Optimización de flujo en redes
Portafolio de evidencias

Universidad Autónoma de Nuevo León
FIME

Posgrado de ingeniería en sistemas
Docente: Dra. Elisa Schaffer con colaboracion del Dr. Arturo Berrones

1455175: Angel Moreno

Semestre enero-junio 2019

6

Optimización de flujo en redes

~~1455175: Angel Moreno~~

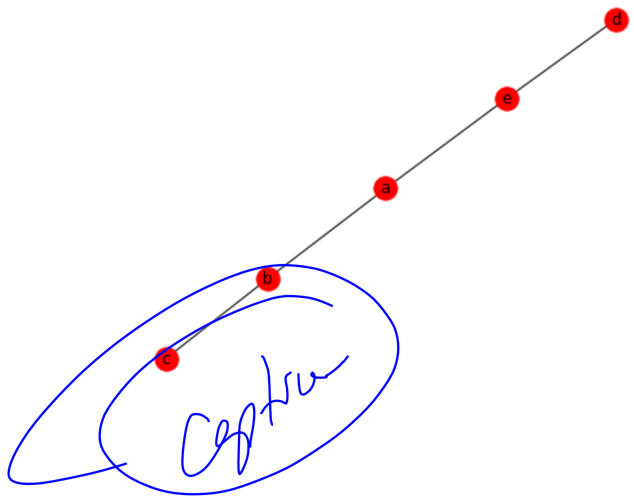
Tarea #1

12 de febrero de 2019

1. Grafo simple no dirigido acíclico

Este primer grafo es el más sencillo, como ejemplo aplicado se utiliza para la representación de ubicaciones de localidades ó ciudades para tener representación gráfica.

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

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "c"), ("a", "e"), ("d", "e")]

G = nx.Graph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)

nx.draw(G, with_labels = True)
plt.show()
```

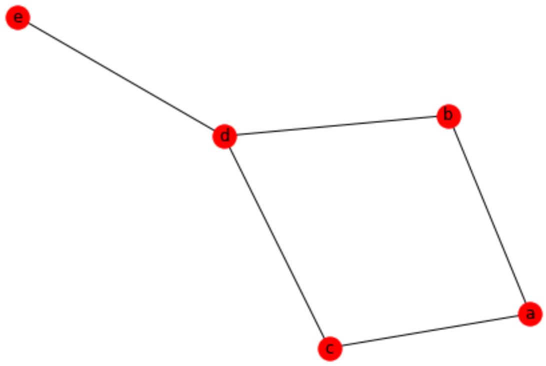


2. Grafo simple no dirigido cíclico

El metro o sistema de transporte es un ejemplo aplicado.

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

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e")]

```

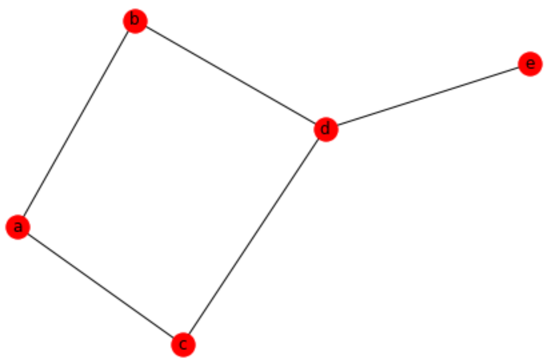
```python
G = nx.Graph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)

nx.draw(G, with_labels = True)
plt.show()
```



3. Grafo simple no dirigido reflexivo

?

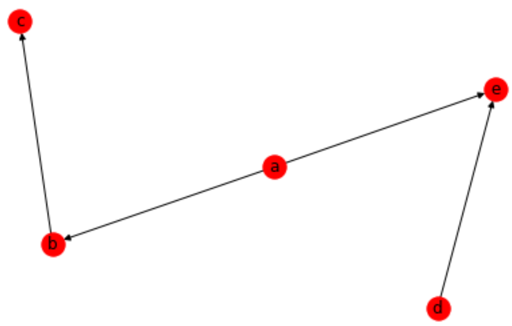
```python
import networkx as nx
import matplotlib.pyplot as plt

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e")]
reflexivo = [("a", "a"), ("b", "b"), ("c", "c"), ("d", "d"), ("e", "e")]

G = nx.Graph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)
G.add_edges_from(reflexivo)

nx.draw(G, with_labels = True)
plt.show()
```



4. Grafo simple dirigido acíclico

Las tuberías de agua y drenaje se puede representar ya que tienen un flujo hacia una dirección.

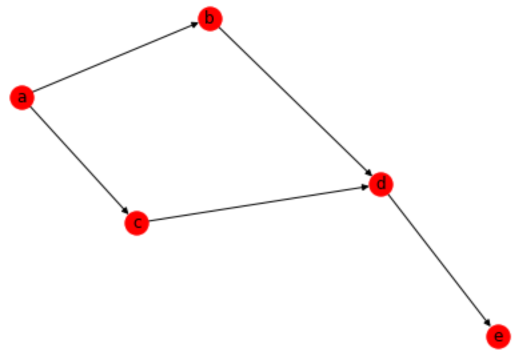
```
import networkx as nx
import matplotlib.pyplot as plt

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "c"), ("a", "e"), ("d", "e")]

G = nx.DiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)

nx.draw(G, with_labels = True)
plt.show()
```



5. Grafo simple dirigido cíclico

Como ejemplo aplicado se toma la pista de carreras de vehículos donde tiene un circuito.

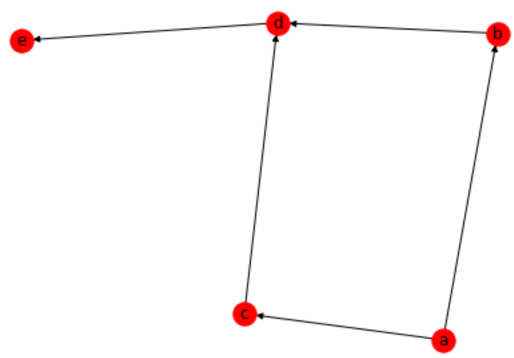
```
import networkx as nx
import matplotlib.pyplot as plt

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e")]

G = nx.DiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)

nx.draw(G, with_labels = True)
plt.show()
```



6. Grafo simple dirigido reflexivo

Las tuberías de petroleo, donde la reflexividad de los nodos es donde el petroleo se puede quedar como un punto de extracción.

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

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e")]
reflexivo = [("a", "a"), ("b", "b"), ("c", "c"), ("d", "d"), ("e", "e")]

G = nx.MutiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)
G.add_edges_from(reflexivo)

nx.draw(G, with_labels = True)
plt.show()
```

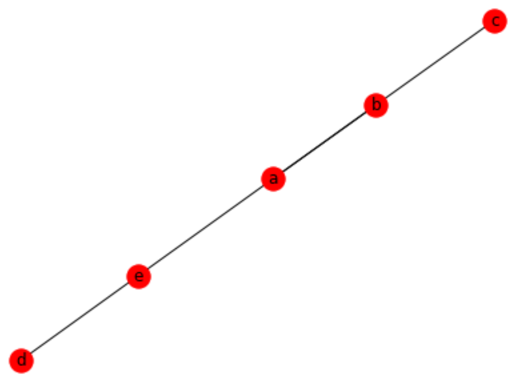


7. Multigrafo no dirigido acíclico

La red total de las tuberías de agua de drenaje representa un ejemplo para esta sección.

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

nodos = ["a", "b", "c", "d", "e"]
aristas = [("a", "b"), ("b", "a"), ("b", "c"), ("a", "e"), ("d", "e")]
```

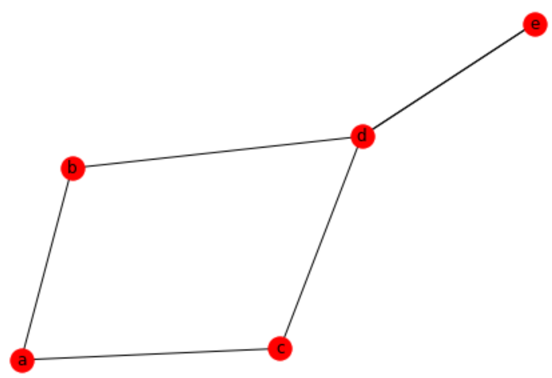
```python

G = nx.MultiDiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(aristas)

nx.draw(G, with_labels = True)
plt.show()
```



8. Multigrafo no dirigido cíclico

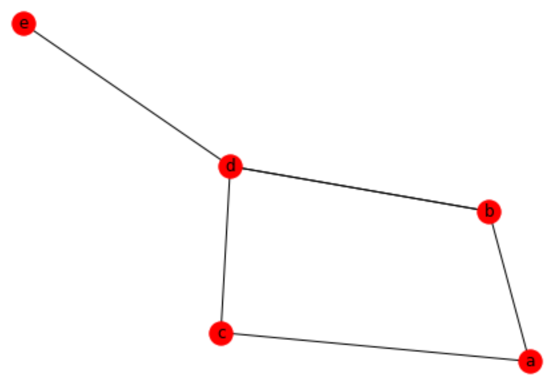
La red de autopistas de una ciudad.

