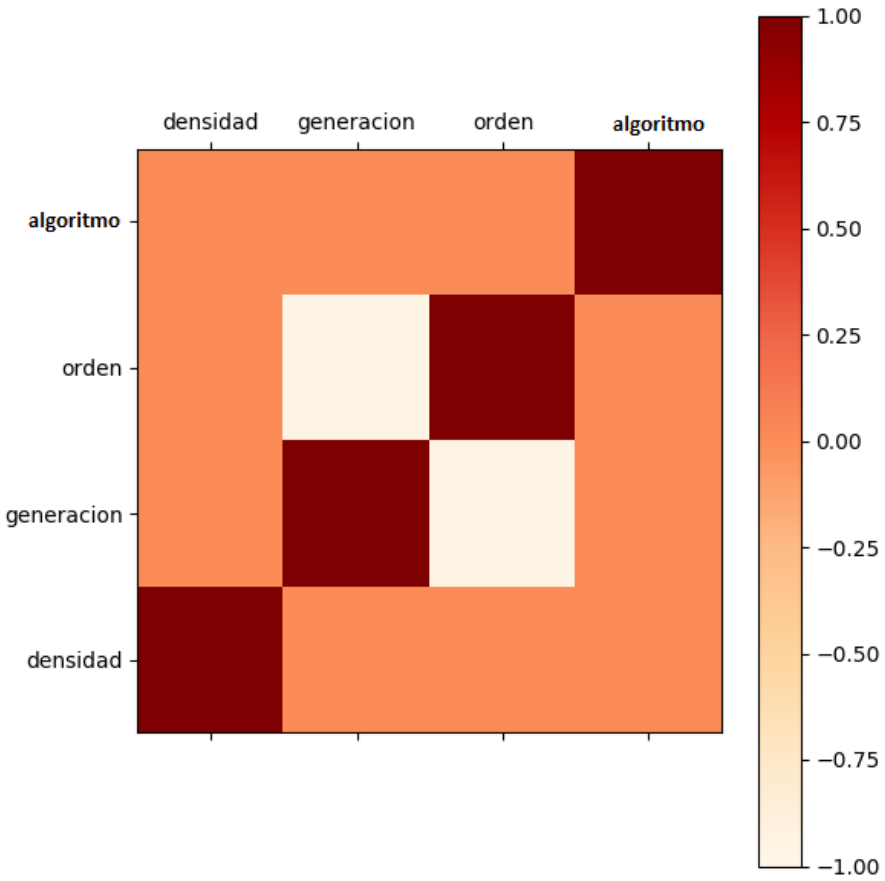


Figura 1: Matriz de correlación.

ANOVA	Grados de Libertad	Valor P
C(generation)	2	6.06E-102
C(algoritmo)	2	0.15[illegible]27
C(orden)	3	1.41E-221
C(densidad)	11	0
C(generation):C(algoritmo)	4	0.49[illegible]76
C(generation):C(orden)	6	0
C(algoritmo):C(orden)	6	0.3[illegible]38
C(generation):C(densidad)	22	0.99[illegible]62
C(algoritmo):C(densidad)	22	2.37E-77
C(orden):C(densidad)	33	0.999058669
C(generation):C(algoritmo):C(orden)	12	7.24E-27
C(generation):C(algoritmo):C(densidad)	44	0.997362183
C(generation):C(orden):C(densidad)	66	7.24E-27
C(algoritmo):C(orden):C(densidad)	66	0.97[illegible]8
C(generation):C(algoritmo):C(orden):C(densidad)	132	0.[illegible]748
Residual	1764	

3. Conclusiones

Como conclusión de esta investigación, se tiene que cada uno de los factores de algoritmo, orden, generación de grafo y densidad afectan directamente con el tiempo de ejecución de los algoritmos, además que, en combinación, todos excepto la combinación de orden-generación afectan al rendimiento del algoritmo. La tabla ANOVA respalda los resultados con los valores de P dados, y además muestran que existe una interacción entre los factores de generación-orden-densidad y generación-algoritmo-orden.

Referencias

[1] Python Software Foundation Versión 3.7.2. https://www.python.org/.

[2] NetworkX developers con última actualización el 19 de Septiembre 2018. https://networkx.github.io/documentation/stable/index.html.

[3] NetworkX developers Versión 2.0. https://networkx.github.io/.

[4] The Matplotlib development team Versión 3.0.2. https://matplotlib.org/.

[5] Patlán Castillo Jesús Angel. Repositorio Optimización Flujo en Redes. https://github.com/JAPatlanC/Flujo-Redes.

Tarea 4

Jesus Angel Patlán Castillo (5261)

2 de abril de 2019

En esta tarea se analizan la complejidad que se tiene en la utilización de métodos algorítmicos de flujo máximo en distintos tipos de grafos, los cuales son generados por algoritmos de generación de grafos proporcionados por librerías de Python. Los algoritmos se obtuvieron por medio de la librería NetworkX [3] de Python [1], la librería Matplotlib [4] es utilizada para generar las gráfica de correlación entre los efectos estudiados. El código empleado se obtuvo consultando la documentación oficial de la librería NetworkX [2]. Las imágenes y el código se encuentran disponibles directamente en mi repositorio [5].

1. Metodología

La librería NetworkX tiene a la mano diversos algoritmos para resolver distintos algoritmos para poder resolver problemas de flujo máximo. Particularmente para esta investigación, se realiza el análisis de los algoritmos:

- Flujo Máximo: El algoritmo de flujo máximo nos permite encontrar el flujo máximo que se puede dar entre dos nodos del grafo (llamados fuente y sumidero). En la librería de NetworkX está disponible por medio de la función `maximum flow(G,N1,N2)`, donde el parámetro G es el grafo, N1 el nodo fuente y N2 el nodo sumidero:

```
nx.maximum_flow(G1,G1S,G1T)
```

- Valor de flujo máximo: El algoritmo de valor de flujo máximo se utiliza para encontrar el valor máximo de flujo entre un par de nodos. En la librería de NetworkX está disponible por medio de la función `maximum flow value(G,N1,N2)`, donde el parámetro G es el grafo, $N1$ es el nodo fuente y $N2$ es el nodo sumidero:

```
nx.maximum_flow_value(G1, G1S, G1T)
```

- Valor de corte mínimo: El algoritmo valor de corte mínimo permite encontrar el valor mínimo entre el corte (una partición de nodos del grafo) que permita minimizar los pesos entre los grafos. En la librería de NetworkX está disponible por medio de la función `minimum cut value(G,N1,N2)`, donde el parámetro G es el grafo, $N1$ es el nodo fuente y $N2$ es el nodo sumidero:

```
nx.minimum_cut_value(G1, G1S, G1T)
```

Cada uno de estos algoritmos fueron ejecutados sobre diez grafos diferentes generados por tres métodos de generación de grafos distintos, los cuales son:

- Grafo paleta: Un grafo paleta es un grafo que consiste en la unión entre un grafo completo y un "puente"sobre uno de los nodos del grafo. En la librería de NetworkX está disponible por medio de la función `lollipop graph(N1,N2)`, donde el parámetro $N1$ es la dimensión del grafo completo y $N2$ es la dimensión del "puente":

```
G1 = nx.lollipop_graph(log, 2)
```

- Grafo Turan: El grafo turan es un grafo completo multipartito de que contiene n nodos con r subconjuntos disjuntos. En la librería de NetworkX está disponible por medio de la función `turan graph(n,r)`, donde el parámetro n es la cantidad de nodos del grafo completo y r es la cantidad de subconjuntos disjuntos:

```
G2 = nx.turan_graph(log, 2)
```

- Grafo escalera: El grafo escalera es un grafo que contiene dos caminos de n nodos, sobre el cual cada par esta conectado por una única arista. En la librería de NetworkX está disponible por medio de la función `ladder graph(n)`, donde el parámetro n es la dimensión del grafo:

```
G3 = nx.ladder_graph(log)
```

Para cada combinación de métodos de flujo máximo y de generación de grafo, se realizaron 10 réplicas de grafos distintos, para cada uno de los cuales se realizaron 5 repeticiones distintas para 5 pares de fuente-sumidero distintos. Además, se consideraron 4 distintos órdenes de tamaño para cada grafo, considerando una escala logarítmica en base 2^n con valores de $n \in \{7, 8, 9, 10\}$. Después de realizar las replicas, se obtuvo el tiempo que tomo en ejecutar cada algoritmo en cada uno de los grafos, y se realizaron las pruebas estadísticas ANOVA para determinar si el tiempo de ejecución es afectado por el orden del grafo, el algoritmo de flujo máximo utilizado, el algoritmo generador del grafo y la densidad de cada grafo; y si existe una correlación entre estos factores. A cada arista de cada grafo se le asignó un peso dado por una distribución normal con media 10 y desviación 2.5. Las siguientes líneas de código representan la generación de la recopilación de datos de las réplicas, la matriz de correlación y la tabla de ANOVA:

```
#Orden logaritmico
for logarithmOrder in range(3,5):
    print('Orden: ',logarithmOrder)
    log = 2**logarithmOrder
    ban=True
    # 3 Metodos de generacion distintos
```

```
    G1 = nx.lollipop_graph(log, 2)
    for e in G1.edges():
        G1[e[0]][e[1]]['capacity'] = np.random.normal(mu, sigma)
    G2 = nx.turan_graph(log, 2)
    for e in G2.edges():
        G2[e[0]][e[1]]['capacity'] = np.random.normal(mu, sigma)
    G3 = nx.ladder_graph(log)
    for e in G3.edges():
        G3[e[0]][e[1]]['capacity'] = np.random.normal(mu, sigma)
    for newPair in range(5):
        print('Pareja: ',newPair)
        #10 Grafos
        for graphRepetition in range(10):
            if ban:
                ban=False
                while G1T==G1S:
                    G1S = choice(list(G1.nodes()))
                    G1T = choice(list(G1.nodes()))
                while G2T==G2S:
                    G2S = choice(list(G2.nodes()))
                    G2T = choice(list(G2.nodes()))
                while G3T==G3S:
                    G3S = choice(list(G3.nodes()))
                    G3T = choice(list(G3.nodes()))
            # GRAFO 1
            g1_start_time = time.time()
            nx.maximum_flow(G1,G1S,G1T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(0, 0, logarithmOrder, nx.density
    (G1), g1_end_time))

            g1_start_time = time.time()
            nx.maximum_flow_value(G1, G1S, G1T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(0, 1, logarithmOrder, nx.density
    (G1), g1_end_time))

            g1_start_time = time.time()
            nx.minimum_cut_value(G1, G1S, G1T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(0, 2, logarithmOrder, nx.density
    (G1), g1_end_time))

            # GRAFO 2
            g1_start_time = time.time()
            nx.maximum_flow(G2, G2S, G2T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(1, 0, logarithmOrder, nx.density
    (G2), g1_end_time))

            g1_start_time = time.time()
            nx.maximum_flow_value(G2, G2S, G2T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(1, 1, logarithmOrder, nx.density
    (G2), g1_end_time))

            g1_start_time = time.time()
```

```
            nx.minimum_cut_value(G2, G2S, G2T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(1, 2, logarithmOrder, nx.density(G2), g1_end_time))

            # GRAFO 3
            g1_start_time = time.time()
            nx.maximum_flow(G3, G3S, G3T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(2, 0, logarithmOrder, nx.density(G3), g1_end_time))

            g1_start_time = time.time()
            nx.maximum_flow_value(G3, G3S, G3T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(2, 1, logarithmOrder, nx.density(G3), g1_end_time))

            g1_start_time = time.time()
            nx.minimum_cut_value(G3, G3S, G3T)
            g1_end_time = time.time() - g1_start_time
            totalTests.append(Test(2, 2, logarithmOrder, nx.density(G3), g1_end_time))
print(totalTests.__sizeof__())

data = pd.DataFrame.from_records([s.to_dict() for s in totalTests])

#ANOVA
formula1 = 'time ~ C(generation)'
formula2 = 'time ~ C(algorithm)'
formula3 = 'time ~ C(nodes)'
formula4 = 'time ~ C(density)'
formula5 = 'time ~ C(generation)*C(algorithm)'
formula6 = 'time ~ C(generation)*C(nodes)'
formula7 = 'time ~ C(generation)*C(density)'
formula8 = 'time ~ C(algorithm)*C(nodes)'
formula9 = 'time ~ C(algorithm)*C(density)'
formula10 = 'time ~ C(nodes)*C(density)'
formula11 = 'time ~ C(generation)*C(algorithm)*C(nodes)'
formula12 = 'time ~ C(generation)*C(algorithm)*C(density)'
formula13 = 'time ~ C(algorithm)*C(nodes)*C(density)'
formula14 = 'time ~ C(generation)*C(algorithm)*C(nodes)*C(density)'
model1 = ols(formula1, data).fit()
model2 = ols(formula2, data).fit()
model3 = ols(formula3, data).fit()
model4 = ols(formula4, data).fit()
model5 = ols(formula5, data).fit()
model6 = ols(formula6, data).fit()
model7 = ols(formula7, data).fit()
model8 = ols(formula8, data).fit()
model9 = ols(formula9, data).fit()
model10 = ols(formula10, data).fit()
model11 = ols(formula11, data).fit()
model12 = ols(formula12, data).fit()
model13 = ols(formula13, data).fit()
model14 = ols(formula14, data).fit()
anova_table1 = anova_lm(model1, typ=2)
```

```
anova_table2 = anova_lm(model2, typ=2)
anova_table3 = anova_lm(model3, typ=2)
anova_table4 = anova_lm(model4, typ=2)
anova_table5 = anova_lm(model5, typ=2)
anova_table6 = anova_lm(model6, typ=2)
anova_table7 = anova_lm(model7, typ=2)
anova_table8 = anova_lm(model8, typ=2)
anova_table9 = anova_lm(model9, typ=2)
anova_table10 = anova_lm(model10, typ=2)
anova_table11 = anova_lm(model11, typ=2)
anova_table12 = anova_lm(model12, typ=2)
anova_table13 = anova_lm(model13, typ=2)
anova_table14 = anova_lm(model14, typ=2)

#MATRIZ CORRELACION
data = data.iloc[:, :-1]
data.columns = ['algoritmo','densidad','generacion',
                'orden']
corr = data.corr()
print(corr.columns)
fig, ax = plt.subplots(figsize=(6, 6))
ax.matshow(corr)
plt.xticks(range(len(corr.columns)), corr.columns)
plt.yticks(range(len(corr.columns)), corr.columns)
cax = ax.matshow(corr,cmap='OrRd', vmin=-1, vmax=1, aspect='equal',
    origin='lower')
fig.colorbar(cax)
plt.show()
```

Este proceso se realizó en una laptop con las siguientes características:

- Procesador: Intel Core i7-7500U 2.7GHz
- Memoria RAM: 16GB
- Sistema Operativo: Windows 10 64 bits

2. Resultados

Se obtuvieron los siguientes resultados de las réplicas, una matriz de correlación en la figura 4, gráficas de caja y bigote para el rendimiento individual por algoritmo, generación y nodos, y las pruebas estadísticas de ANOVA para determinar los efectos de los factores:

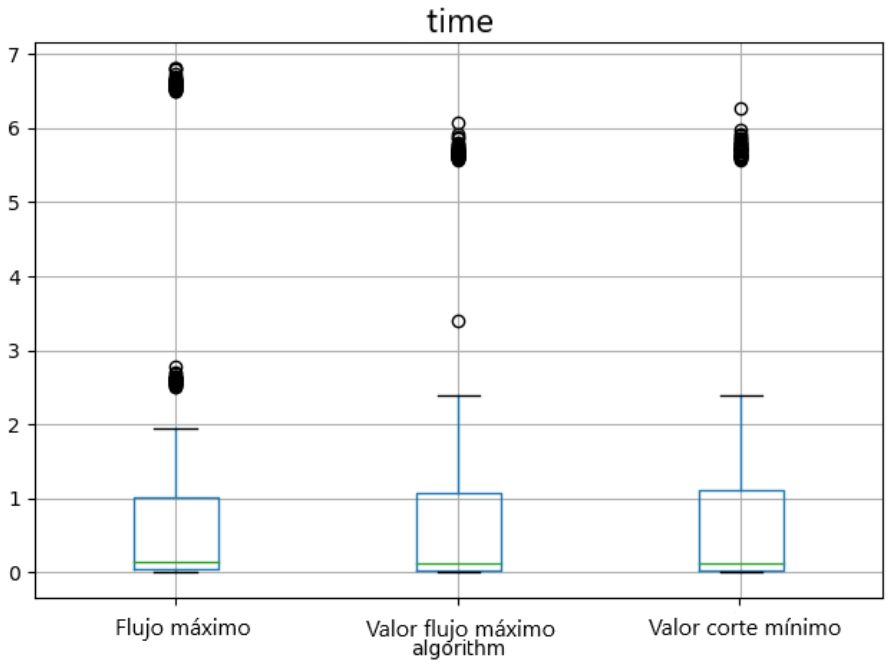


Figura 1: Gráfica caja y bigote para los algoritmos.

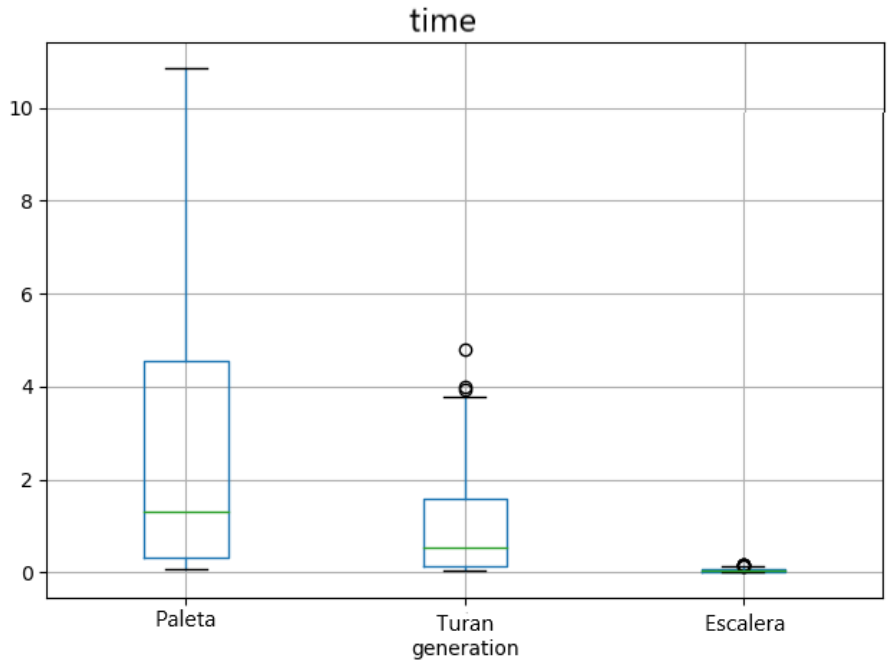


Figura 2: Gráfica caja y bigote para la generación de grafos.