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

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e"), ("e", "d")]

G = nx.MultiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)

nx.draw(G, with_labels = True)
plt.show()
```



9. Multigrafo no dirigido reflexivo

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

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e"), ("d", "b
    ")]
reflexivo = [("a", "a"), ("b", "b"), ("c", "c"), ("d", "d"), ("e", "e")]

G = nx.MultiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)
G.add_edges_from(reflexivo)

nx.draw(G, with_labels = True)
plt.show()
```

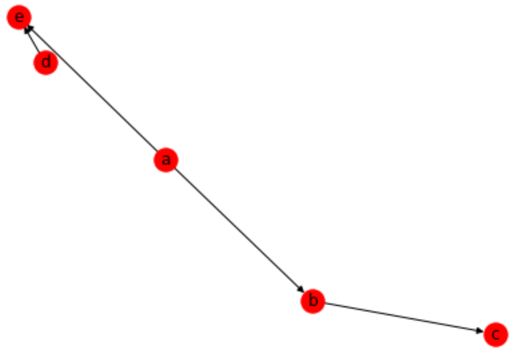


10. Multigrafo dirigido acíclico

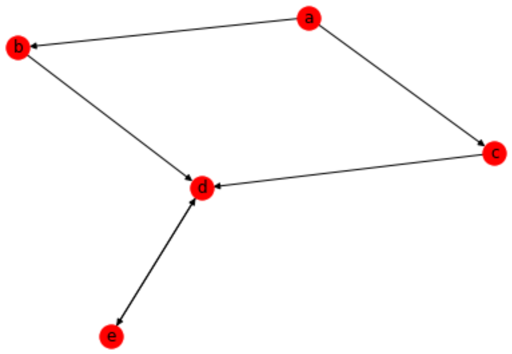
```
import networkx as nx
import matplotlib.pyplot as plt

nodos = ["a", "b", "c", "d", "e"]
aristas = [("a", "b"), ("b", "c"), ("a", "e"), ("d", "e")]

G = nx.MultiDiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(aristas)

nx.draw(G, with_labels = True)
plt.show()
```

11. Multigrafo dirigido cíclico

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

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e"), ("e", "d
    ")]

```

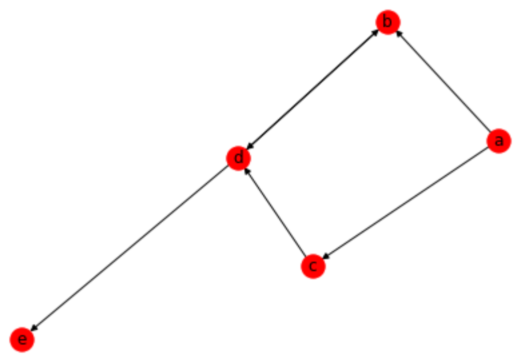


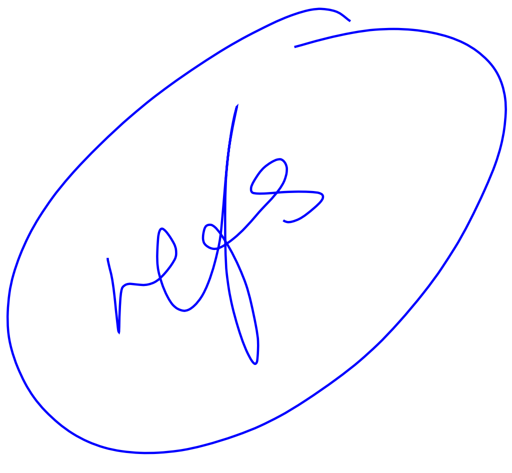

```
G = nx.MultiDiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)

nx.draw(G, with_labels = True)
plt.show()
```



12. Multigrafo dirigido reflexivo

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

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e"), ("d", "b")]
reflexivo = [("a", "a"), ("b", "b"), ("c", "c"), ("d", "d"), ("e", "e")]

G = nx.MultiDiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)
G.add_edges_from(reflexivo)

nx.draw(G, with_labels = True)
plt.show()
```


b
a
d
c
e



Optimización de flujo en redes

1455175: Angel Moreno
Tarea #1

1 de junio de 2019

Resumen

En esta tarea se hicieron correciones de ortografía, ejemplos aplicados de algunos grafos que no fueron añadidos, descripciones de las figuras (`caption`) y se añadieron las referencias bibliográficas.

1. Grafo simple no dirigido acíclico

Este primer grafo es el más sencillo, como aplicaciíon de este grafo es la representacion de ciudades ó localidades (nodos) conectadas mediante rutas de caminos (aristas) para llegar a otra ciudad.

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

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "c"), ("a", "e"), ("d", "e")]

G = nx.Graph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)

nx.draw(G, with_labels = True)
plt.show()
```

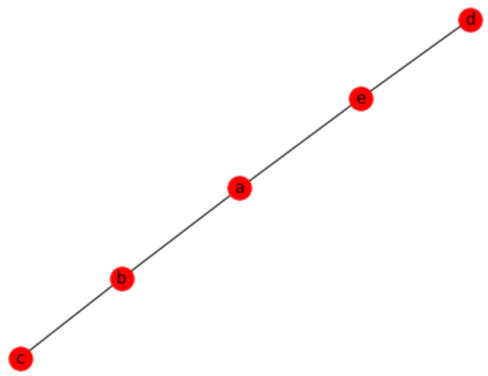


Figura 1: Grafo simple no dirigido acíclico.

2. Grafo simple no dirigido cíclico

Los puntos de paradas de los sistemas de transporte o lineas del metro (nodos) con las rutas o caminos (aristas) que unen cada punto de parada forma un grafo cíclico.

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

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e")]

G = nx.Graph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)

nx.draw(G, with_labels = True)
plt.show()
```

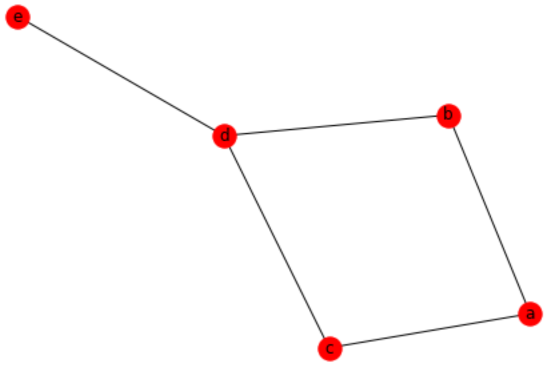


Figura 2: Grafo simple no dirigido cíclico.

3. Grafo simple no dirigido reflexivo

Este tipo de grafo tiene como característica que todos los nodos están conectados con sigo mismo, un ejemplo aplicado es la representación de circuitos eléctricos conectados unos con otros, donde la conexión reflexiva es que la carga se mantenga en el circuito.

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

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e")]
reflexivo = [("a", "a"), ("b", "b"), ("c", "c"), ("d", "d"), ("e", "e")]

G = nx.Graph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)
G.add_edges_from(reflexivo)

nx.draw(G, with_labels = True)
plt.show()
```

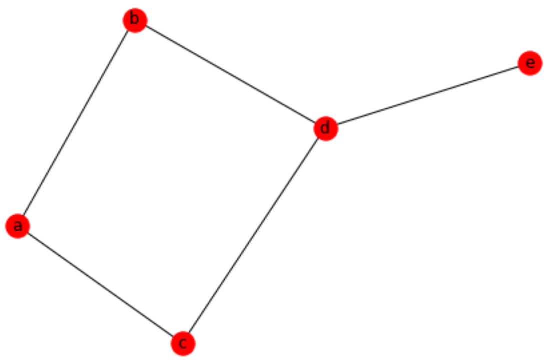


Figura 3: Grafo simple no dirigido reflexivo.

4. Grafo simple dirigido acíclico

Las tuberías de agua y drenaje se puede representar ya que tienen un flujo hacia una dirección.

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

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "c"), ("a", "e"), ("d", "e")]

G = nx.DiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)

nx.draw(G, with_labels = True)
plt.show()
```

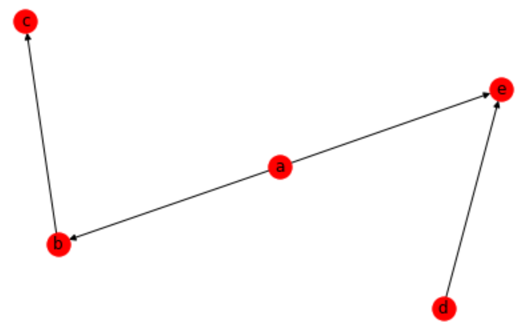


Figura 4: Grafo simple dirigido acíclico.

5. Grafo simple dirigido cíclico

Como ejemplo aplicado se toma la pista de carreras de vehículos donde tiene un circuito.

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

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e")]

G = nx.DiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)

nx.draw(G, with_labels = True)
plt.show()
```

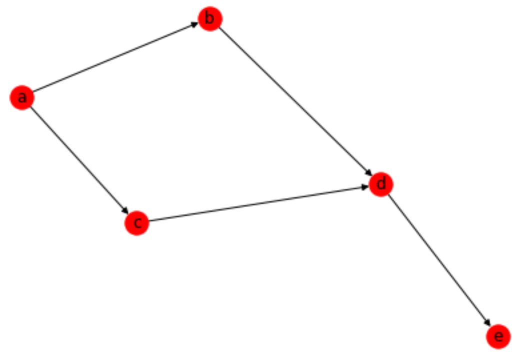


Figura 5: Grafo simple dirigido cíclico.

6. Grafo simple dirigido reflexivo

Las tuberías de petroleo, donde la reflexividad de los nodos es donde el petroleo se puede quedar como un punto de extracción.

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

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e")]
reflexivo = [("a", "a"), ("b", "b"), ("c", "c"), ("d", "d"), ("e", "e")]

G = nx.MutiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)
G.add_edges_from(reflexivo)

nx.draw(G, with_labels = True)
plt.show()
```

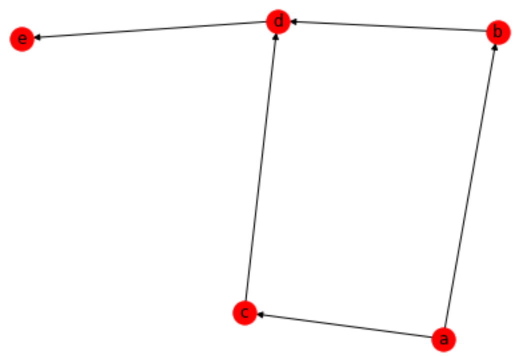


Figura 6: Grafo simple dirigido reflexivo.

7. Multigrafo no dirigido acíclico

En la ciudad se puede encontrar múltiples formas de llegar de un punto a otro, ya que existen múltiples caminos esto formar un multigrafo.

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

nodos = ["a", "b", "c", "d", "e"]
aristas = [("a", "b"), ("b", "a"), ("b", "c"), ("a", "e"), ("d", "e")]

G = nx.MultiDiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(aristas)

nx.draw(G, with_labels = True)
plt.show()
```

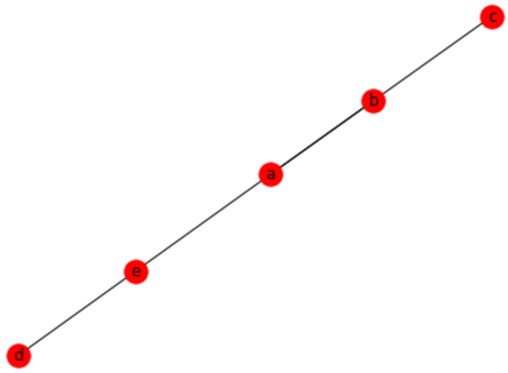


Figura 7: Multigrafo no dirigido acíclico.

8. Multigrafo no dirigido cíclico

La red de autopistas de una ciudad.

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

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e"), ("e", "d")]

G = nx.MultiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)

nx.draw(G, with_labels = True)
plt.show()
```

9. Multigrafo no dirigido reflexivo

Este tipo de grafo lo podemos aplicar en la red de mensajes donde uno mismo se puede enviar correo y las distintas plataformas de correos electrónicos son los disntintos caminos a usar.

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

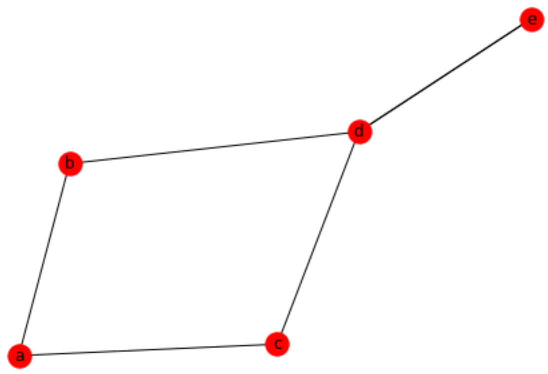


Figura 8: Multigrafo no dirigido cíclico.

```

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e"), ("d", "b
    ")]
reflexivo = [("a", "a"), ("b", "b"), ("c", "c"), ("d", "d"), ("e", "e")]

G = nx.MultiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)
G.add_edges_from(reflexivo)

nx.draw(G, with_labels = True)
plt.show()
```

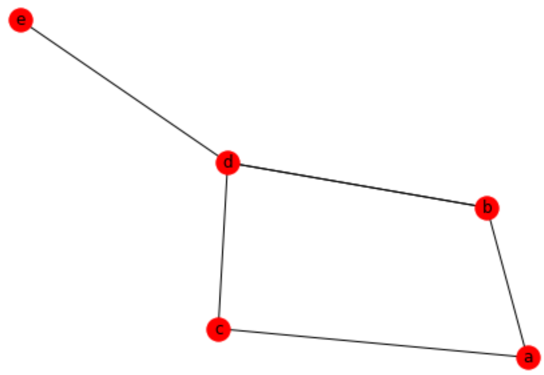


Figura 9: Multigrafo no dirigido reflexivo.

10. Multigrafo dirigido acíclico

En los videojuegos para terminar el juego se tiene que pasar los niveles para se puede terminar el nivel de muchas formas distintas, por tanto tiene múltiples caminos. Se crea una dirección y sin retorno del nivel.

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

nodos = ["a", "b", "c", "d", "e"]
```

```python
aristas = [("a", "b"), ("b", "c"), ("a", "e"), ("d", "e")]

G = nx.MultiDiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(aristas)

nx.draw(G, with_labels = True)
plt.show()
```

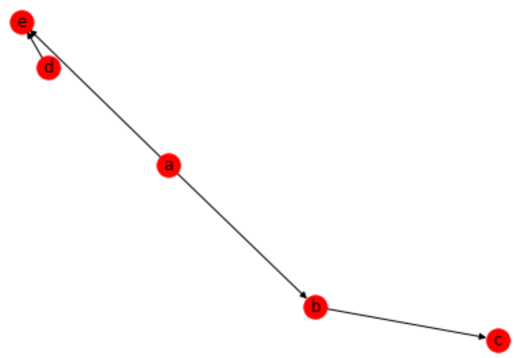


Figura 10: Multigrafo dirigido acíclico.

11. Multigrafo dirigido cíclico

Las distintas formas de estar conectado con un persona proporciona multiples caminos de comunicacion y los amigos en comun hacen que se formen los ciclos.

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

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e"), ("e", "d")]

G = nx.MultiDiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)

nx.draw(G, with_labels = True)
plt.show()
```

12. Multigrafo dirigido reflexivo

La paginas web tienen muchas formas de conectarse unas con otras y ademas multiples hipervinculos hacia una misma pagina. Este ejemplo es una aplicacion de un multigrafo dirigido reflexivo.