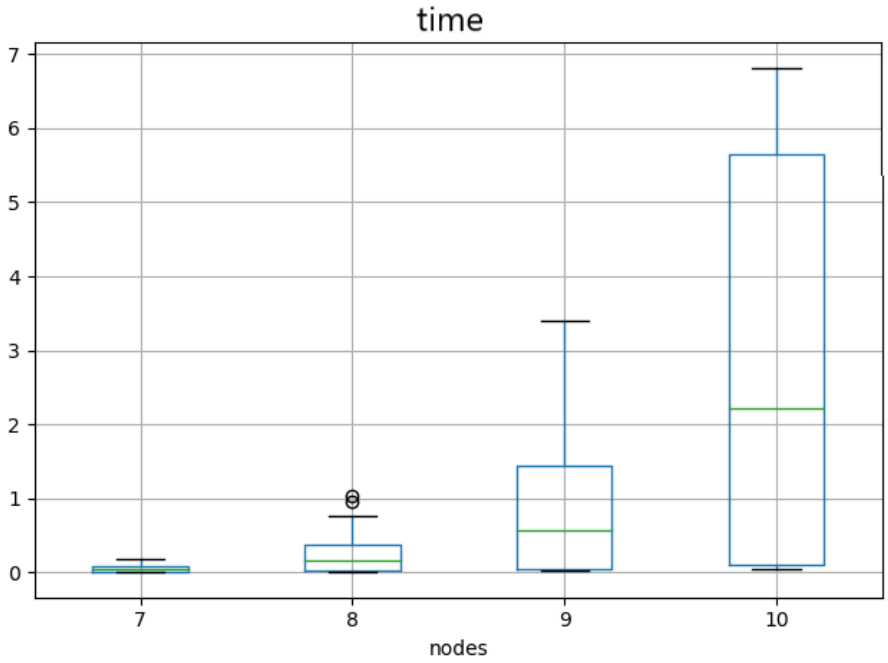


Figura 3: Gráfica caja y bigote para el orden del número de nodos.

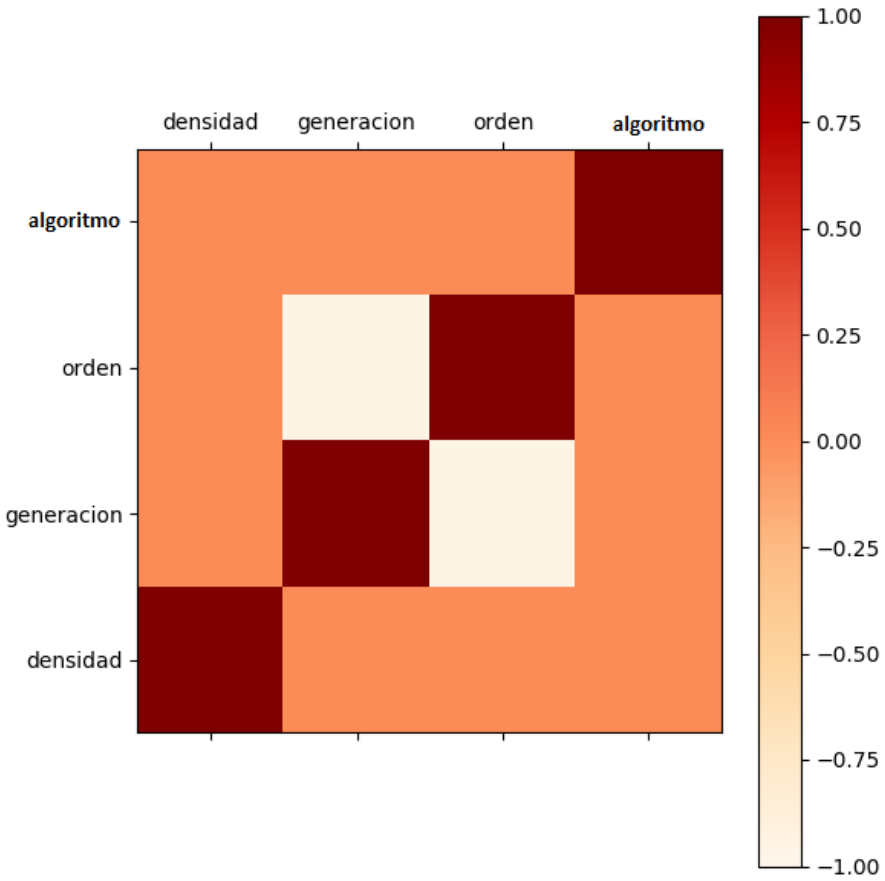


Figura 4: Matriz de correlación.

ANOVA	Grados de Libertad	Valor P
C(generation)	2	0
C(algoritmo)	2	0.16
C(orden)	3	0
C(densidad)	11	0
C(generation):C(algoritmo)	4	0.49
C(generation):C(orden)	6	0
C(algoritmo):C(orden)	6	0.34
C(generation):C(densidad)	22	1
C(algoritmo):C(densidad)	22	0
C(orden):C(densidad)	33	1
C(generation):C(algoritmo):C(orden)	12	0
C(generation):C(algoritmo):C(densidad)	44	1
C(generation):C(orden):C(densidad)	66	0
C(algoritmo):C(orden):C(densidad)	66	0.97
C(generation):C(algoritmo):C(orden):C(densidad)	132	1
Residual	1764	

3. Conclusiones

Como conclusión de esta investigación, se tiene que cada uno de los factores de algoritmo, orden, generación de grafo y densidad afectan directamente con el tiempo de ejecución de los algoritmos, además que, en combinación, todos excepto la combinación de orden-generación afectan al rendimiento del algoritmo. La tabla ANOVA respalda los resultados con los valores de P dados, y además muestran que existe una interacción entre los factores de generación-orden-densidad y generación-algoritmo-orden.

Referencias

[1] Python Software Foundation Versión 3.7.2. https://www.python.org/.

[2] NetworkX developers con última actualización el 19 de Septiembre 2018. https://networkx.github.io/documentation/stable/index.html.

[3] NetworkX developers Versión 2.0. https://networkx.github.io/.

[4] The Matplotlib development team Versión 3.0.2. https://matplotlib.org/.

[5] Patlán Castillo Jesús Angel. Repositorio Optimización Flujo en Redes. https://github.com/JAPatlanC/Flujo-Redes.

6. Tarea 5

Para esta tarea:

- Se realizaron pequeñas correcciones al documento de diseño y errores ortográficos.
- Se arreglo cambio el formato de la tabla a Latex.
- Se agregaron gráficas que apoyan los resultados y las conclusiones.

8.5

Tarea 5

Jesus Angel Patlán Castillo (5261)

30 de abril de 2019

En esta tarea se analizan las características de grafos que son generados por un algoritmo de generación de grafos proporcionado por librerías de Python, además de estudiar como estas características afectan el tiempo de ejecución al momento de obtener el flujo máximo entre dos nodos dentro del grafo. Los algoritmos se obtuvieron por medio de la librería NetworkX [3] de Python [1], la librería Matplotlib [4] es utilizada para generar las gráfica de correlación entre los efectos estudiados. El código empleado se obtuvo consultando la documentación oficial de la librería NetworkX [2]. Las imágenes y el código se encuentran disponibles directamente en mi repositorio [5].

1. Metodología

Basándonos en las tareas anteriores, se decide utilizar de algoritmo de generador de grafos de llamado grafo de cavernícola conexo, el cual tiene una mayor aplicación para el problema de flujo máximo dado que es posible representar cada cluster de nodos como una isla en la que se necesita transportar recursos a otro cluster. Además, tenemos que el algoritmo de flujo máximo que utilizaremos será el tradicional flujo máximo entre dos nodos, dado que fue el algoritmo que mostró un mejor desempeño en tiempo de ejecución. Para una mejor visualización se eligió el algoritmo de acomodo resorte, el cual da una mejor visualización de los nodos y las aristas del grafo. Para las pruebas, se crearon 5 instancias distintas de grafo de cavernícola conexo, los cuales se muestran a continuación:

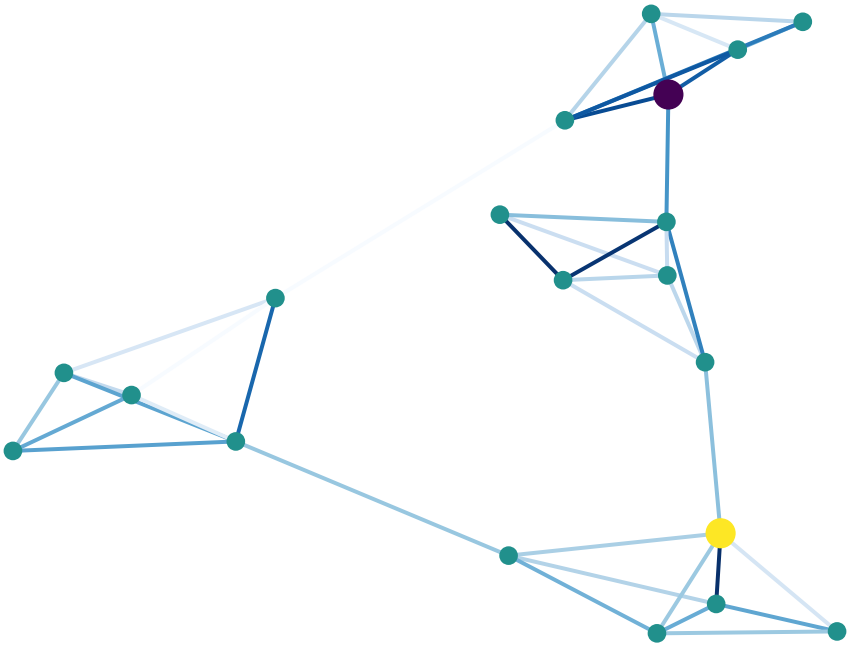


Figura 1: Grafo 1

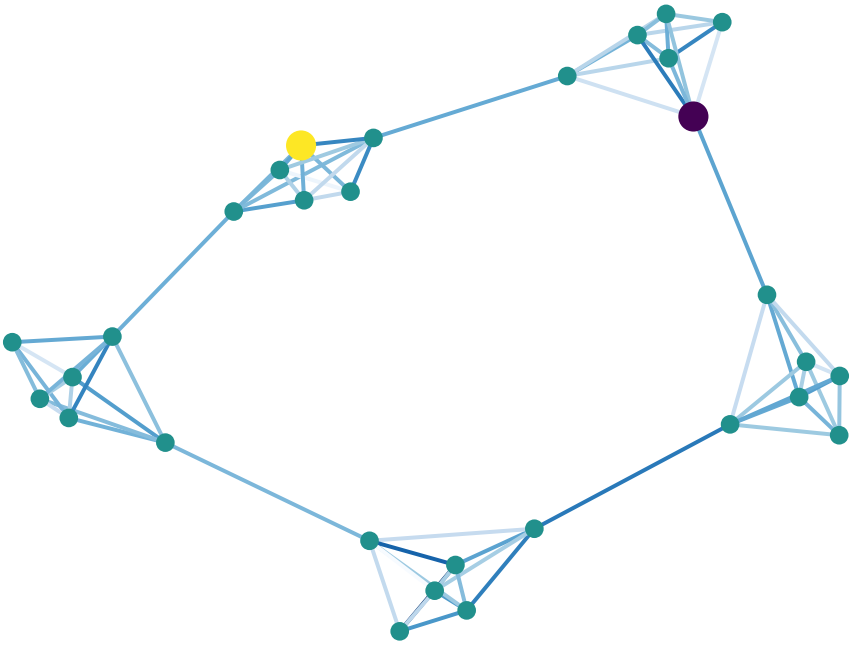


Figura 2: Grafo 2

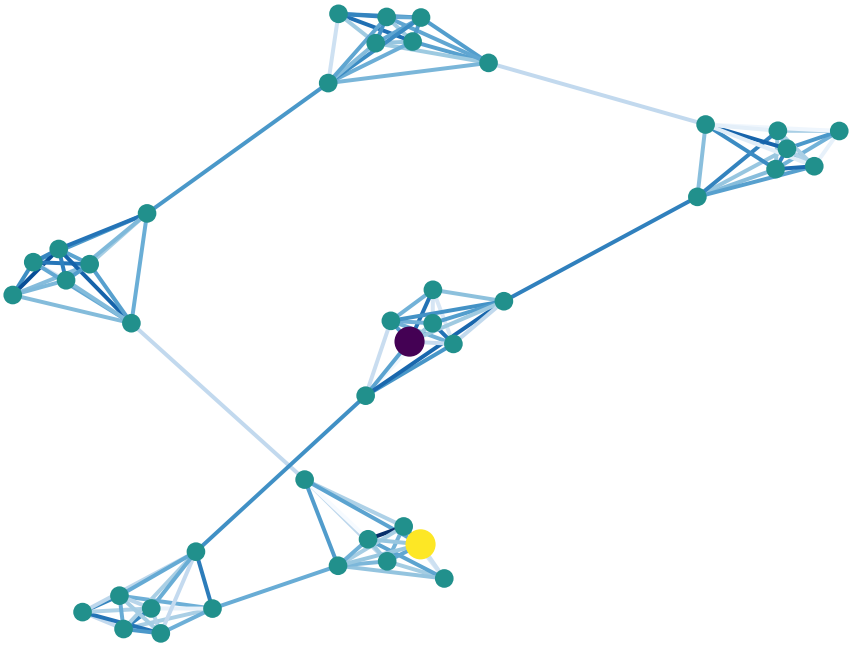


Figura 3: Grafo 3

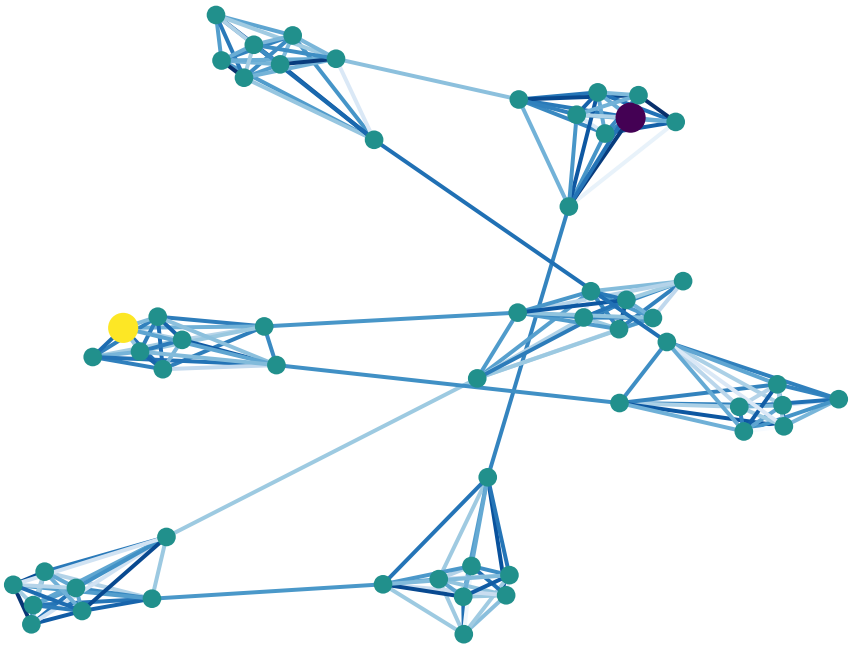


Figura 4: Grafo 4

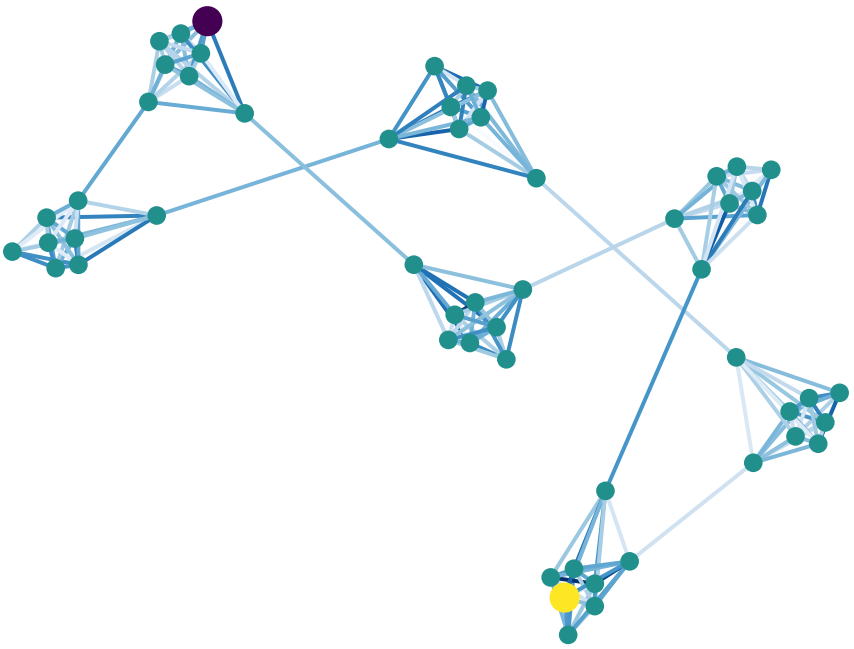


Figura 5: Grafo 5

Para cada grafo, señalamos un nodo fuente (en amarillo) y un nodo sumidero (en negro) de ejemplo para el problema de flujo máximo, además que resaltamos cada arista con un tono de azul más oscuro cuanto mayor sea el peso que tenga. Para cada uno de los grafos, se realizaron distintas pruebas intercambiando los nodos fuente-sumidero para determinar el rendimiento del algoritmo de flujo máximo. Además, a cada grafo se le determinan 5 características:

- Distribución de grado: Es el número de aristas asociadas a un nodo.
- Coeficiente de agrupamiento: Determina que tanto un nodo está asociado a sus nodos vecinos.
- Centralidad de cercanía: Determina qué tan cercano o alejado esta un nodo con respecto a los demás nodos vecinos.
- Centralidad de carga: Determina el número de caminos más cortos que se pasan a través del nodo.
- Excentricidad: Determina la longitud máxima que hay entre el nodo y otro nodo del grafo.
- PageRank: Similar a la distribución de grado, pero da distinto pesos a las aristas de acuerdo al grado que tenga el nodo al que se conecta.