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

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e"), ("d", "b")]
```

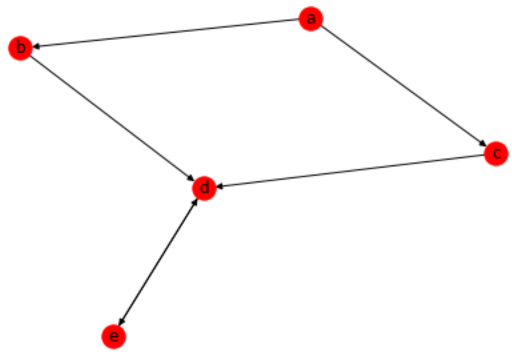


Figura 11: Multigrafo dirigido cíclico.

```
reflexivo = [("a", "a"), ("b", "b"), ("c", "c"), ("d", "d"), ("e", "e")]

G = nx.MultiDiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)
G.add_edges_from(reflexivo)

nx.draw(G, with_labels = True)
plt.show()
```

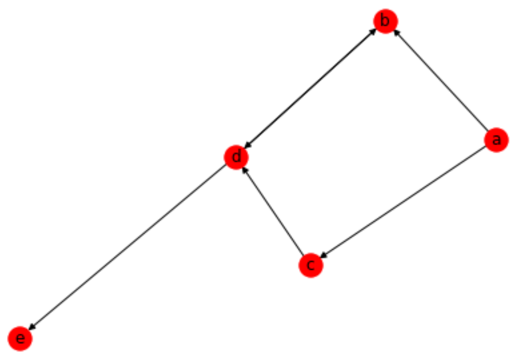


Figura 12: Multigrafo dirigido reflexivo.

Referencias

[1] SCHAEFFER E. *Optimización de flujo en redes, 2019.*
https://elisa.dyndns-web.com/teaching/opt/flow/

[2] TOMIHISA KAMADA, SATORU KAWAI. *An Algorithm for Drawing General Undirected Graphs*
Information Processing Letters, 1988.

[3] SAUS L. *Repository of Github, 2018.*
https://github.com/pejli/simulacion

Optimización de flujo en redes

6

5 175

Tarea #2

26 de febrero de 2019

1. Acomodo bipartito

Este tipo de acomodo divide el grafo en dos conjuntos ajenos, luego une los dos conjuntos con las aristas correspondientes, muy util para observar relacion entre elementos de los conjuntos. En seguida el codigo utilizando dicho acomodo y la figura 1 muestra el grafo.

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

G = nx.Graph()

G.add_nodes_from(['A', 'B', 'C'], bipartite = 0)
G.add_nodes_from(['D', 'E', 'F'], bipartite = 1)

G.add_edges_from([('A', 'D'), ('A', 'E')])
G.add_edges_from([('B', 'E')])
G.add_edges_from([('C', 'D'), ('C', 'F')])

nx.draw(G, pos = nx.bipartite_layout(G, ['A', 'B', 'C']), with_labels = True)

plt.show()
```

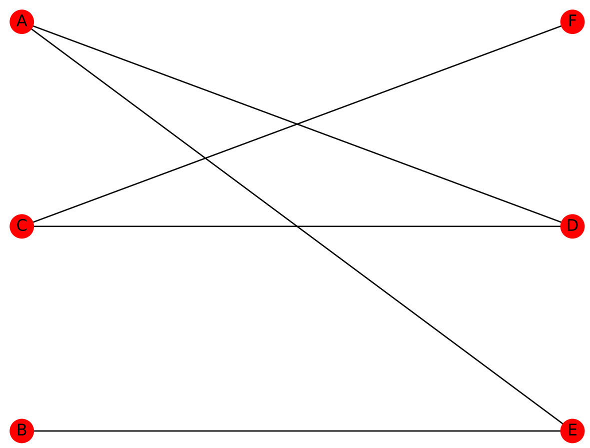


Figura 1: Grafo bipartito.

2. Acomodo circular

Los nodos son acomodados de tal forma para crear una circunferencia al unir los nodos. La figura 2 muestra un ejemplo del acomodo circular.

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

G=nx.Graph()
G.add_nodes_from(['A', 'B', 'C', 'D', 'E', 'F', 'G'])

G.add_edges_from([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'), ('E', 'F'), ('F', 'G'), ('G', 'A')])
G.add_edges_from([('A', 'D'), ('A', 'G')])
G.add_edges_from([('E', 'C'), ('E', 'G')])


nx.draw(G, pos = nx.circular_layout(G), with_labels=True)

plt.show()
```

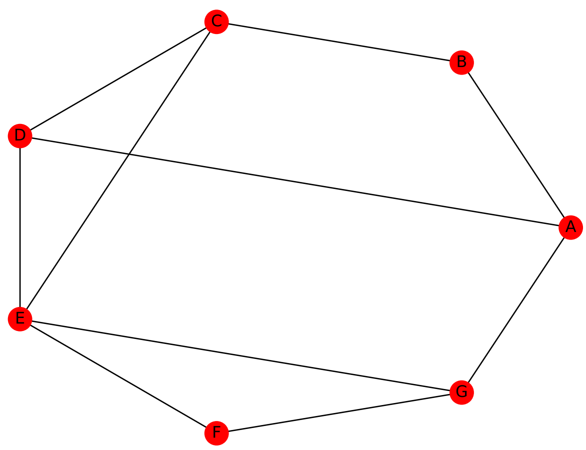


Figura 2: Ejemplo de acomodo circular de un grafo.

3. Acomodo kamada-kawai

Este acomodo hace que la distancia teórica del gráfico entre vértices en un gráfico está relacionada con la distancia geométrica entre ellos en el dibujo. El algoritmo tiene muchas buenas propiedades; dibujos simétricos, un número relativamente pequeño de cruces de bordes y dibujos casi congruentes de gráficos isomorfos. La figura 3 lo demuestra.

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

G=nx.Graph()
G.add_nodes_from(['A', 'B', 'C', 'D', 'E', 'F', 'G'])

G.add_edges_from([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'), ('E', 'F'), ('F', 'G'), ('G', 'A')])
G.add_edges_from([('A', 'D'), ('A', 'G')])
G.add_edges_from([('E', 'C'), ('E', 'G')])

nx.draw(G, pos = nx.kamada_kawai_layout(G), with_labels=True)

plt.show()
```

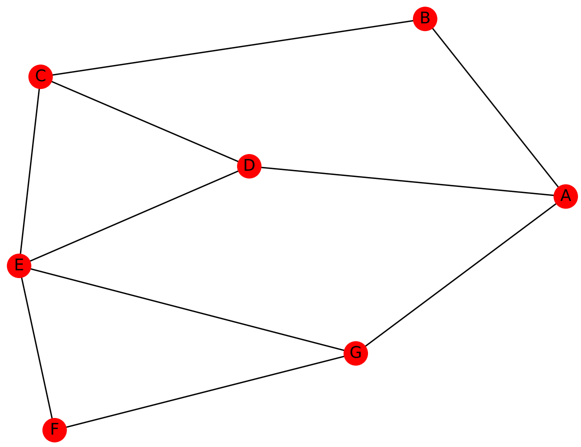


Figura 3: Ejemplo del acomodo kamada-kawai.

4. Acomodo aleatorio

El acomodo de los nodos es aleatoria, en base a una distribucion dada. Un ejemplo se muestra en la figura 4.

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

G=nx.DiGraph()

G.add_node("A")
G.add_nodes_from(["B", "C", "D", "E"])

G.add_edges_from([("A","B"), ("A","C"), ("A","D"), ("A","E")])

nx.draw(G, pos = nx.random_layout(G), with_labels = True)

plt.show()
```

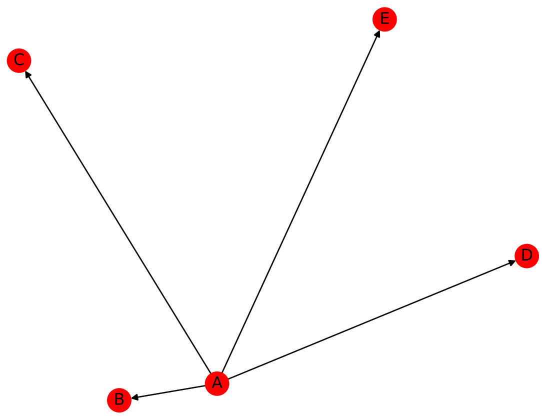


Figura 4: Ejemplo de acomodo aleatorio de un grafo.

5. Acomodo de cascara

Los nodos son acomodados de tal forma que se encuentren concentricos (como un cascaron). Se muestra un ejemplo en la figura 5.

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

G = nx.DiGraph()

G.add_node("A")
G.add_nodes_from(["B", "C", "D", "E", "F"])
G.add_nodes_from(["G", "H", "I", "J", "K", "L"])

G.add_path(["B", "C", "D", "E", "F", "B"])
G.add_path(["G", "H", "I", "J", "K", "L", "G"])

G.add_edges_from([("H", "C"), ("J", "E"), ("D", "A"), ("F", "A")])

nx.draw(G, pos = nx.shell_layout(G), with_labels=True)

plt.show()
```

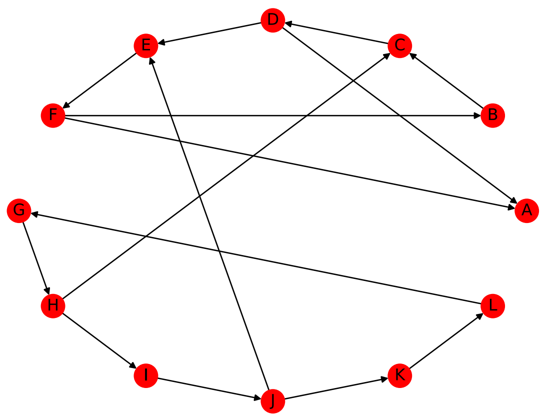


Figura 5: Grafo dibujado con el acomodo de cascara.

6. Acomodo espectral

El acomodo espectral coloca los nodos del gráfico en función de los vectores propios del gráfico Laplaciano. La figura 6 muestra un ejemplo.

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

M = nx.MultiGraph()

M.add_nodes_from(['X','Y','Z'])
M.add_edges_from([('X','Y'),('X','Y'),('Y','Z')])

nx.draw_spectral(M, with_labels=True)
plt.show()
```

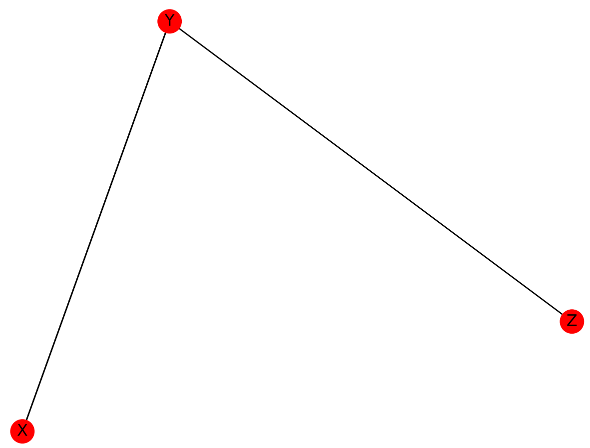


Figura 6: Ejemplo de acomodo espectral.

7. Acomodo de verano

Introduce los nodos utilizando el algoritmo dirigido por fuerza de Fruchterman-Reingold. En la figura 7 se observa el ejemplo.

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

G = nx.DiGraph()

G.add_edges_from([("A", "B", {'myweight':1}), ("B", "C", {'myweight':2}),
                  ("A", "D", {'myweight':3}), ("B", "D", {'myweight':10})])

nx.draw_networkx(G, nx.spring_layout(G, weight = "myweight"))

plt.axis('off')
plt.show()
```

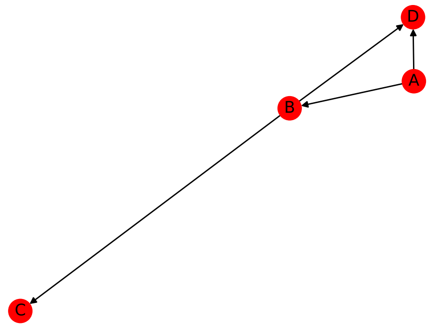


Figura 7: Ejemplo de acomodo de verano.

Referencias

[1] Schaeffer E. *Optimización de flujo en redes, 2019.*
`https://elisa.dyndns-web.com/teaching/opt/flow/`

[2] Tomihisa Kamada, Satoru Kawai. *An Algorithm for Drawing General Undirected Graphs*
`Information Processing Letters, 1988.`

Optimización de flujo en redes

175
Tarea #2

2 de junio de 2019

Resumen

Las correcciones hechas fueron faltas de ortografia y se añadieron 5 gráficas faltantes de grafos de la tarea anterior junto con códigos actualizados.

1. Acomodo bipartito

Este tipo de acomodo divide el grafo en dos conjuntos ajenos, luego une los dos conjuntos con las aristas correspondientes, muy util para observar relacion entre elementos de los conjuntos. En seguida el codigo utilizando dicho acomodo y la figura 1 muestra el grafo.

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

G = nx.Graph()

G.add_nodes_from(['A', 'B', 'C'], bipartite = 0)
G.add_nodes_from(['D', 'E', 'F'], bipartite = 1)

G.add_edges_from([('A', 'D'), ('A', 'E')])
G.add_edges_from([('B', 'E')])
G.add_edges_from([('C', 'D'), ('C', 'F')])

nx.draw(G, pos = nx.drawing.layout.bipartite_layout(G, ['A', 'B', 'C']),
    with_labels = True)
plt.savefig('bipartite_layout.eps')

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e"), ("d", "b
    ")]
reflexivo = [("a", "a"), ("b", "b"), ("c", "c"), ("d", "d"), ("e", "e")]

G = nx.MultiDiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)
G.add_edges_from(reflexivo)

nx.draw(G, pos = nx.drawing.layout.bipartite_layout(G, ['a', 'b', 'c']),
    with_labels = True)
plt.savefig('bipartite_layou2.eps')
```

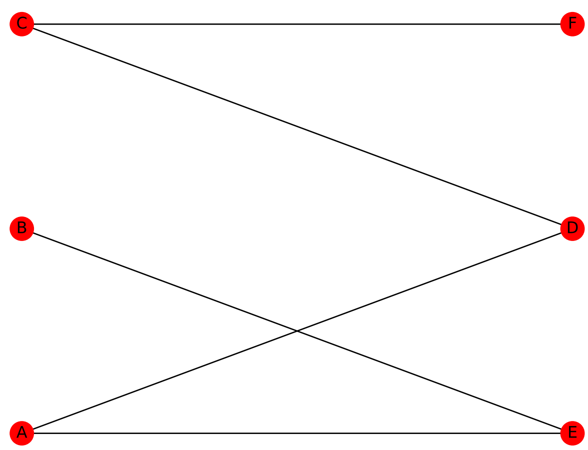


Figura 1: Grafo bipartito.

2. Acomodo circular

Los nodos son acomodados de tal forma para crear una circunferencia al unir los nodos. La figura 4 muestra un ejemplo del acomodo circular.

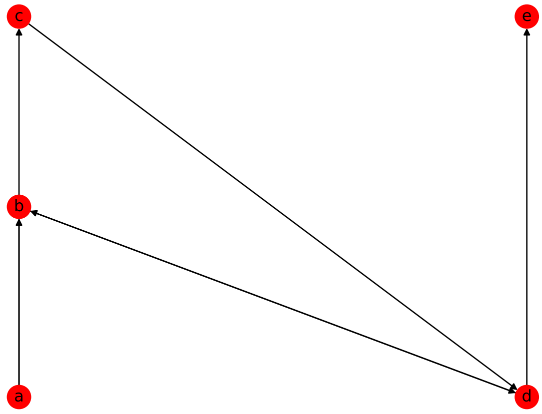


Figura 2: Grafo bipartito.

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

G=nx.Graph()
G.add_nodes_from(['A', 'B', 'C', 'D', 'E', 'F', 'G'])

G.add_edges_from([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'), ('E', 'F'), ('
    F', 'G'), ('G', 'A')])
G.add_edges_from([('A', 'D'), ('A', 'G')])
G.add_edges_from([('E', 'C'), ('E', 'G')])


nx.draw(G, pos = nx.circular_layout(G), with_labels=True)

plt.savefig('circular_layout.eps')

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e"), ("d", "b
    ")]
reflexivo = [("a", "a"), ("b", "b"), ("c", "c"), ("d", "d"), ("e", "e")]

G = nx.MultiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)
G.add_edges_from(reflexivo)

nx.draw(G, pos = nx.circular_layout(G), with_labels = True)

plt.savefig('circular_layout2.eps')
```

3. Acomodo Kamada-Kawai

Este acomodo hace que la distancia teórica del gráfico entre vértices en un gráfico está relacionada con la distancia geométrica entre ellos en el dibujo. El algoritmo tiene muchas buenas propiedades; dibujos simétricos, un número relativamente pequeño de cruces de bordes y dibujos casi congruentes de gráficos isomorfos. La figura 6 lo demuestra.