Para la obtención de estas características y la generación de los grafos, se utilizo el siguiente código:

```
def graph_attributes(G):
    global num_fig
    nodes=G.nodes()
    degrees=[G.degree(i) for i in nodes]
    clustering=[nx.clustering(G,i) for i in nodes]
    closeness=[nx.closeness_centrality(G,i) for i in nodes]
    load=[nx.load_centrality(G,i) for i in nodes]
    eccentricity=[nx.eccentricity(G,i) for i in nodes]
    pageRank=nx.pagerank(G,weight="capacity")
    pageRank_G=[pageRank[i] for i in nodes]
    plt.figure(0)
    plt.ylabel('Replicas')
    plt.xlabel('Grado del nodo')
    plt.hist(degrees)
    plt.figure(1)
    plt.savefig('hist-grado-'+str(num_fig)+'.eps', format='eps',
    dpi=1000)
    #plt.clf()
    plt.ylabel('Replicas')
    plt.xlabel('Coeficiente de agrupamiento del nodo')
    plt.hist(clustering)
    plt.figure(2)
    plt.savefig('hist-agrupamiento-' + str(num_fig) + '.eps',
    format='eps', dpi=1000)
    plt.clf()
    plt.ylabel('Replicas')
    plt.xlabel('Coeficiente de cercania del nodo')
    plt.hist(closeness)
    plt.figure(3)
    plt.savefig('hist-cercania-' + str(num_fig) + '.eps', format='
    eps', dpi=1000)
    plt.clf()
    plt.ylabel('Replicas')
    plt.xlabel('Centralidad del nodo')
    plt.hist(load)
    plt.figure(4)
    plt.savefig('hist-centralidad-' + str(num_fig) + '.eps', format
    ='eps', dpi=1000)
    plt.clf()
    plt.ylabel('Replicas')
    plt.xlabel('Excentricidad del nodo')
    plt.hist(eccentricity)
    plt.figure(5)
    plt.savefig('hist-excentricidad-' + str(num_fig) + '.eps',
    format='eps', dpi=1000)
    plt.clf()
    plt.ylabel('Replicas')
    plt.xlabel('PageRank del nodo')
    plt.hist(pageRank_G)
    plt.figure(6)
    plt.savefig('hist-pagerank-' + str(num_fig) + '.eps', format='
    eps', dpi=1000)
    plt.clf()
```

En base a estas 5 características, se realiza una prueba ANOVA para determi-

nar como influyen estos en el tiempo de ejecución del algoritmo. Un ejemplo aplicando el flujo máximo en cada grafo se muestra a continuación, donde se muestran solamente las aristas por donde el flujo se distribuye:

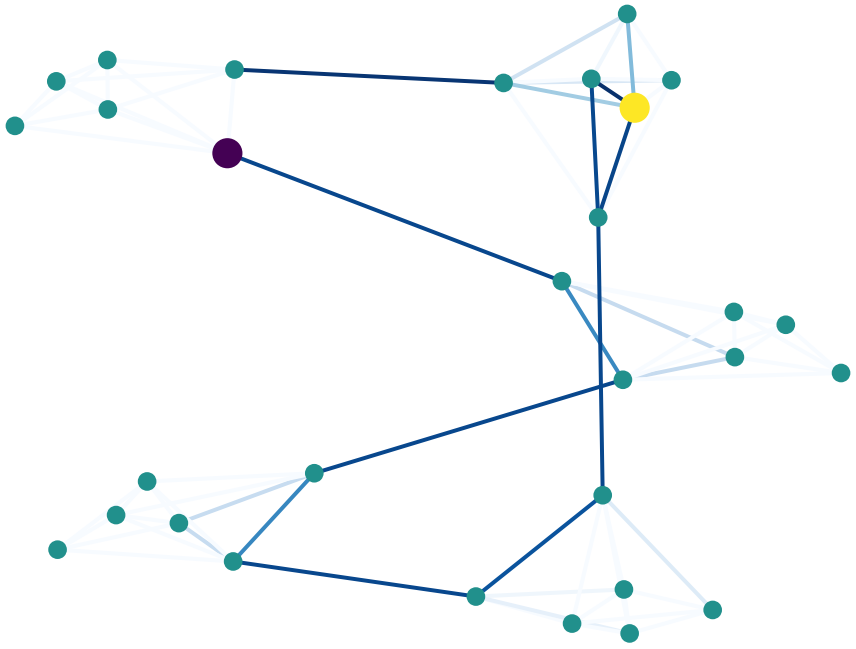


Figura 6: Grafo 1

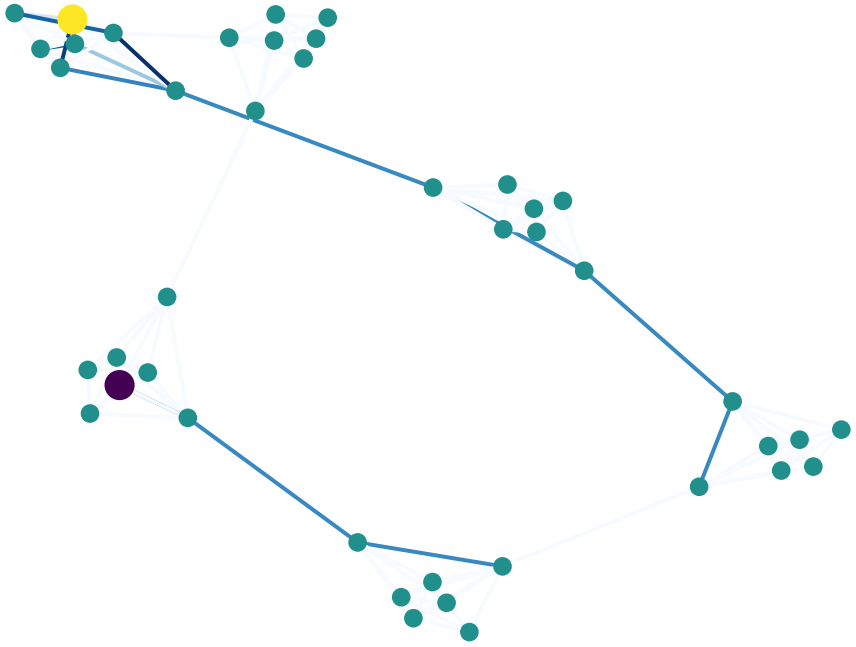


Figura 7: Grafo 2

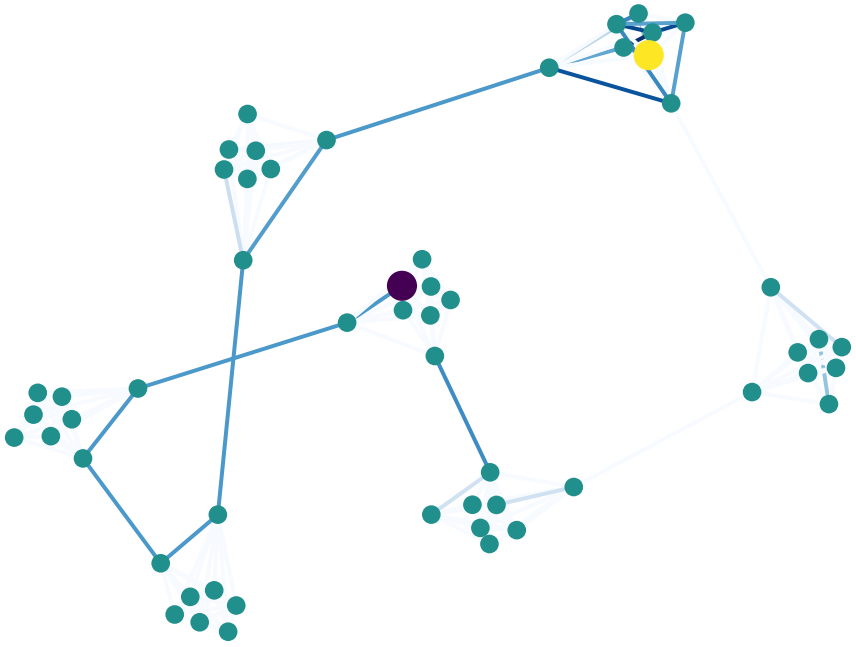


Figura 8: Grafo 3

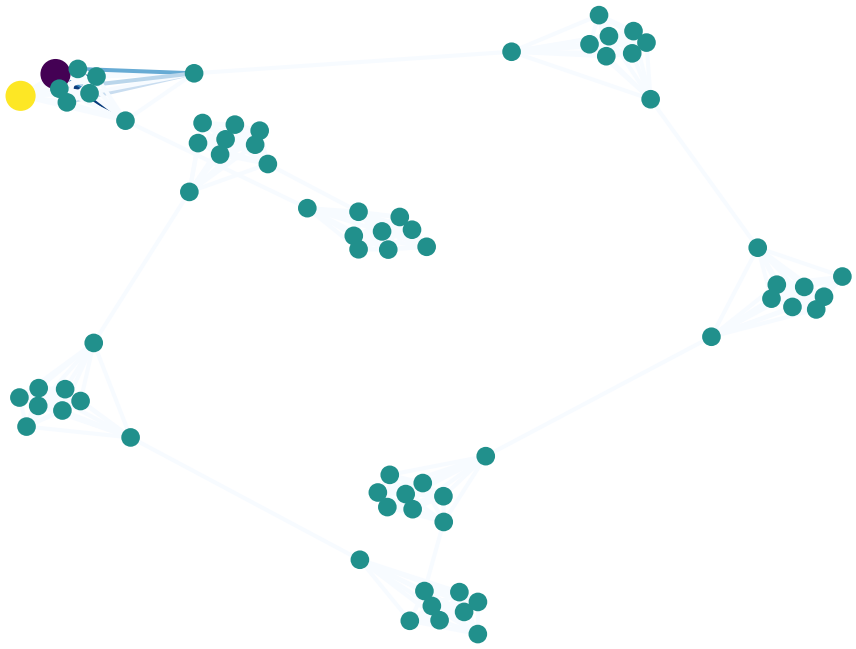


Figura 9: Grafo 4

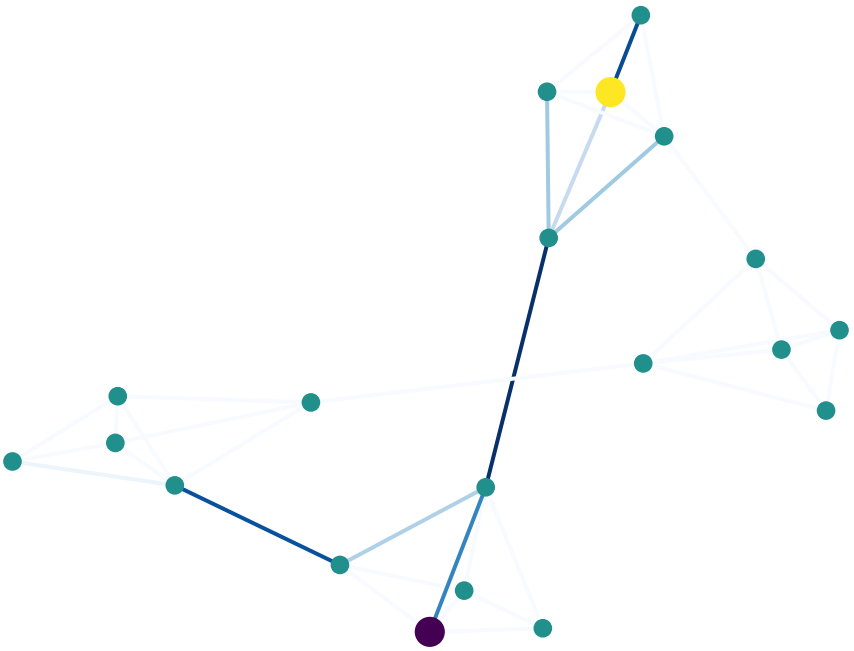


Figura 10: Grafo 5

Este proceso se realizó en una laptop con las siguientes características:

- Procesador: Intel Core i7-7500U 2.7GHz
- Memoria RAM: 16GB
- Sistema Operativo: Windows 10 64 bits

Los resultados fueron determinados por medio de un ANOVA utilizando el siguiente código:

```
dataframe = pd.DataFrame(total_data)
dataframe.to_csv('datos.xls',sep='\t')
model = ols('Tiempo~G*A*C*Ce*E*P',data=dataframe).fit()
anova = anova_lm(model,typ=2)
anova.to_csv('anova1.xls',sep='\t')
```

2. Resultados

Estas son las características encontradas de las instancias de grafos:

2.1. Grado de nodos

A continuación se muestran histogramas relacionadas a los grados de los nodos de cada grafo.

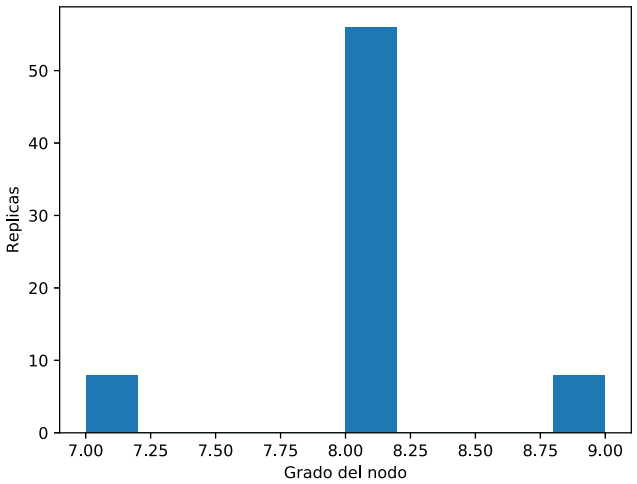


Figura 11: Grafo 1

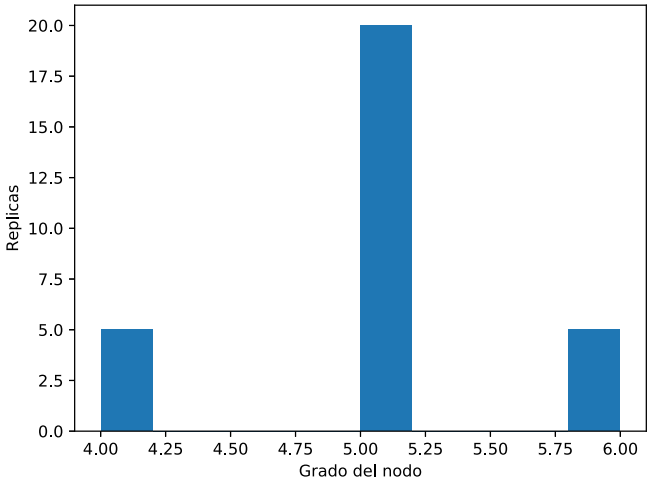


Figura 12: Grafo 2

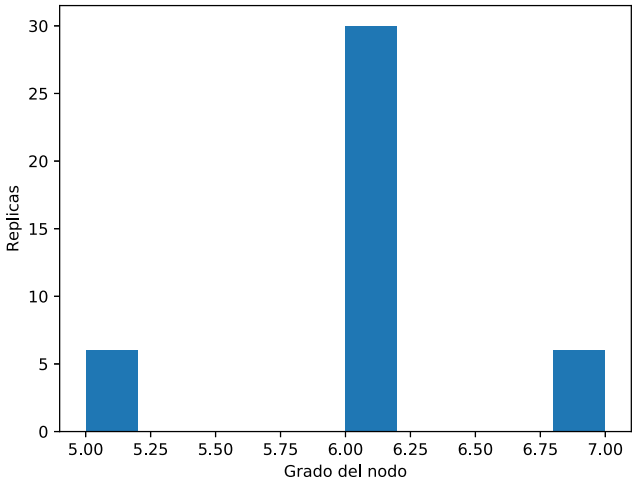


Figura 13: Grafo 3

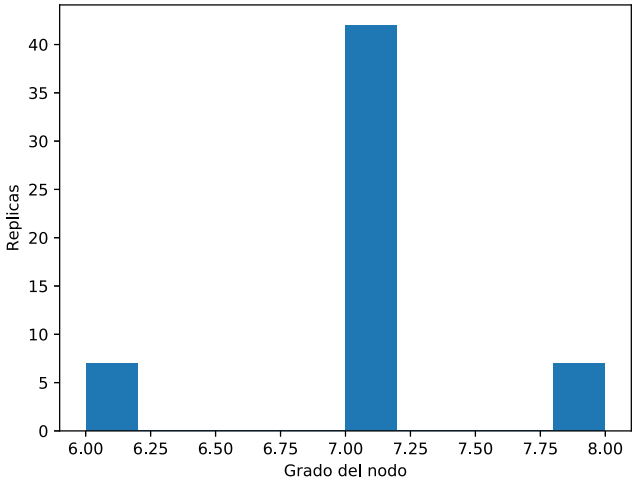


Figura 14: Grafo 4

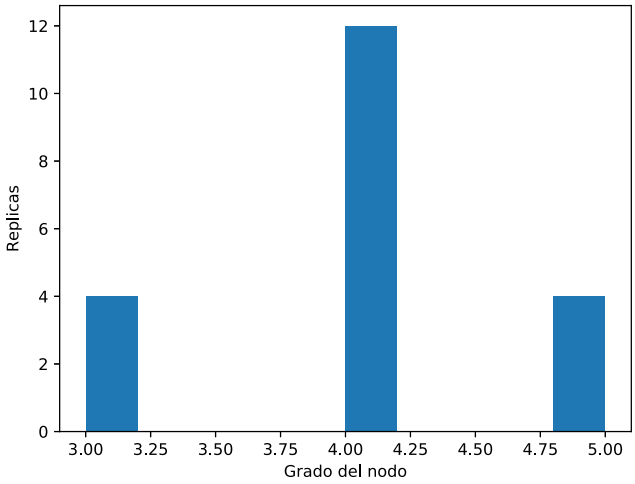


Figura 15: Grafo 5

2.2. Coeficiente de agrupamiento entre nodos

A continuación se muestran histogramas relacionadas a los coeficiente de agrupamiento de los nodos de cada grafo:

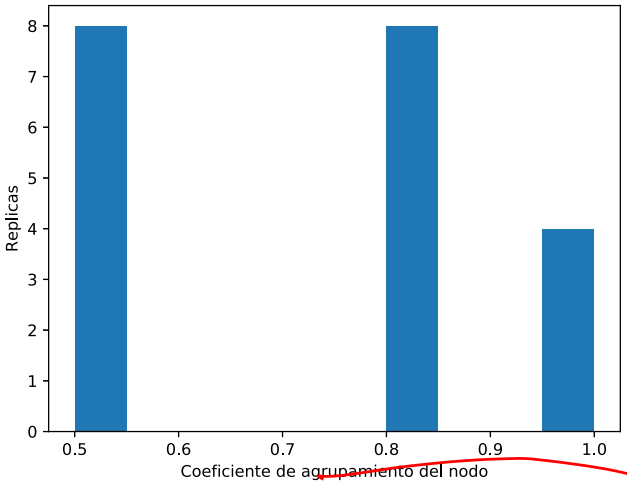


Figura 16: Grafo 1

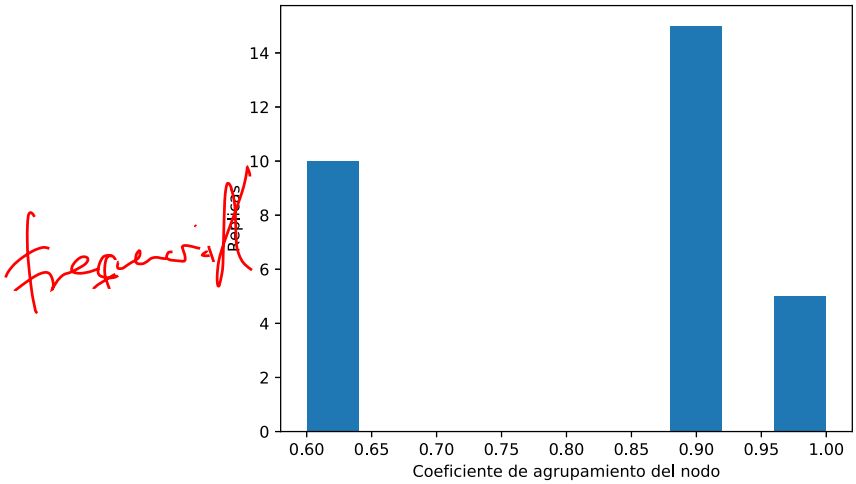


Figura 17: Grafo 2

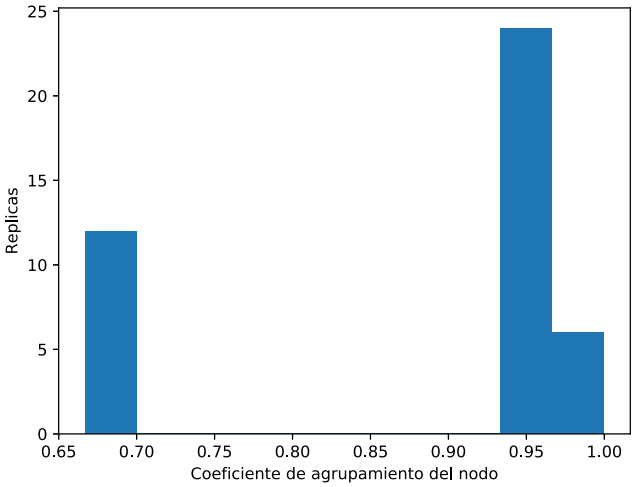


Figura 18: Grafo 3

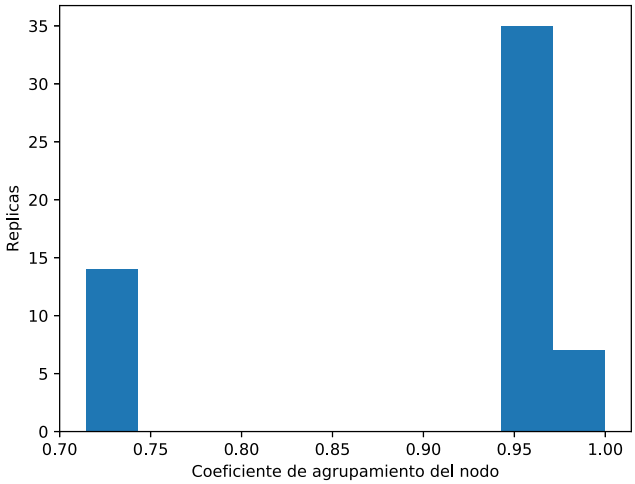


Figura 19: Grafo 4

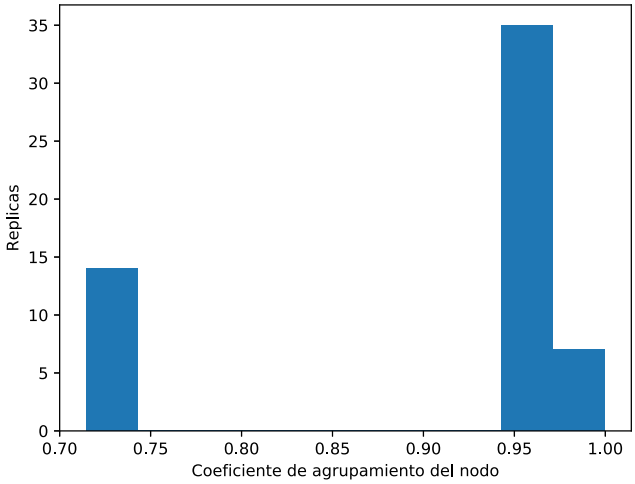


Figura 20: Grafo 5

2.3. Centralidad de cercanía entre nodos

A continuación se muestran histogramas relacionadas a la centralidad de cercanía de los nodos de cada grafo:

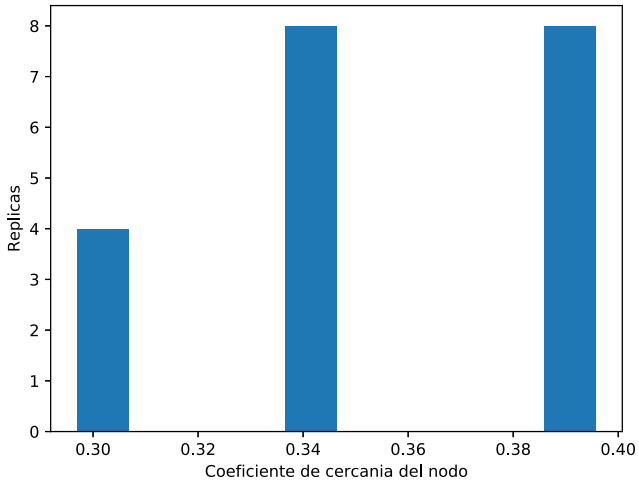


Figura 21: Grafo 1

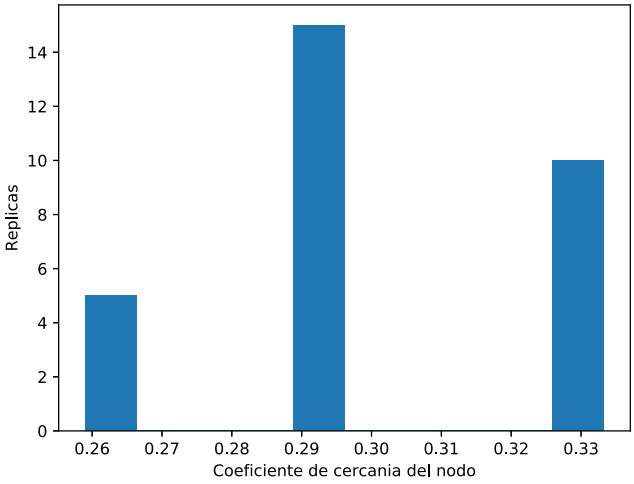


Figura 22: Grafo 2

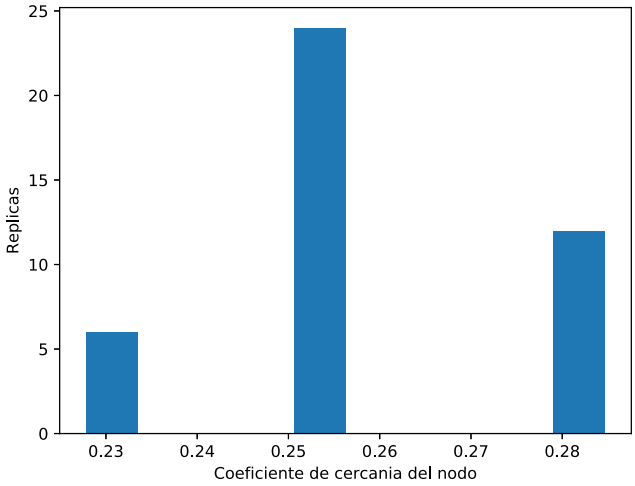


Figura 23: Grafo 3

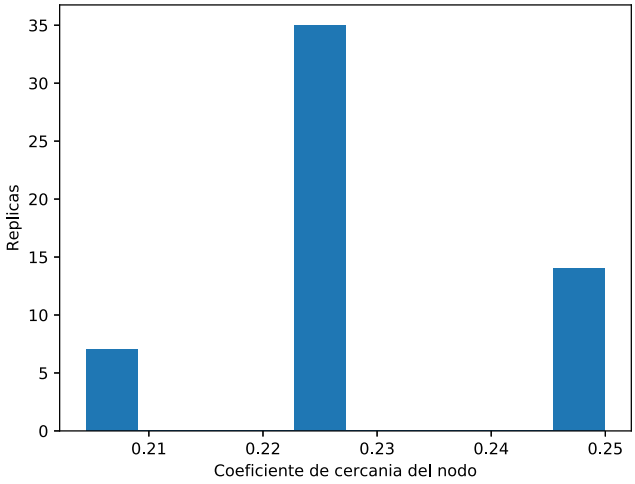


Figura 24: Grafo 4

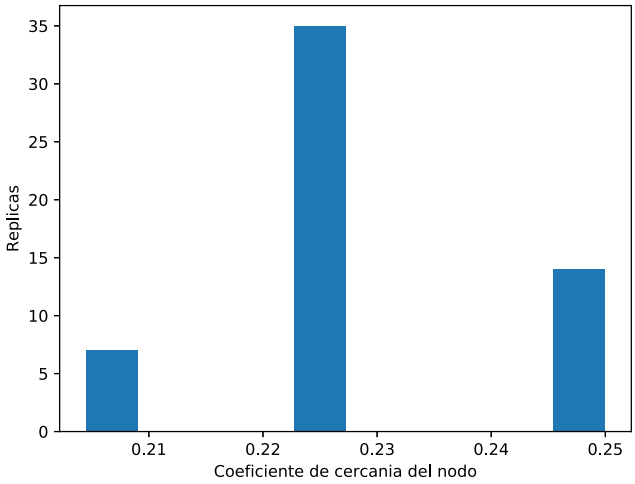


Figura 25: Grafo 5

2.4. Centralidad de carga entre nodos

A continuación se muestran histogramas relacionadas a la centralidad de carga de los nodos de cada grafo:

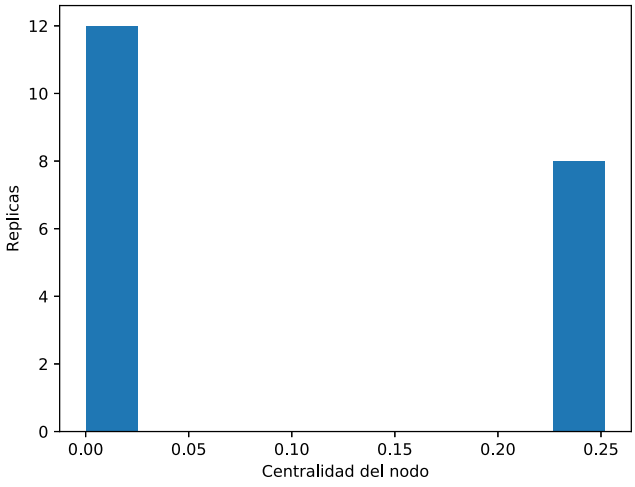


Figura 26: Grafo 1

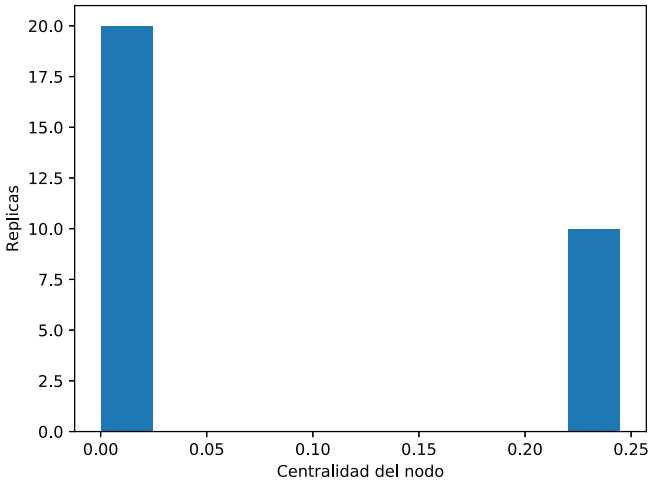


Figura 27: Grafo 2

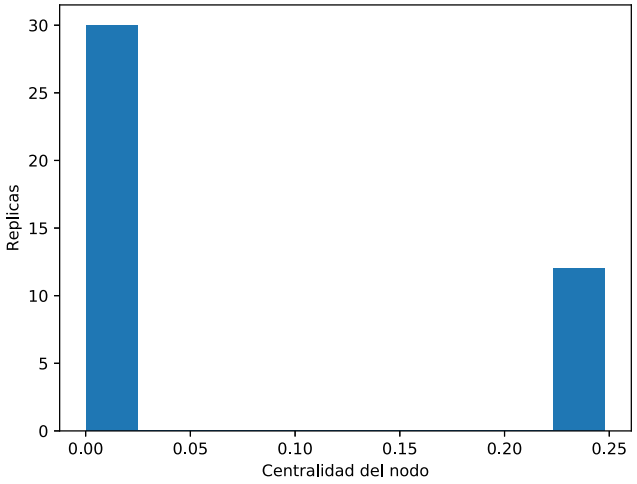


Figura 28: Grafo 3

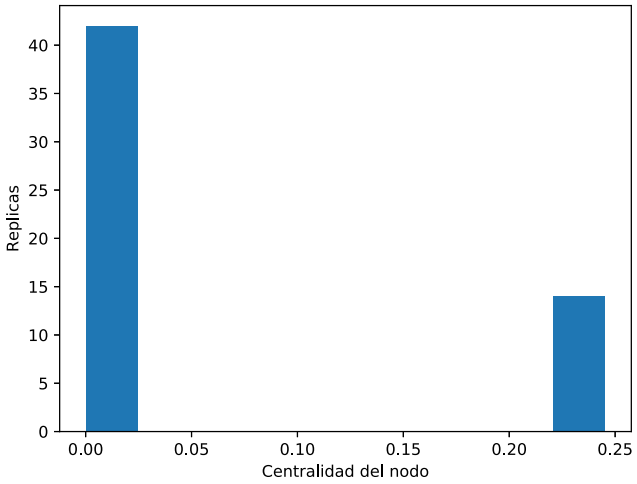


Figura 29: Grafo 4

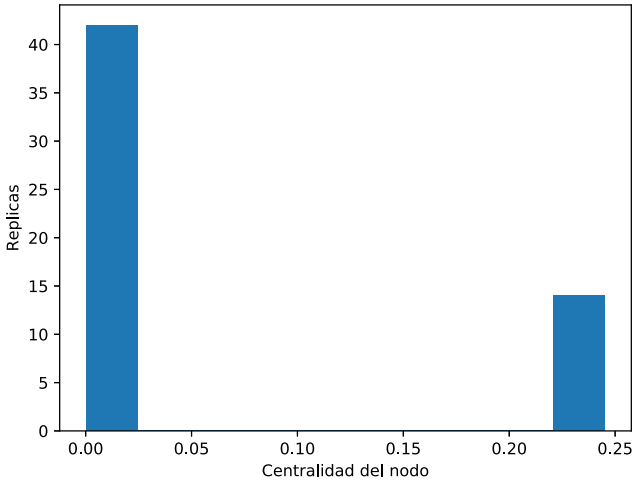


Figura 30: Grafo 5

2.5. Excentricidad entre nodos

A continuación se muestran histogramas relacionadas a la excentricidad de los nodos de cada grafo:

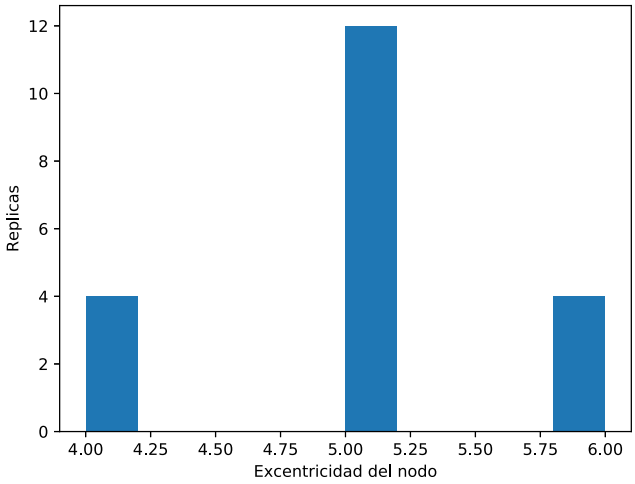


Figura 31: Grafo 1

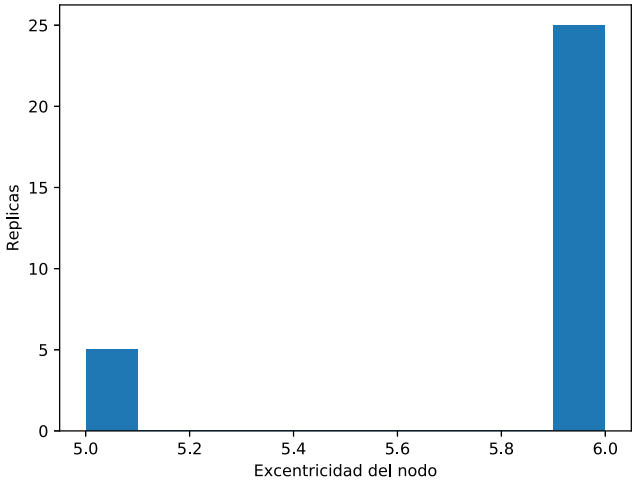


Figura 32: Grafo 2

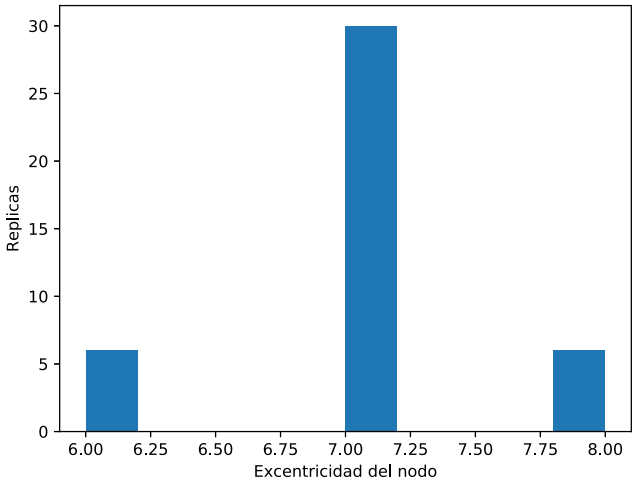


Figura 33: Grafo 3

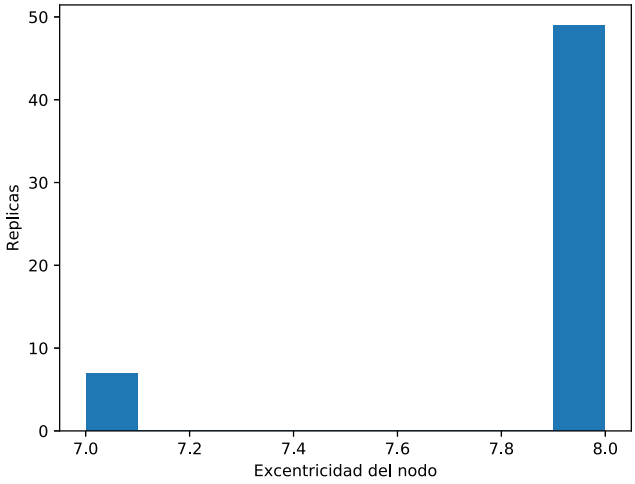


Figura 34: Grafo 4

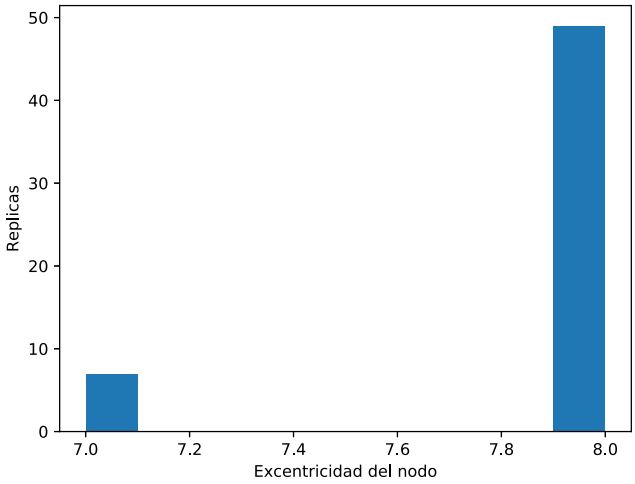


Figura 35: Grafo 5

2.6. PageRank entre nodos

A continuación se muestran histogramas relacionadas al PageRank de los nodos de cada grafo:

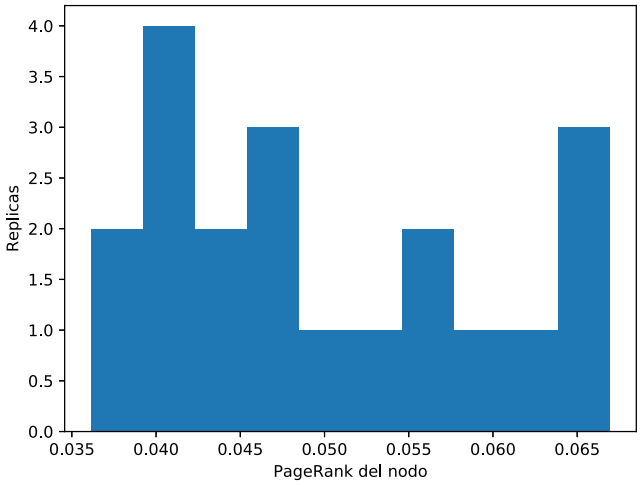


Figura 36: Grafo 1

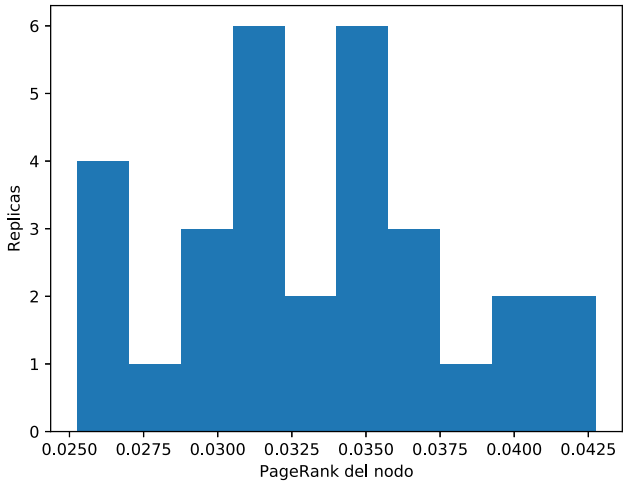


Figura 37: Grafo 2

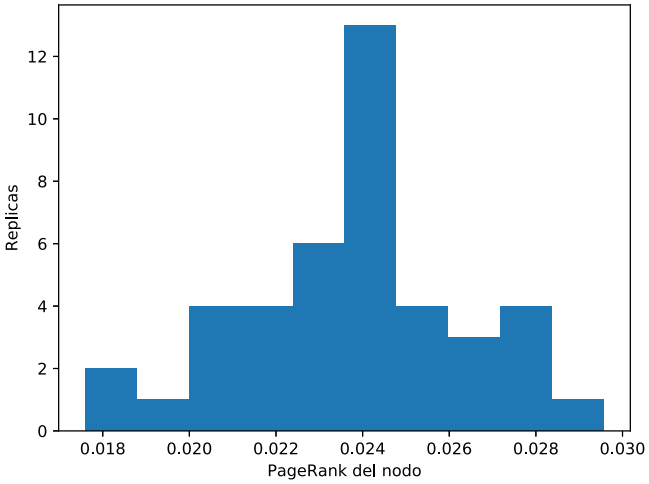


Figura 38: Grafo 3

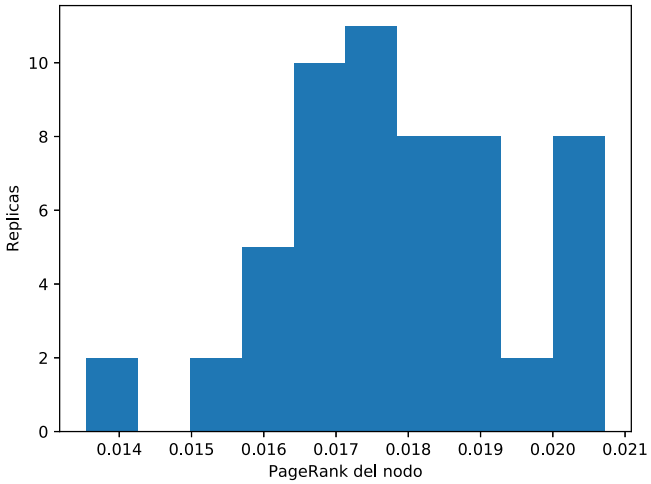


Figura 39: Grafo 4

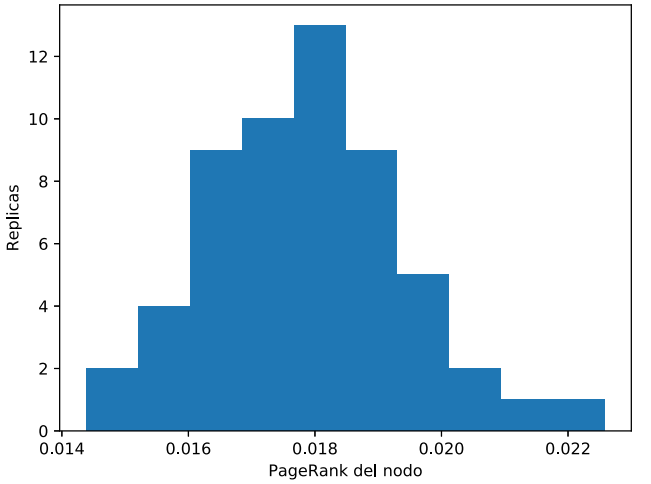


Figura 40: Grafo 5

Se obtuvieron los siguientes resultados de las replicas, las pruebas estadísticas de ANOVA para determinar los efectos de los factores:

	GL	PR(>F)
G	1	1
A	1	1
G:A	1	1
C	1	1
G:C	1	1
A:C	1	0.993
G:A:C	1	1
Ce	1	1
G:Ce	1	1
A:Ce	1	1
G:A:Ce	1	1
C:Ce	1	1
G:C:Ce	1	1
A:C:Ce	1	1
G:A:C:Ce	1	0.093
E	1	1
G:E	1	1
A:E	1	1
G:A:E	1	1
C:E	1	1
G:C:E	1	1
A:C:E	1	1
G:A:C:E	1	0
Ce:E	1	1
G:Ce:E	1	1
A:Ce:E	1	1
G:A:Ce:E	1	0.301
C:Ce:E	1	1
G:C:Ce:E	1	0.66
A:C:Ce:E	1	0
G:A:C:Ce:E	1	0
P	1	0.993
G:P	1	0.97
A:P	1	0.992
G:A:P	1	0.96
C:P	1	0.872
G:C:P	1	1
A:C:P	1	0.992
G:A:C:P	1	0.003
Ce:P	1	1
G:Ce:P	1	1
A:Ce:P	1	1
G:A:Ce:P	1	0.041
C:Ce:P	1	1
G:C:Ce:P	1	0.537
A:C:Ce:P	1	0.676
G:A:C:Ce:P	1	0
E:P	1	0.99
G:E:P	1	1
A:E:P	1	1
G:A:E:P	1	0.003
C:E:P	1	0.958
G:C:E:P	1	0
A:C:E:P	1	0
G:A:C:E:P	1	0
Ce:E:P	1	1
G:Ce:E:P	1	0.635
A:Ce:E:P	1	0.014
G:A:Ce:E:P	1	0
C:Ce:E:P	1	0.519
G:C:Ce:E:P	1	0
A:C:Ce:E:P	1	0
G:A:C:Ce:E:P	1	0.54
Residual	45	0

G	Grado
A	Agrupamiento
C	Centralidad
Ce	Cercanía
E	Excentricidad
P	PageRank

3. Conclusiones

Como conclusión de esta investigación, se tiene que es solamente en combinación de las características estudiadas cuando afecta el rendimiento de tiempo de ejecución del algoritmo de flujo máximo, particularmente el PageRank es la característica que más afecta en combinación con otros. La tabla ANOVA respalda los resultados con los valores de P dados.

Referencias

[1] Python Software Foundation Versión 3.7.2. https://www.python.org/.

[2] NetworkX developers con última actualización el 19 de Septiembre 2018. https://networkx.github.io/documentation/stable/index.html.

[3] NetworkX developers Versión 2.0. https://networkx.github.io/.

[4] The Matplotlib development team Versión 3.0.2. https://matplotlib.org/.

[5] Jesús Angel Patlán Castillo. Repositorio Optimización Flujo en Redes. https://github.com/JAPatlanC/Flujo-Redes.

Tarea 5

Jesus Angel Patlán Castillo (5261)

30 de abril de 2019

En esta tarea se analizan las características de grafos que son generados por un algoritmo de generación de grafos proporcionado por librerías de Python, además de estudiar como estas características afectan el tiempo de ejecución al momento de obtener el flujo máximo entre dos nodos dentro del grafo. Los algoritmos se obtuvieron por medio de la librería NetworkX [3] de Python [1], la librería Matplotlib [4] es utilizada para generar las gráfica de correlación entre los efectos estudiados. El código empleado se obtuvo consultando la documentación oficial de la librería NetworkX [2]. Las imágenes y el código se encuentran disponibles directamente en mi repositorio [5].

1. Metodología

Basándonos en las tareas anteriores, se decide utilizar de algoritmo de generador de grafos de llamado grafo de cavernícola conexo, el cual tiene una mayor aplicación para el problema de flujo máximo dado que es posible representar cada cluster de nodos como una isla en la que se necesita transportar recursos a otro cluster. Además, tenemos que el algoritmo de flujo máximo que utilizaremos será el tradicional flujo máximo entre dos nodos, dado que fue el algoritmo que mostró un mejor desempeño en tiempo de ejecución. Para una mejor visualización se eligió el algoritmo de acomodo resorte, el cual da una mejor visualización de los nodos y las aristas del grafo. Para las pruebas, se crearon cinco instancias distintas de grafo de cavernícola conexo, los cuales se muestran a continuación:

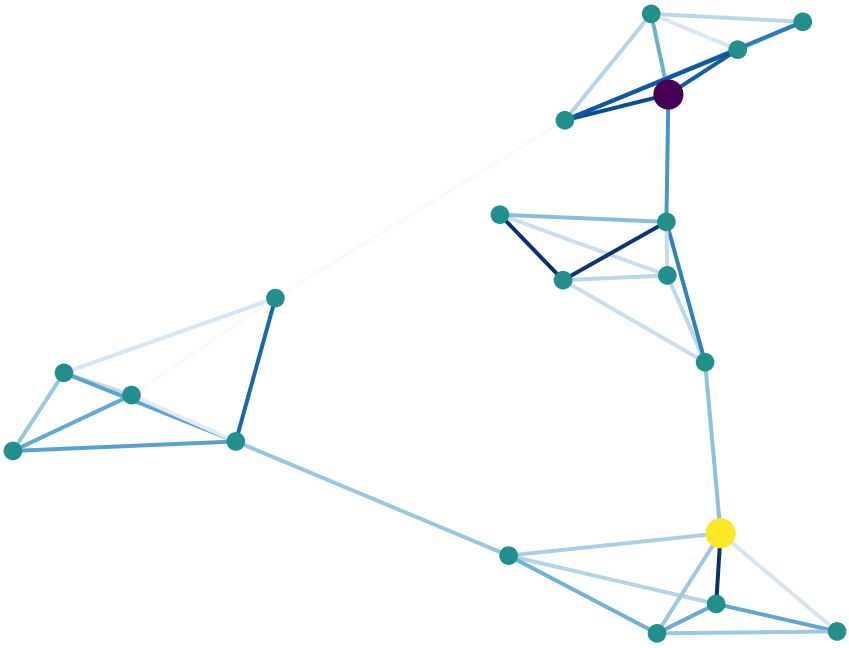


Figura 1: Grafo 1

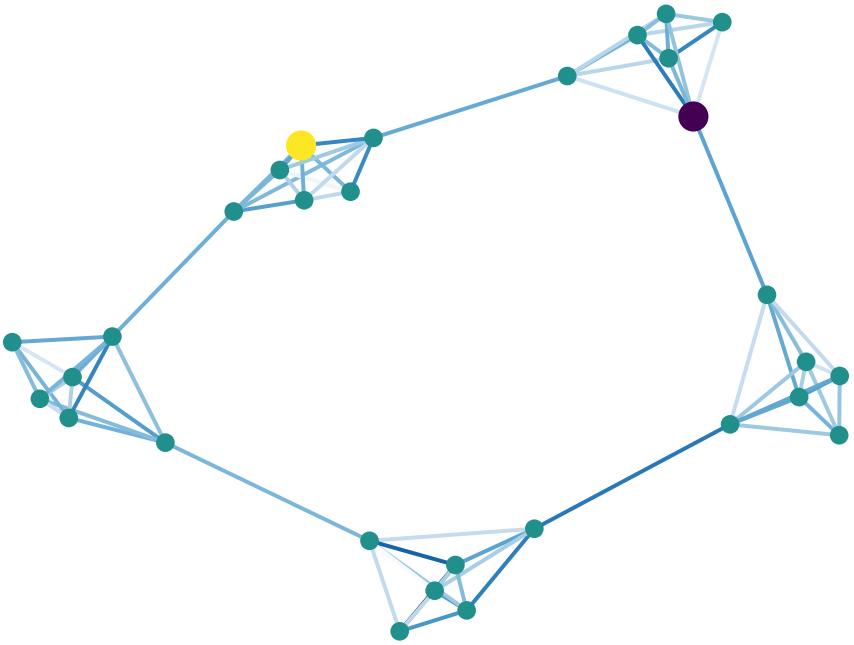


Figura 2: Grafo 2

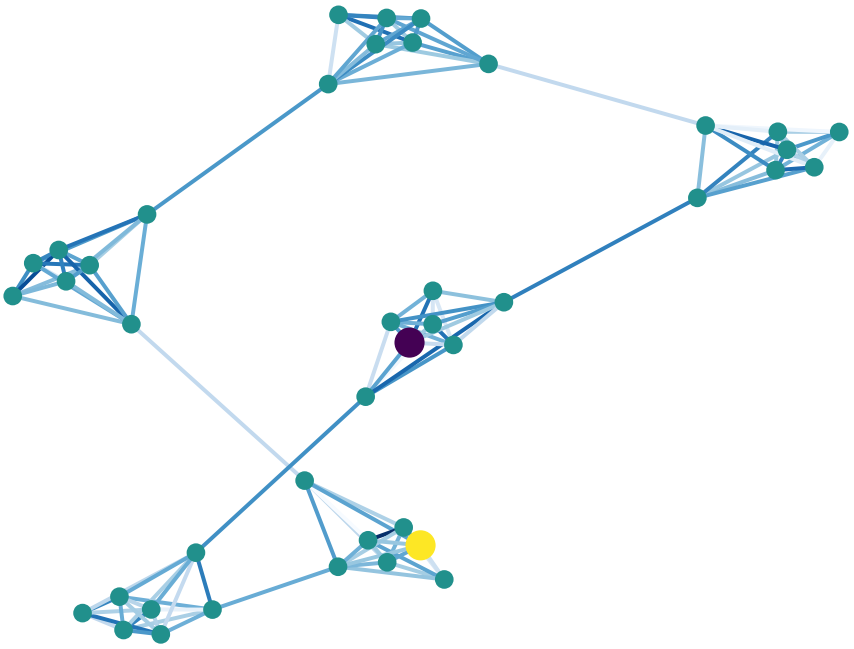


Figura 3: Grafo 3

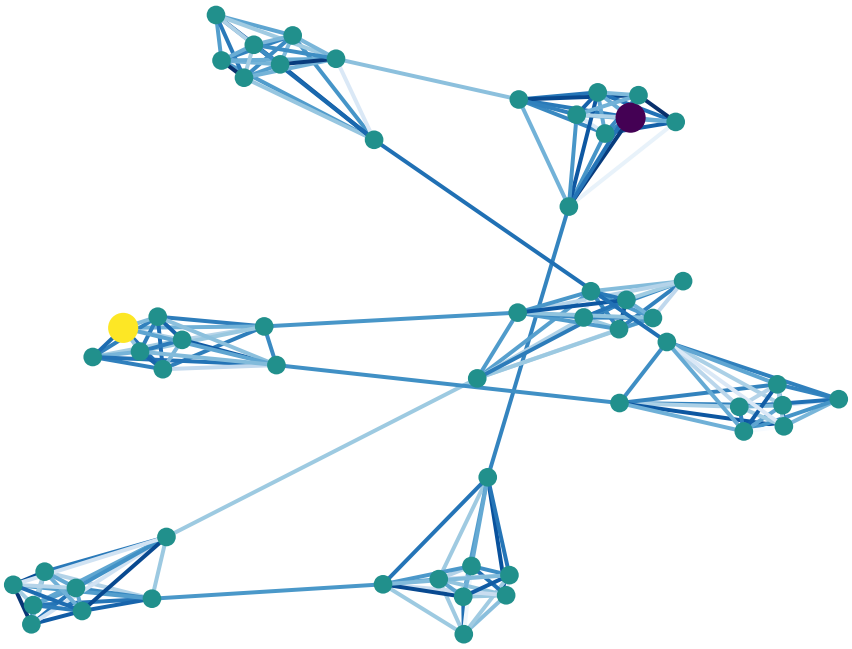


Figura 4: Grafo 4

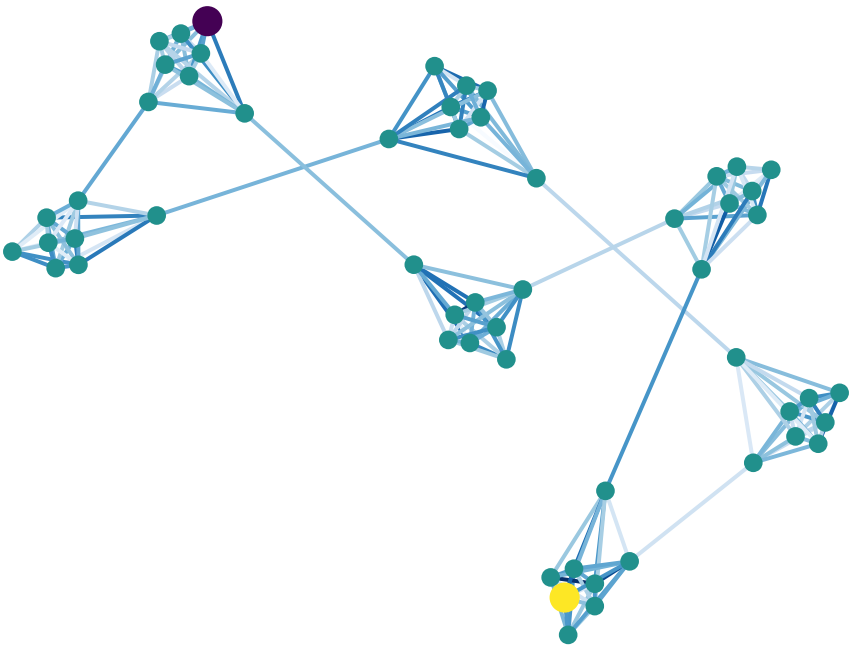


Figura 5: Grafo 5

Para cada grafo, señalamos un nodo fuente (en amarillo) y un nodo sumidero (en negro) de ejemplo para el problema de flujo máximo, además que resaltamos cada arista con un tono de azul más oscuro cuanto mayor sea el peso que tenga. Para cada uno de los grafos, se realizaron distintas pruebas intercambiando los nodos fuente-sumidero para determinar el rendimiento del algoritmo de flujo máximo. Además, a cada grafo se le determinan cinco características:

- Distribución de grado: Es el número de aristas asociadas a un nodo.
- Coeficiente de agrupamiento: Determina que tanto un nodo está asociado a sus nodos vecinos.
- Centralidad de cercanía: Determina qué tan cercano o alejado está un nodo con respecto a los demás nodos vecinos.
- Centralidad de carga: Determina el número de caminos más cortos que se pasan a través del nodo.
- Excentricidad: Determina la longitud máxima que hay entre el nodo y otro nodo del grafo.
- PageRank: Similar a la distribución de grado, pero da pesos a las aristas de acuerdo al grado que tenga el nodo al que se conecta.

Para la obtención de estas características y la generación de los grafos, se utilizó el siguiente código:

```
def graph_attributes(G):
    global num_fig
    nodes=G.nodes()
    degrees=[G.degree(i) for i in nodes]
    clustering=[nx.clustering(G,i) for i in nodes]
    closeness=[nx.closeness_centrality(G,i) for i in nodes]
    load=[nx.load_centrality(G,i) for i in nodes]
    eccentricity=[nx.eccentricity(G,i) for i in nodes]
    pageRank=nx.pagerank(G,weight="capacity")
    pageRank_G=[pageRank[i] for i in nodes]
    plt.figure(0)
    plt.ylabel('Replicas')
    plt.xlabel('Grado del nodo')
    plt.hist(degrees)
    plt.figure(1)
    plt.savefig('hist-grado-'+str(num_fig)+'.eps', format='eps',
    dpi=1000)
    #plt.clf()
    plt.ylabel('Replicas')
    plt.xlabel('Coeficiente de agrupamiento del nodo')
    plt.hist(clustering)
    plt.figure(2)
    plt.savefig('hist-agrupamiento-' + str(num_fig) + '.eps',
    format='eps', dpi=1000)
    plt.clf()
    plt.ylabel('Replicas')
    plt.xlabel('Coeficiente de cercania del nodo')
    plt.hist(closeness)
    plt.figure(3)
    plt.savefig('hist-cercania-' + str(num_fig) + '.eps', format='
    eps', dpi=1000)
    plt.clf()
    plt.ylabel('Replicas')
    plt.xlabel('Centralidad del nodo')
    plt.hist(load)
    plt.figure(4)
    plt.savefig('hist-centralidad-' + str(num_fig) + '.eps', format
    ='eps', dpi=1000)
    plt.clf()
    plt.ylabel('Replicas')
    plt.xlabel('Excentricidad del nodo')
    plt.hist(eccentricity)
    plt.figure(5)
    plt.savefig('hist-excentricidad-' + str(num_fig) + '.eps',
    format='eps', dpi=1000)
    plt.clf()
    plt.ylabel('Replicas')
    plt.xlabel('PageRank del nodo')
    plt.hist(pageRank_G)
    plt.figure(6)
    plt.savefig('hist-pagerank-' + str(num_fig) + '.eps', format='
    eps', dpi=1000)
    plt.clf()
```

En base a estas cinco características, se realiza una prueba ANOVA para deter-

minar como influyen estos en el tiempo de ejecución del algoritmo. Un ejemplo aplicando el flujo máximo en cada grafo se muestra a continuación, donde se muestran solamente las aristas por donde el flujo se distribuye:

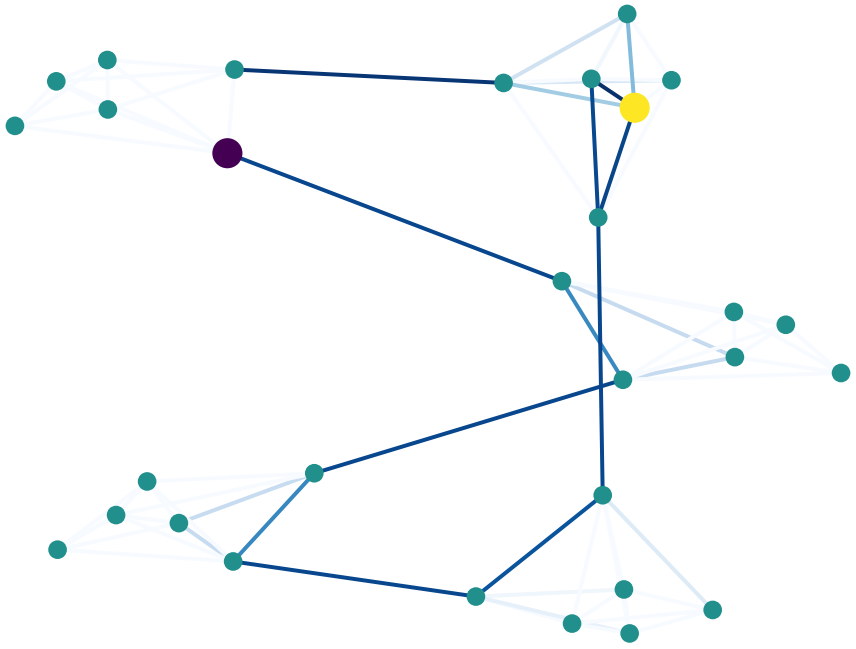


Figura 6: Grafo 1

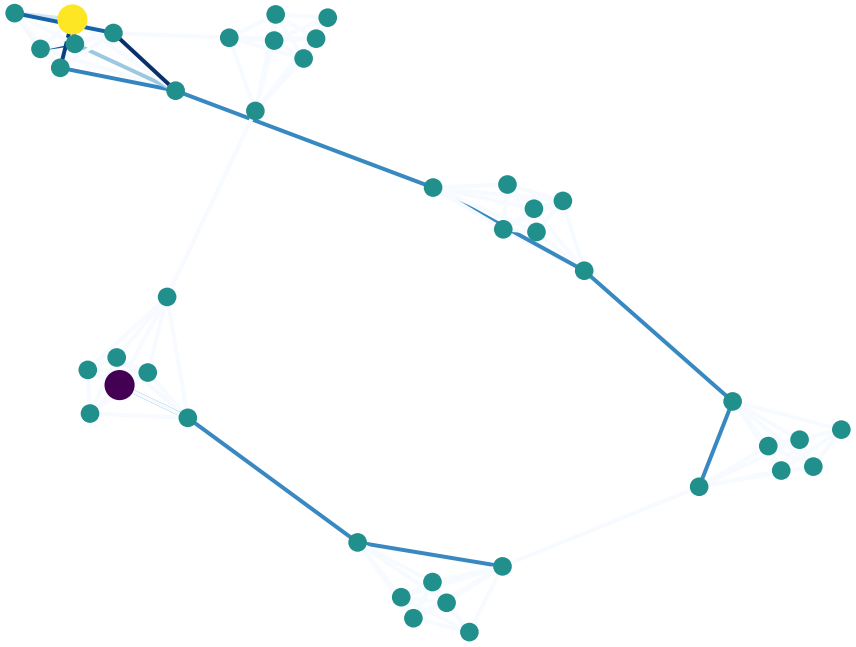


Figura 7: Grafo 2

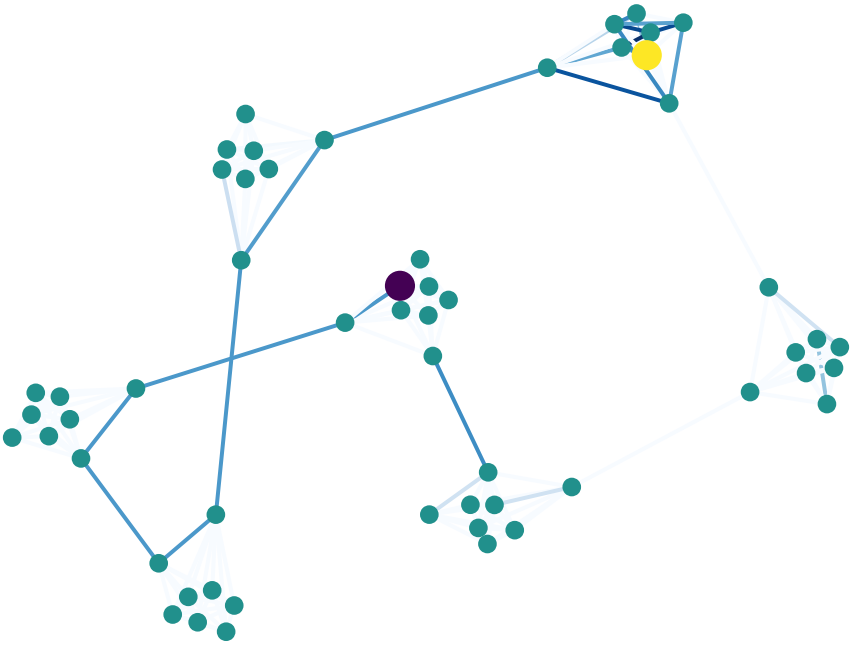


Figura 8: Grafo 3

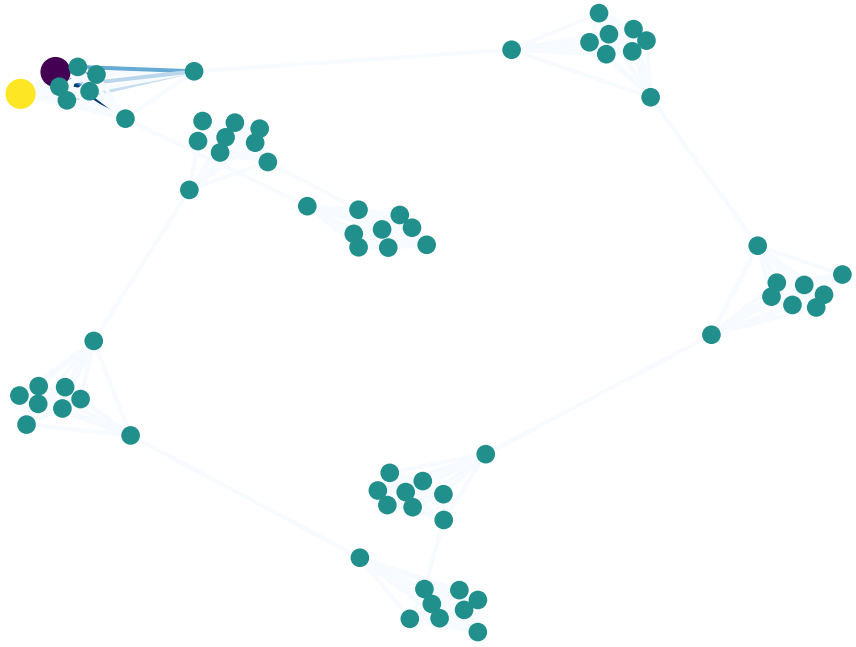


Figura 9: Grafo 4

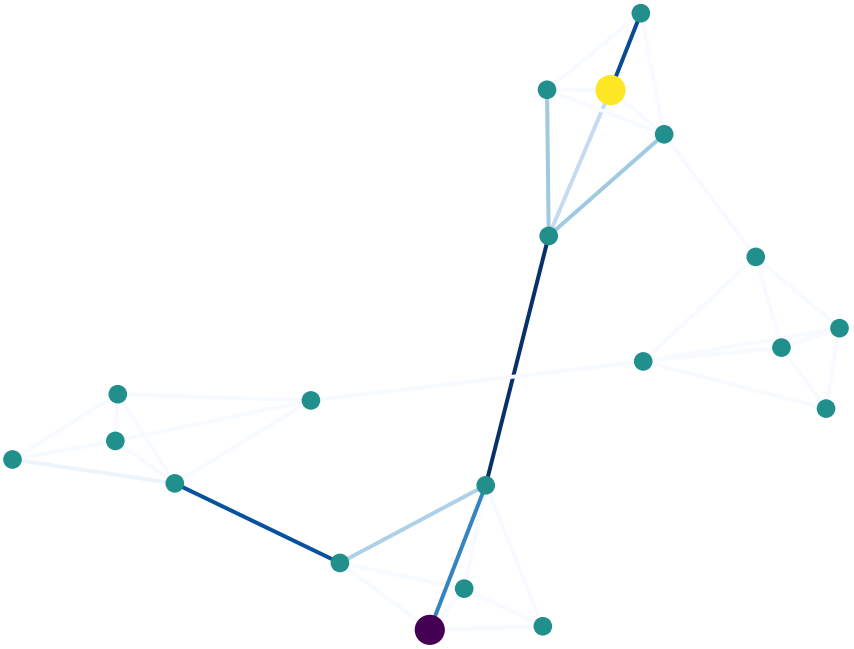


Figura 10: Grafo 5

Este proceso se realizó en una computadora con las siguientes características:

- Procesador: Intel Core i7-7500U 2.7GHz
- Memoria RAM: 16GB
- Sistema Operativo: Windows 10 64 bits

Los resultados fueron determinados por medio de un ANOVA utilizando el siguiente código:

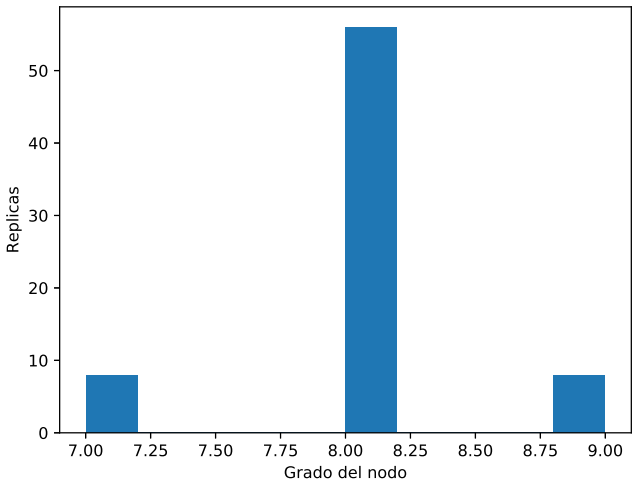
```
dataframe = pd.DataFrame(total_data)
dataframe.to_csv('datos.xls',sep='\t')
model = ols('Tiempo~G*A*C*Ce*E*P',data=dataframe).fit()
anova = anova_lm(model,typ=2)
anova.to_csv('anova1.xls',sep='\t')
```

2. Resultados

Estas son las características encontradas de las instancias de grafos:

2.1. Grado de nodos

Se muestran histogramas relacionadas a los grados de los nodos de cada grafo:



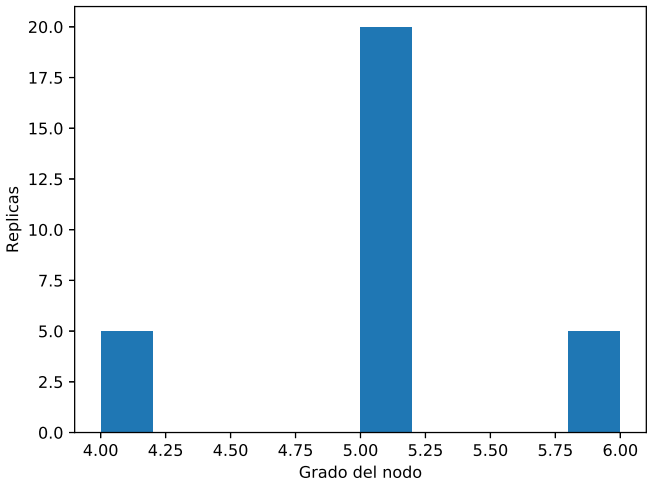


Figura 11: Grafo 2

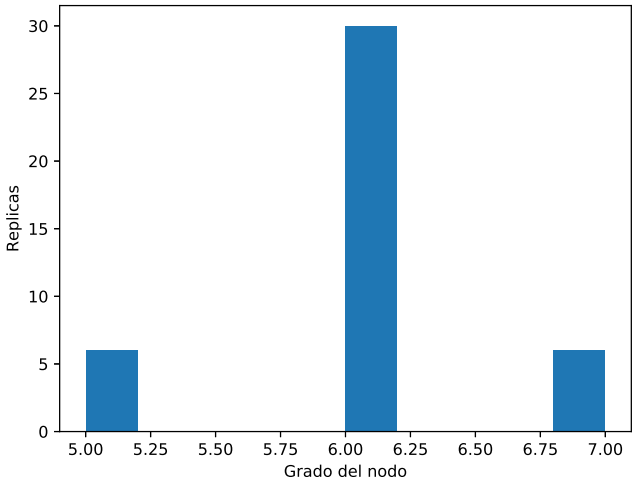


Figura 12: Grafo 3

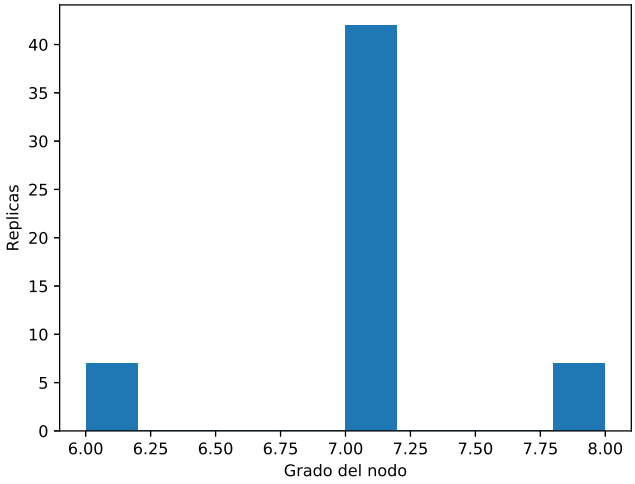


Figura 13: Grafo 4

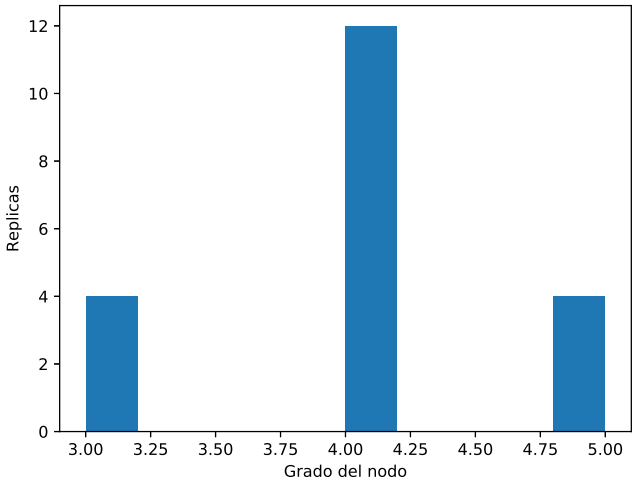


Figura 14: Grafo 5

2.2. Coeficiente de agrupamiento entre nodos

Se muestran histogramas relacionadas a los coeficiente de agrupamiento de los nodos de cada grafo:

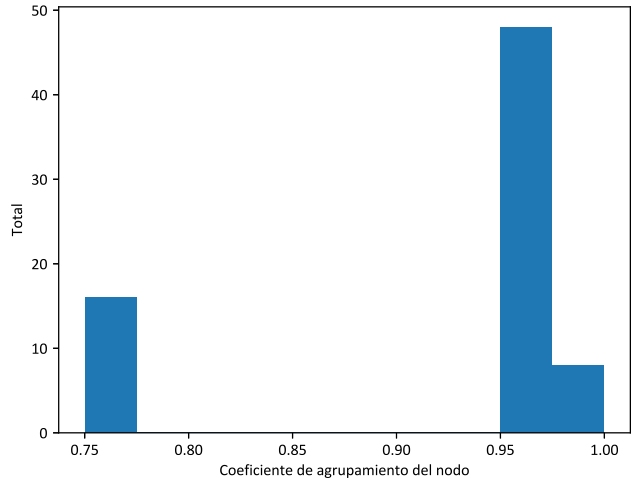


Figura 15: Grafo 1

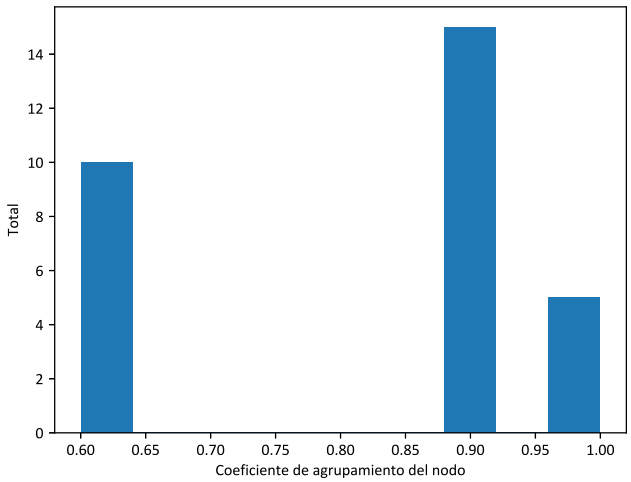


Figura 16: Grafo 2

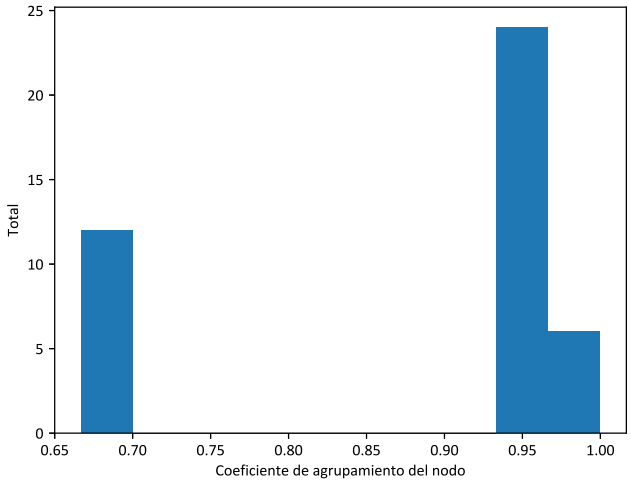


Figura 17: Grafo 3

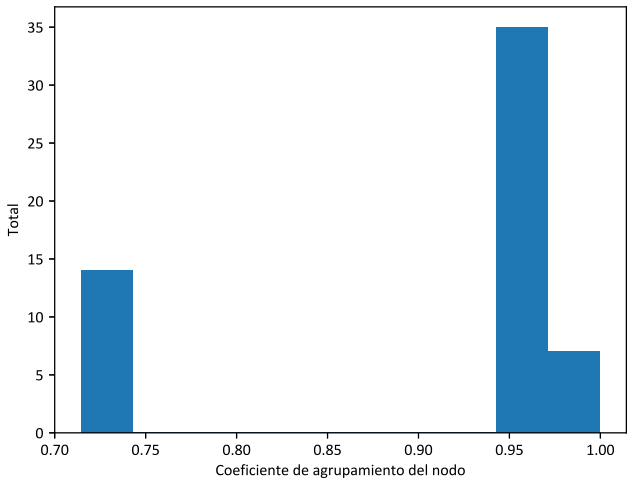


Figura 18: Grafo 4

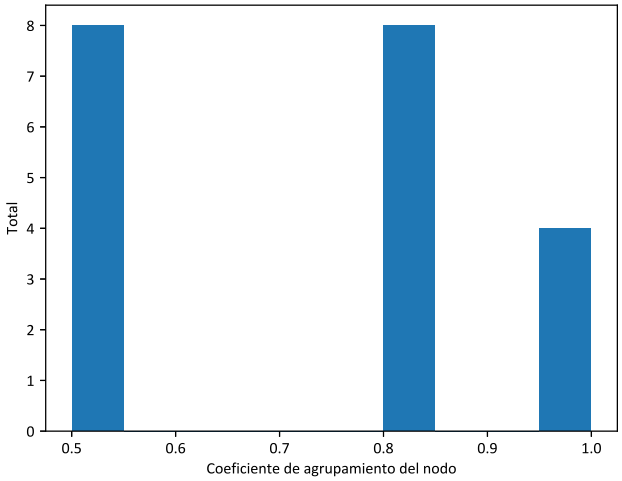


Figura 19: Grafo 5

2.3. Centralidad de cercanía entre nodos

Se muestran histogramas relacionadas a la cercanía de nodos de cada grafo:

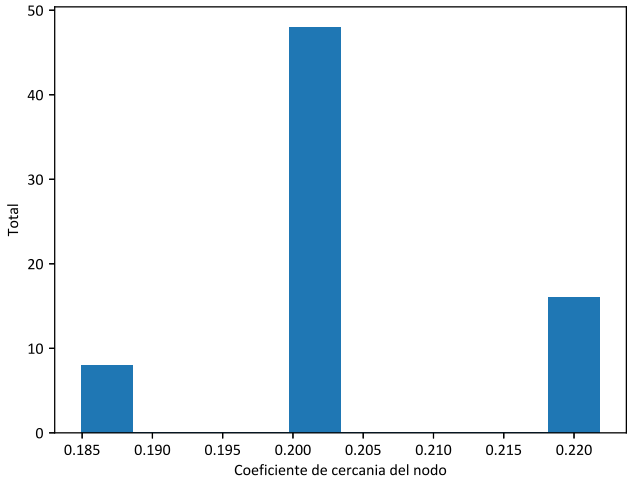


Figura 20: Grafo 1

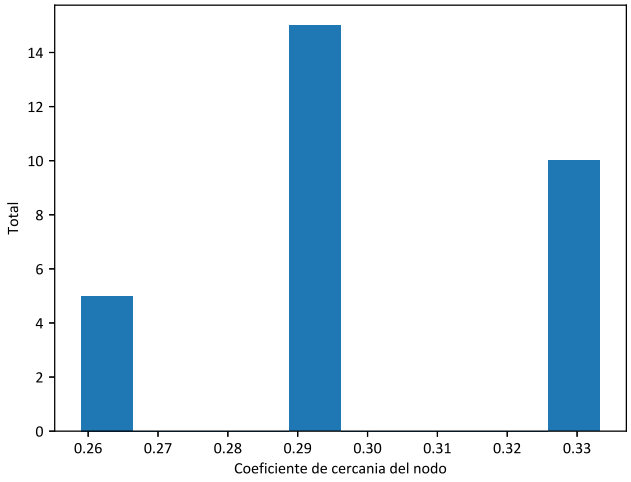


Figura 21: Grafo 2

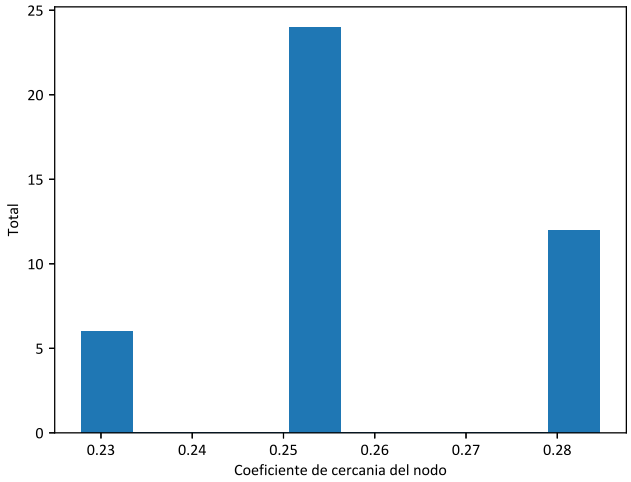


Figura 22: Grafo 3

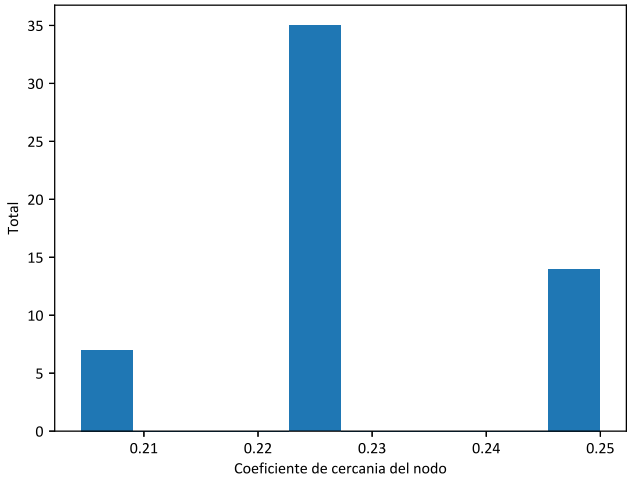


Figura 23: Grafo 4

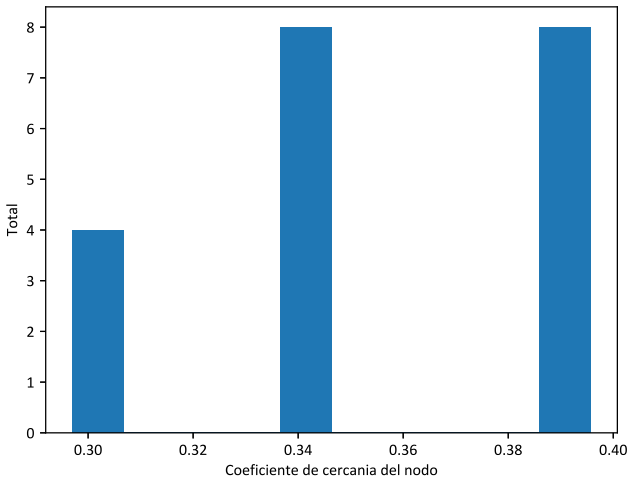


Figura 24: Grafo 5

2.4. Centralidad de carga entre nodos

Se muestran histogramas relacionadas a la centralidad de carga de los nodos de cada grafo:

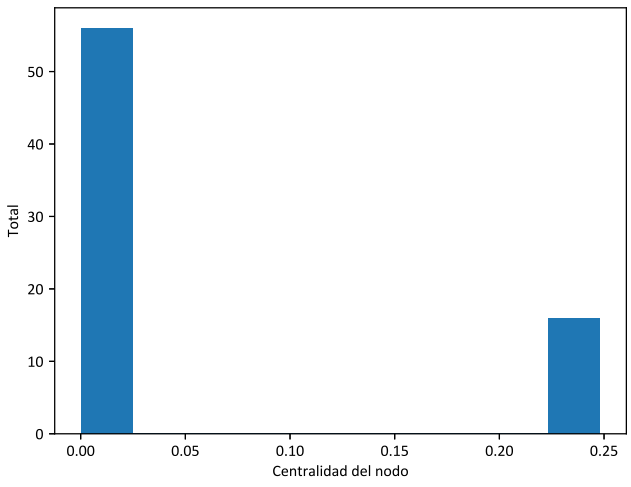


Figura 25: Grafo 1

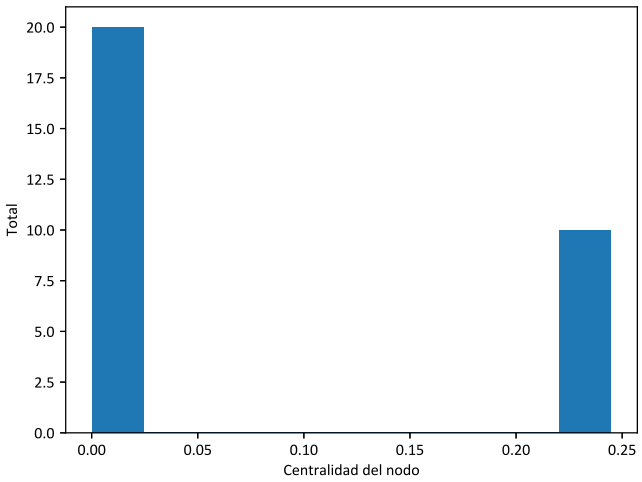


Figura 26: Grafo 2

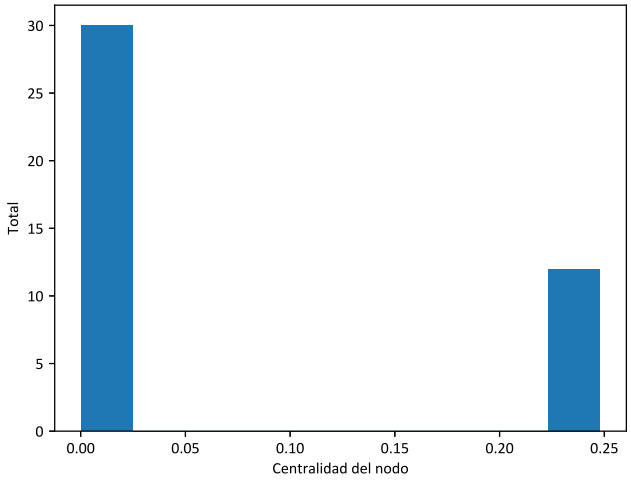


Figura 27: Grafo 3

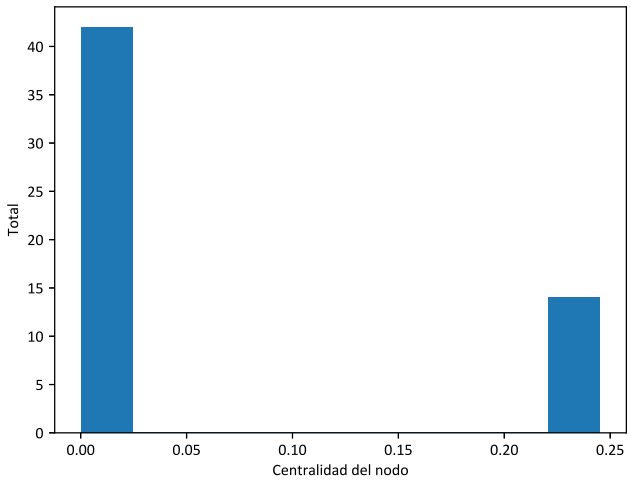


Figura 28: Grafo 4

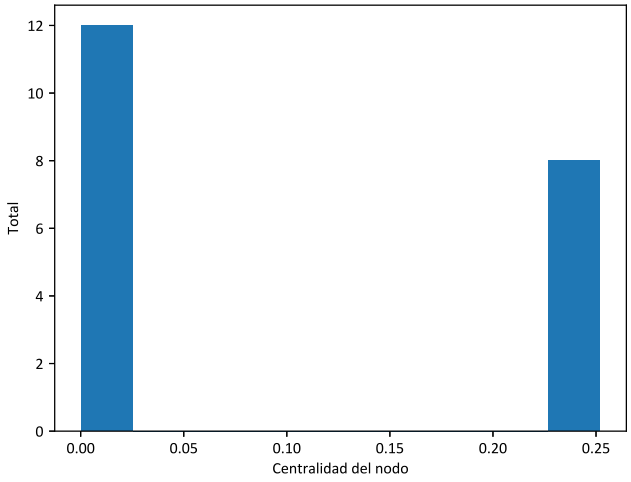


Figura 29: Grafo 5

2.5. Excentricidad entre nodos

Se muestran histogramas relacionadas a la excentricidad de los nodos de cada grafo:

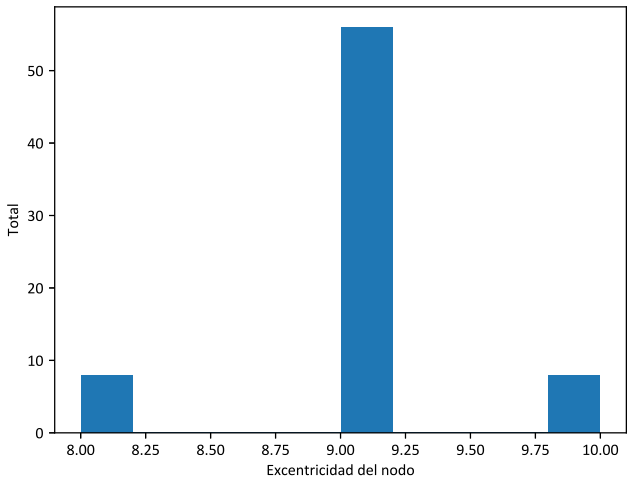


Figura 30: Grafo 1

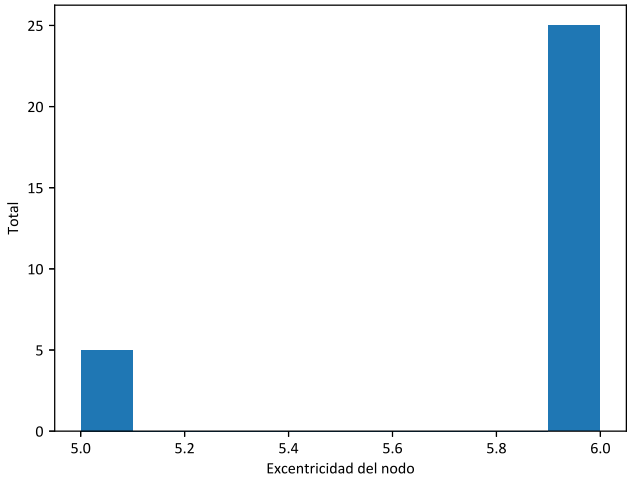


Figura 31: Grafo 2

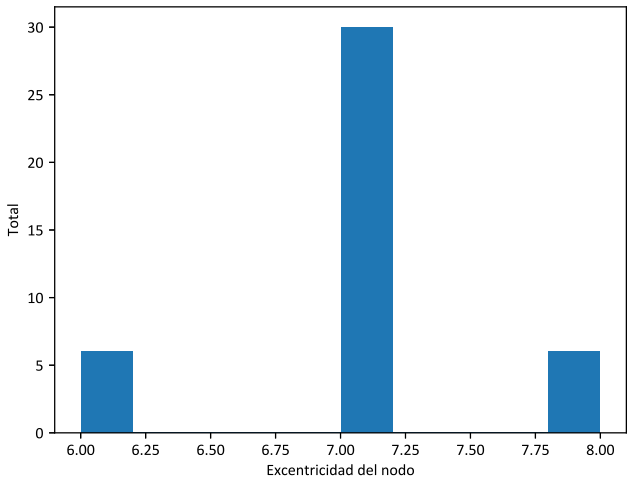


Figura 32: Grafo 3

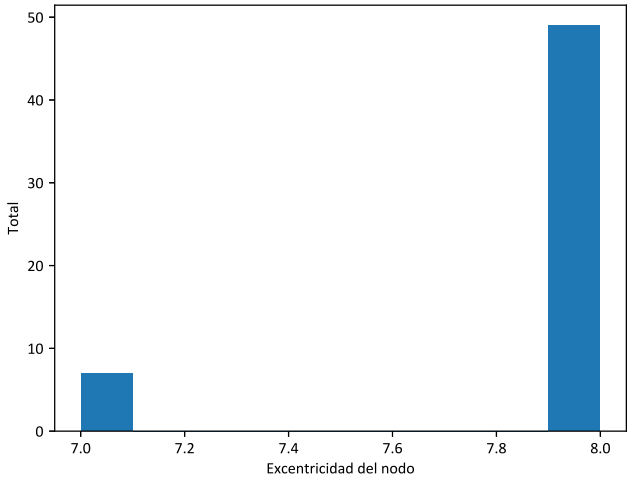


Figura 33: Grafo 4

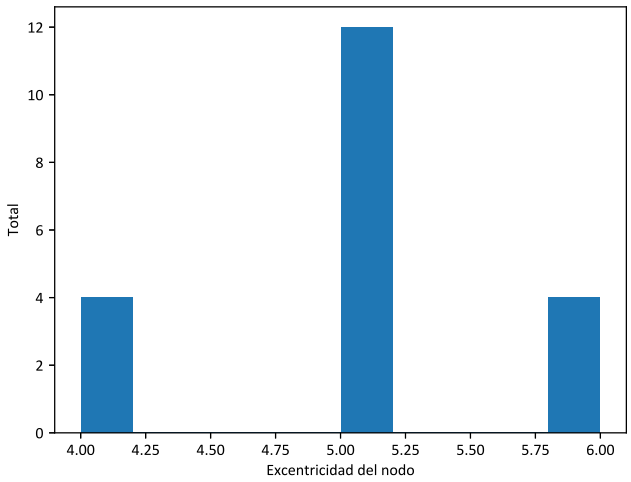


Figura 34: Grafo 5

2.6. PageRank entre nodos

Se muestran histogramas relacionadas al PageRank de los nodos de cada grafo:

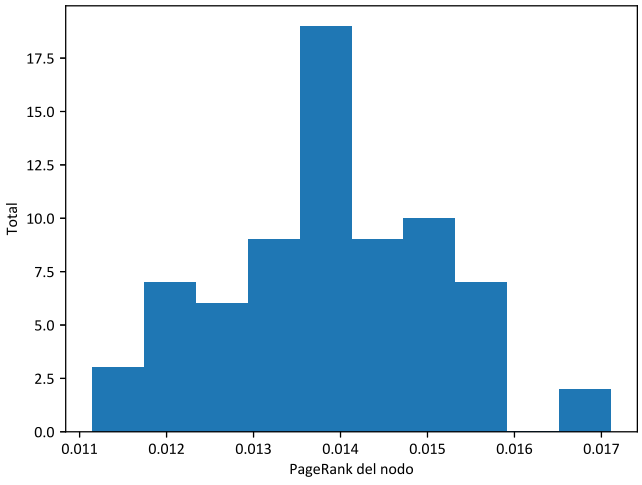


Figura 35: Grafo 1

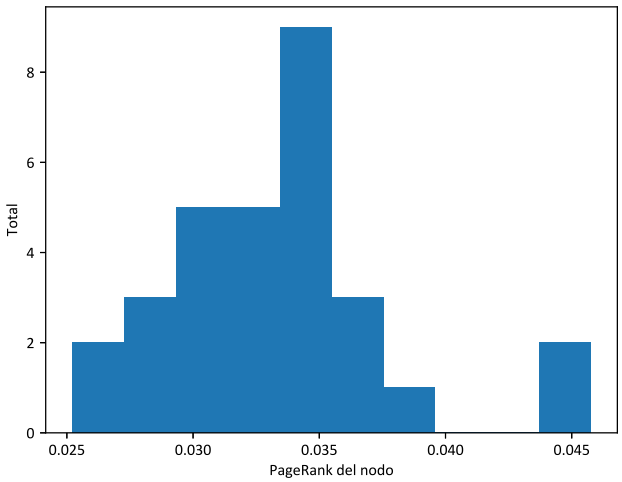


Figura 36: Grafo 2

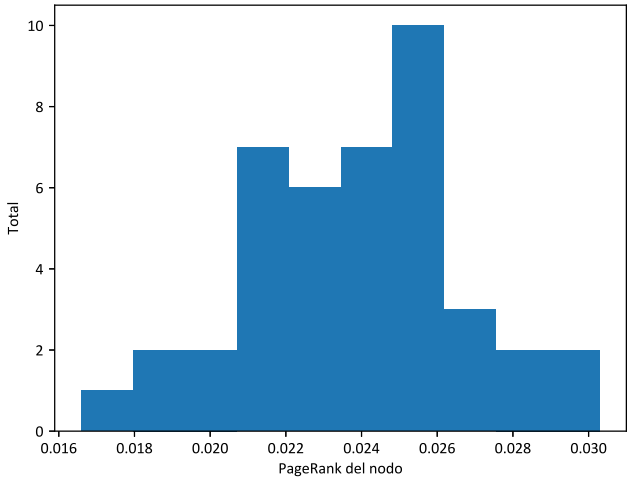


Figura 37: Grafo 3

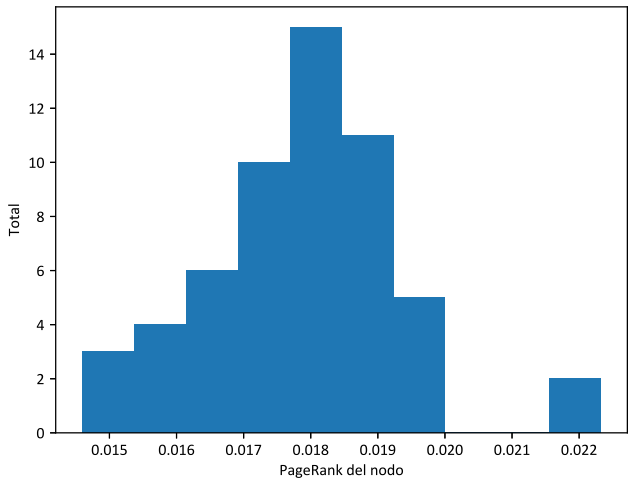


Figura 38: Grafo 4

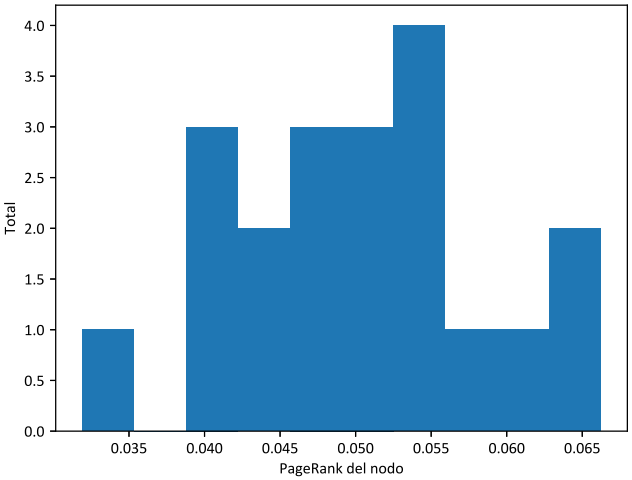


Figura 39: Grafo 5

Se obtuvieron los siguientes resultados de las réplicas, las pruebas estadísticas de ANOVA para determinar los efectos de los factores, los gráficos caja y bigote muestran como la excentricidad y el grado del afectan el rendimiento en el tiempo de ejecución. La tabla muestra únicamente las combinaciones que afectan

el rendimiento según su valor P:

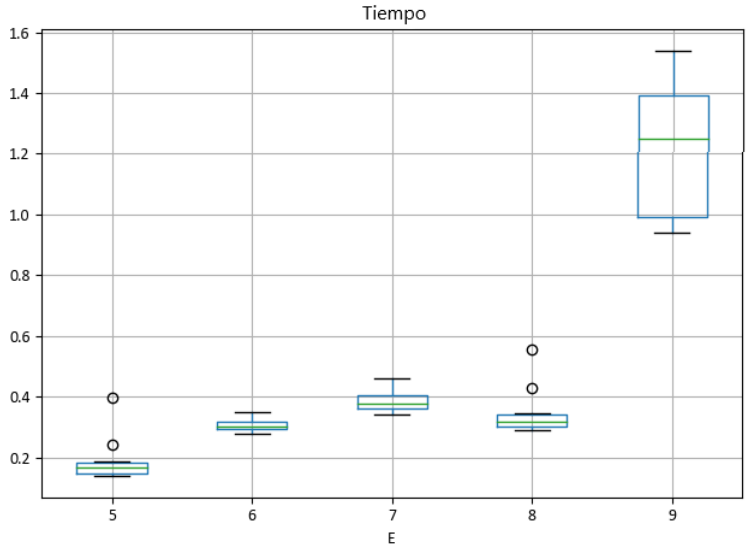


Figura 40: Gráfico caja y bigote de la excentricidad

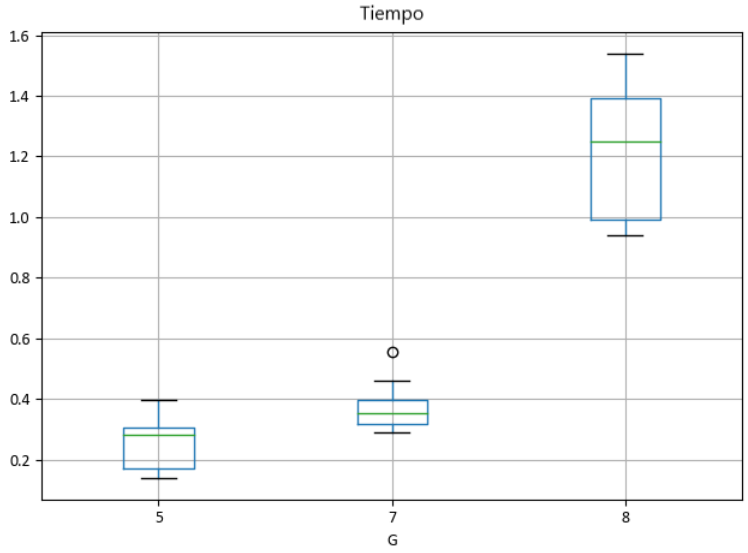


Figura 41: Gráfico caja y bigote del grado

Estrategia	GL	PR(>F)
G:A:C:Ce	1	0.093
G:A:C:E	1	0
G:A:Ce:E	1	0.301
G:C:Ce:E	1	0.66
G:A:C:P	1	0.003
G:A:Ce:P	1	0.041
G:C:Ce:P	1	0.537
G:C:Ce:P	1	0.676
A:C:Ce:P	1	0
G:A:C:Ce:P	1	0.003
G:A:E:P	1	0
G:C:E:P	1	0
A:C:E:P	1	0
G:A:C:E:P	1	0.635
G:Ce:E:P	1	0.014
A:Ce:E:P	1	0
C:Ce:E:P	1	0.519
G:C:Ce:E:P	1	0
A:C:Ce:E:P	1	0
G:A:C:Ce:E:P	1	0.54
Residual	45	0

G	Grado
A	Agrupamiento
C	Centralidad
Ce	Cercanía
E	Excentricidad
P	PageRank

3. Conclusiones

Como conclusión de esta investigación, se tiene que es solamente en combinación de las características estudiadas cuando afecta el rendimiento de tiempo de ejecución del algoritmo de flujo máximo, particularmente el PageRank es la característica que más afecta en combinación con otros. La tabla ANOVA respalda los resultados con los valores de P dados.

Referencias

[1] Python Software Foundation Versión 3.7.2. https://www.python.org/.

[2] NetworkX developers con última actualización el 19 de Septiembre 2018. https://networkx.github.io/documentation/stable/index.html.

[3] NetworkX developers Versión 2.0. https://networkx.github.io/.

[4] The Matplotlib development team Versión 3.0.2. https://matplotlib.org/.

[5] Jesús Angel Patlán Castillo. Repositorio Optimización Flujo en Redes. https://github.com/JAPatlanC/Flujo-Redes.

7. Conclusión de aprendizaje

En este curso aprendí muchas propiedades acerca de los grafos y como estos impactan en el rendimiento de solución a problemas de flujo máximo, además de aplicar métodos estadísticos como ANOVA para determinar si ciertas propiedades del grafo tienen un impacto significativo directo en el tiempo de ejecución de los algoritmos utilizados en estas tareas. Además, aprendí a utilizar el lenguaje Python para realizar las pruebas estadísticas y la generación de grafos, y dentro de Latex aprendí a utilizar diversos paquetes para el diseño de documentos y muchos aspectos claves para tener un trabajo limpio y formal. Me encuentro bastante satisfecho por lo aprendido en clase y considero que aprendí más de lo que esperaba, sobretodo sobre como elaborar un trabajo científico.

Referencias

[1] Jesús Angel Patlán Castillo. Repositorio Optimización Flujo en Redes. https://github.com/JAPatlanC/Flujo-Redes.