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

```

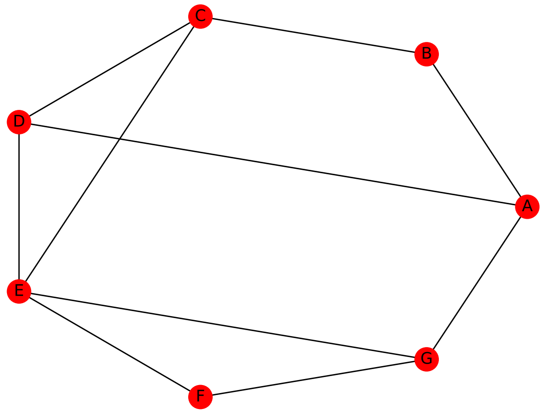


Figura 3: Ejemplo de acomodo circular de un grafo.

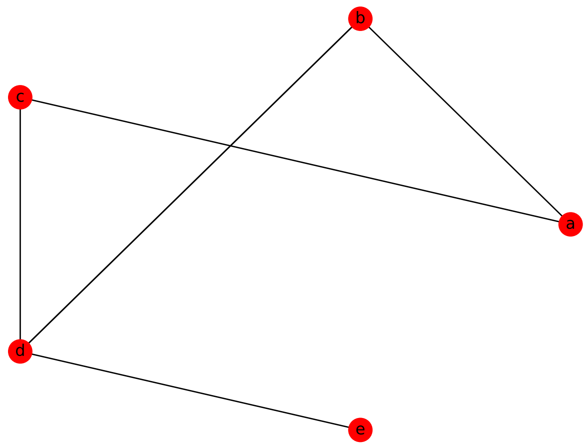


Figura 4: Ejemplo de acomodo circular de un grafo.

```python
G=nx.Graph()
G.add_nodes_from(['A', 'B', 'C', 'D', 'E', 'F', 'G'])

G.add_edges_from([('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'), ('E', 'F'), ('F', 'G'), ('G', 'A')])
G.add_edges_from([('A', 'D'), ('A', 'G')])
G.add_edges_from([('E', 'C'), ('E', 'G')])

nx.draw(G, pos = nx.kamada_kawai_layout(G), with_labels=True)

plt.savefig('kamada_kawai_layout.eps')

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e"), ("d", "b")]
reflexivo = [("a", "a"), ("b", "b"), ("c", "c"), ("d", "d"), ("e", "e")]

G = nx.MultiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)
G.add_edges_from(reflexivo)

nx.draw(G, pos = nx.kamada_kawai_layout(G), with_labels = True)

```

```
plt.savefig('kamada_kawai_layout2.eps')
```

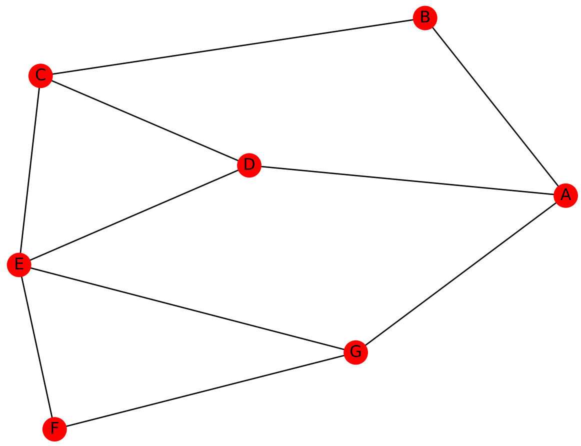


Figura 5: Ejemplo del acomodo Kamada-Kawai.

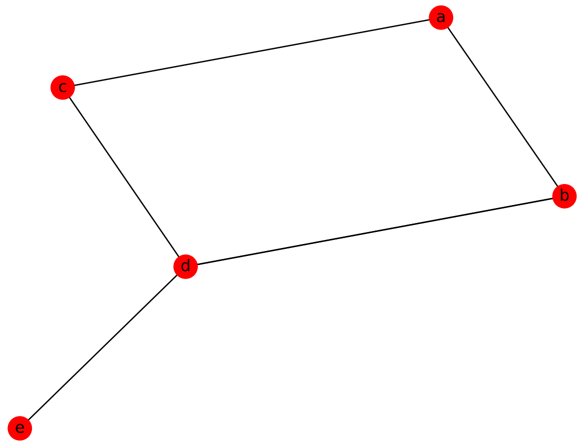


Figura 6: Ejemplo del acomodo Kamada-Kawai.

4. Acomodo aleatorio

El acomodo de los nodos es aleatorio, en base a una distribución dada. Un ejemplo se muestra en la figura 8.

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

G=nx.DiGraph()

G.add_node("A")
G.add_nodes_from(["B", "C", "D", "E"])

G.add_edges_from([("A","B"), ("A","C"), ("A","D"), ("A","E")])

nx.draw(G, pos = nx.random_layout(G), with_labels = True)

nodos = ["a", "b", "c", "d", "e"]
aristas = [("a", "b"), ("b", "c"), ("a", "e"), ("d", "e")]

G = nx.MultiDiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(aristas)

nx.draw(G, pos = nx.random_layout(G),  with_labels = True)

plt.savefig('random_layout2.eps')
```

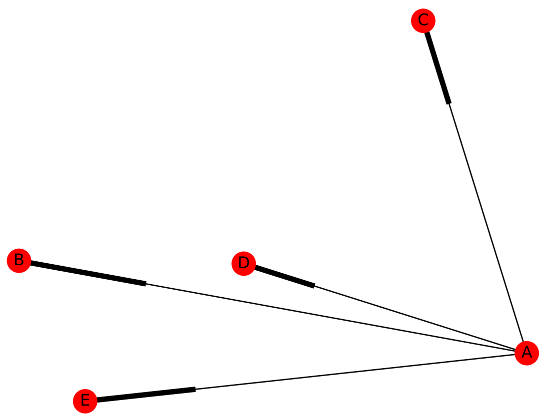


Figura 7: Ejemplo de acomodo aleatorio de un grafo.

5. Acomodo de cáscara

Los nodos son acomodados de tal forma que se encuentren concéntricos (como un cascaron). Se muestra un ejemplo enla figura 10.

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

G = nx.DiGraph()

G.add_node("A")
G.add_nodes_from(["B", "C", "D", "E", "F"])
```

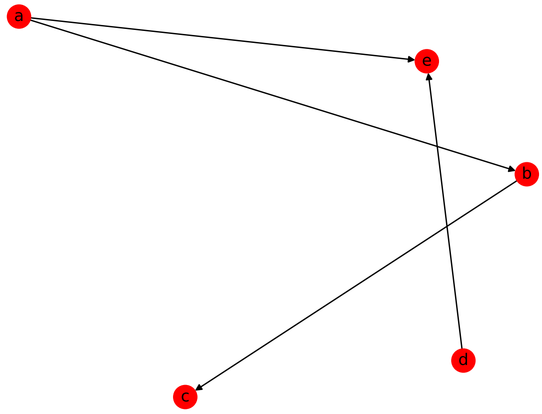


Figura 8: Ejemplo de acomodo aleatorio de un grafo.

```
G.add_nodes_from(["G", "H", "I", "J", "K", "L"])

G.add_path(["B", "C", "D", "E", "F", "B"])
G.add_path(["G", "H", "I", "J", "K", "L", "G"])

G.add_edges_from([("H", "C"), ("J", "E"), ("D", "A"), ("F", "A")])

nx.draw(G, pos = nx.shell_layout(G), with_labels=True)

plt.savefig('shell_layout.eps')

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e"), ("e", "d
    ")]

G = nx.MultiDiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)

nx.draw(G, pos = nx.shell_layout(G), with_labels = True)

plt.savefig('shell_layout2.eps')
```

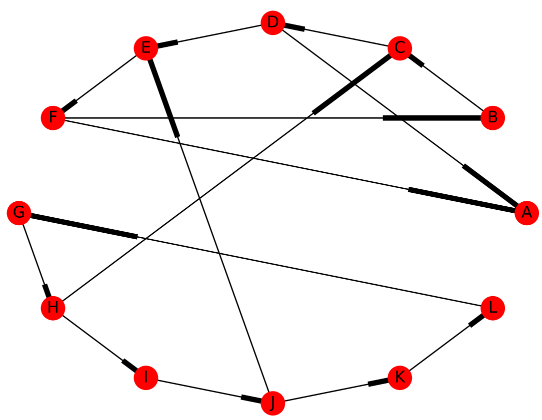


Figura 9: Grafo dibujado con el acomodo de cáscara.

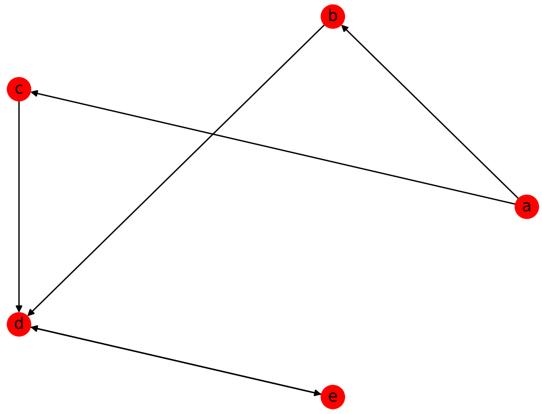


Figura 10: Grafo dibujado con el acomodo de cascara.

6. Acomodo espectral

El acomodo espectral coloca los nodos del gráfico en función de los vectores propios del gráfico Laplaciano. La figura 11 muestra un ejemplo.

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

M = nx.MultiGraph()

M.add_nodes_from(['X','Y','Z'])
M.add_edges_from([('X','Y'),('X','Y'),('Y','Z')])

nx.draw_spectral(M, with_labels=True)
plt.savefig('draw_spectral.eps')
```

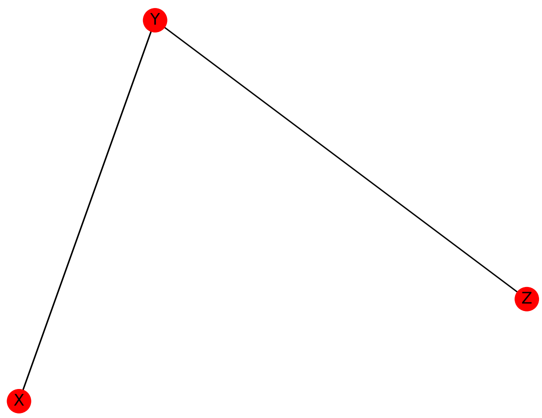


Figura 11: Ejemplo de acomodo espectral.

7. Acomodo de verano

Introduce los nodos utilizando el algoritmo dirigido por fuerza de Fruchterman-Reingold. En la figura 12 se observa el ejemplo.

```
import networkx as nx
```

```python
import matplotlib.pyplot as plt

G = nx.DiGraph()

G.add_edges_from([("A", "B", {'myweight':1}), ("B", "C", {'myweight':2}),
                  ("A", "D", {'myweight':3}), ("B", "D", {'myweight':10})])

nx.draw_networkx(G, nx.spring_layout(G, weight = "myweight"))

plt.axis('off')
plt.savefig('spring_layout.eps')

nodos = ["a", "b", "c", "d", "e"]
vertices = [("a", "b"), ("b", "d"), ("a", "c"), ("c", "d"), ("d", "e"), ("d", "b
    ")]
reflexivo = [("a", "a"), ("b", "b"), ("c", "c"), ("d", "d"), ("e", "e")]

G = nx.MultiDiGraph()
G.add_nodes_from(nodos)
G.add_edges_from(vertices)
G.add_edges_from(reflexivo)

nx.draw(G, with_labels = True)
```

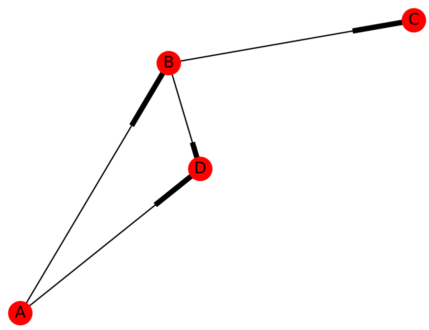


Figura 12: Ejemplo de acomodo de verano.

Referencias

[1] Schaeffer E. *Optimización de flujo en redes, 2019.*
`https://elisa.dyndns-web.com/teaching/opt/flow/`

[2] Kamada Tomihisa, Kawai Satoru. *An Algorithm for Drawing General Undirected Graphs*
`Information Processing Letters, 1988.`

[3] Saus L. *Repository of Github, 2018.*
`https://github.com/pejli/simulacion`

9

Optimización de flujo en redes

5175

Tarea #3

19 de marzo de 2019

Se utilizan algoritmos para grafos de NetworkX, implementando un código Python para ejecutar los ~~siguientes~~ algoritmos:

- `all_shortest_paths`
- `betweenness_centrality`
- `dfs_tree`
- `greedy_color`
- `max_weight_matching`

luego se aplica una cantidad moderada de iteraciones para crear tiempos de ejecución considerables para despues hacer un análisis estadístico.

1. Caminos cortos

Se utiliza el primer grafo de la tarea 2 con el algoritmo `all_shortest_paths` de NetworkX, el cual encuentra todos los caminos cortos de un nodo inicial a un nodos destino, se aplica un total de iteraciones de 8 millones para alcanzar un tiempo de ejecución mayor a 5 segundos como se muestra a continuación:

```
tiempoAngel1 = []
for i in range(30):
    start = tm.time()
    for x in range(8000000):
        nx.all_shortest_paths(G, source='C', target='B')
    end = tm.time()
    tiempoAngel1.append(end - start)
```

La figura 1 muestra los resultados obtenidos de la implementacion, se muestra la prueba estadistica de *shapiro wilk* para probar si la distribucion de los datos se comportan normales, dado que el p valor es mayor a 0.05 se puede concluir que los datos se comportan normales, ademas se muestra la media y desviación estandar.

```
tiempoAngel1 = []
for i in range(30):
    start = tm.time()
    for x in range(8000000):
        nx.all_shortest_paths(G, source='C', target='B')
    end = tm.time()
    tiempoAngel1.append(end - start)
```

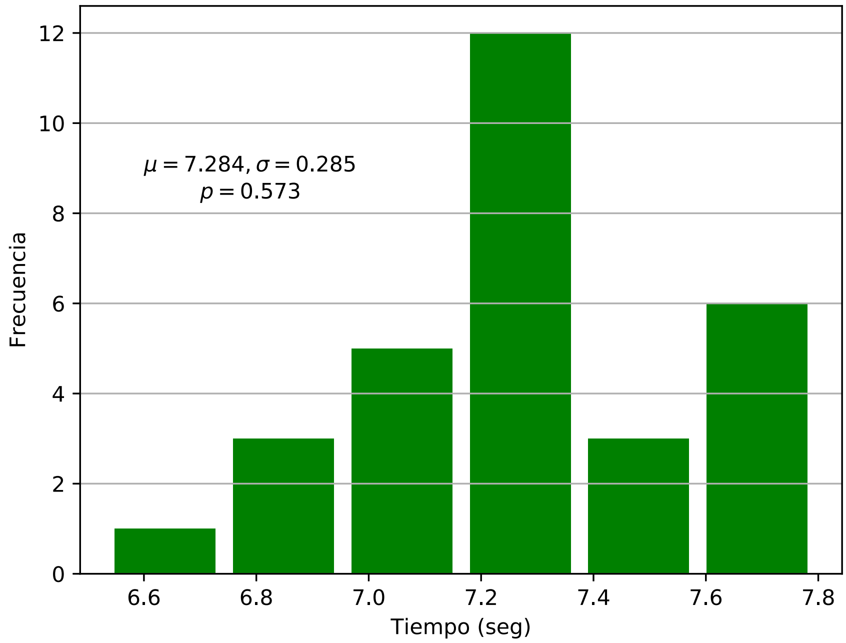


Figura 1: Histograma del tiempo de ejecución del algoritmo.

2. Nodos centrales

Se implementa subrutina similar a la seccion pasada con la diferencia de utilizar el algoritmo `betweenness_centrality`, el cual es perfecto para encontrar caminos cortos desde un nodo inicial central hasta uno final. Se utiliza el segundo grafo de la tarea 2.

La figura 2 muestra el histograma del tiempo de ejecion, se observa que los datos no se comportan normales debido al p valor muy pequeño de la prueba estadistica.

```
tiempoAngel2 = []
for i in range(30):
    start = tm.time()
    for x in range(20000):
        nx.betweenness_centrality(G, normalized=True)
    end = tm.time()
    tiempoAngel2.append(end - start)
```

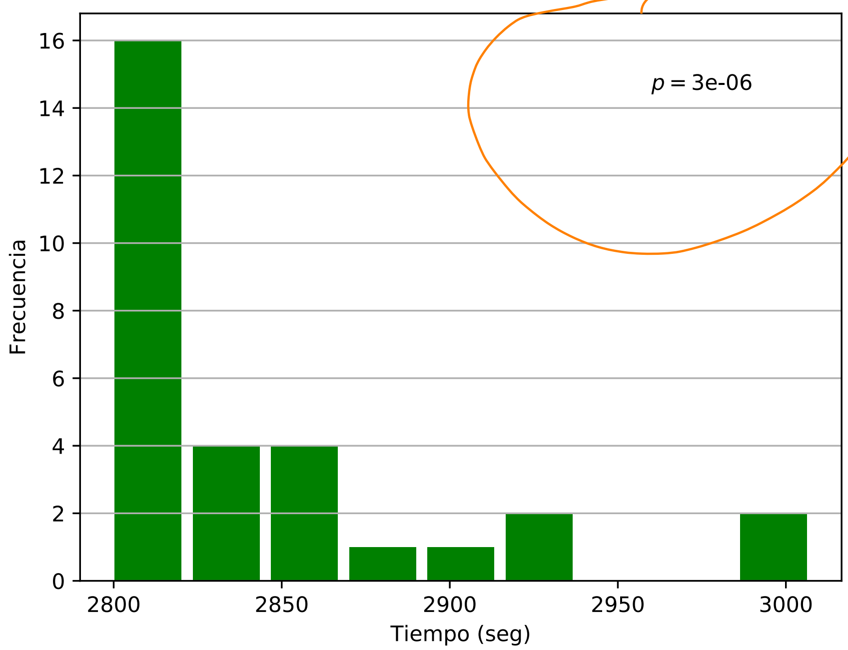


Figura 2: Histograma del tiempo de ejecución del algoritmo `betweenness_centrality`.

3. Busqueda en profundidad

El algoritmo `dfs_tree` construye un arbol orientado partiendo desde un nodo inicial utilizando la busqueda a profundidad. La figura 3 muestra el histograma, se vuelve a rechazar normalidad de los datos.

```
tiempoAngel3 = []
for i in range(30):
    start = tm.time()
    for x in range(100000):
        nx.dfs_tree(G, "D")
    end = tm.time()
    tiempoAngel3.append(end - start)
```

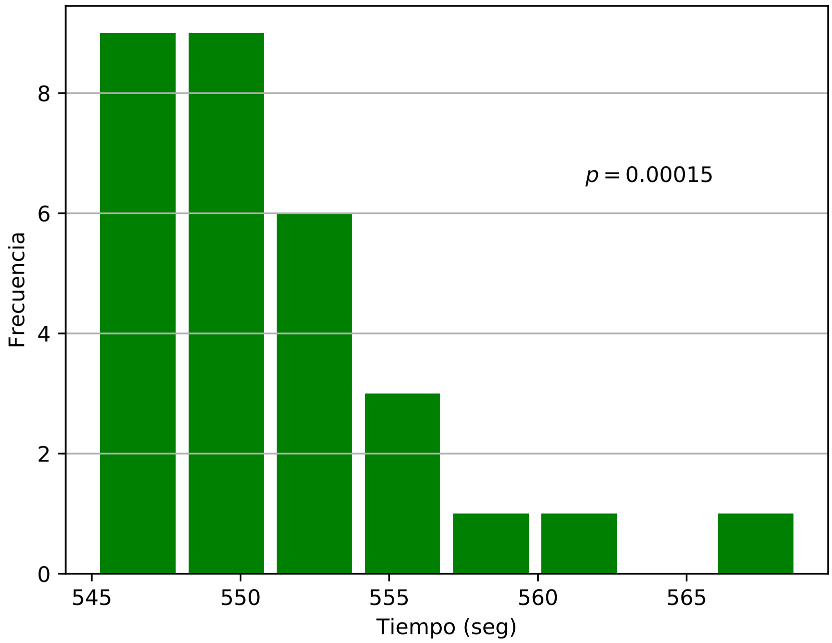


Figura 3: Histograma del tiempo de ejecución utilizando el algoritmo `dfs_tree`.

4. Colores

La figura 4 muestra el histograma del tiempo de ejecución de usar el algoritmo `greedy_color` el cual encuentra los colores adecuados para un grafo utlizando estrategias. Se rechaza normalidad de los datos.

```
tiempoAngel4 = []
for i in range(30):
    start = tm.time()
    for x in range(100000):
        nx.greedy_color(G, strategy='largest_first')
    end = tm.time()
    tiempoAngel4.append(end - start)
```

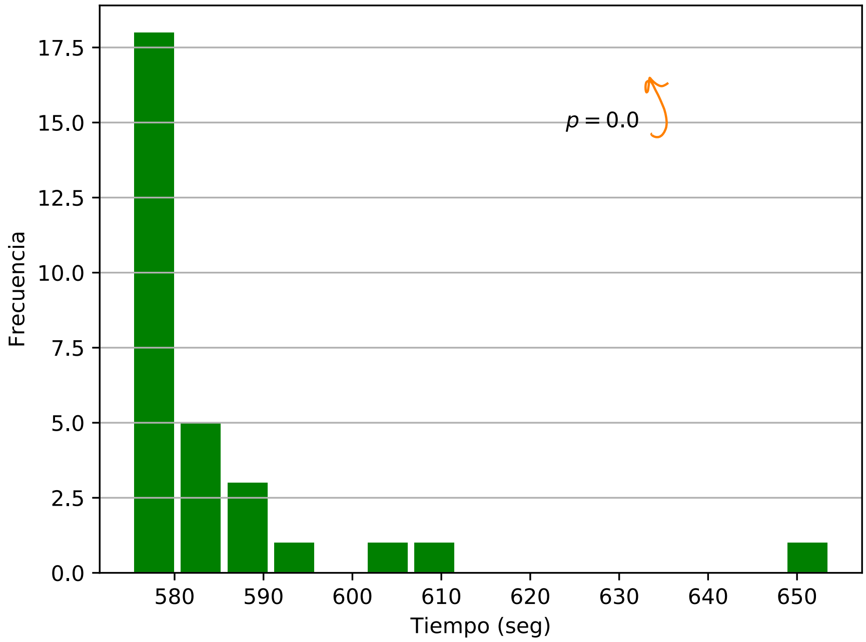


Figura 4: Tiempo de ejecución del algoritmo `greedy_color`.

5. Maximo peso

Utilizando el algoritmo `max_weight_matching` se consigue el histograma de la figura 5, el algoritmo consiste en encontrar el camino con el máximo peso partiendo desde un nodo inicial. Se rechaza normalidad de los datos.

```
tiempoAngel5 = []
for i in range(30):
    start = tm.time()
    for x in range(25000):
        nx.max_weight_matching(G)
    end = tm.time()
    tiempoAngel5.append(end - start)
```

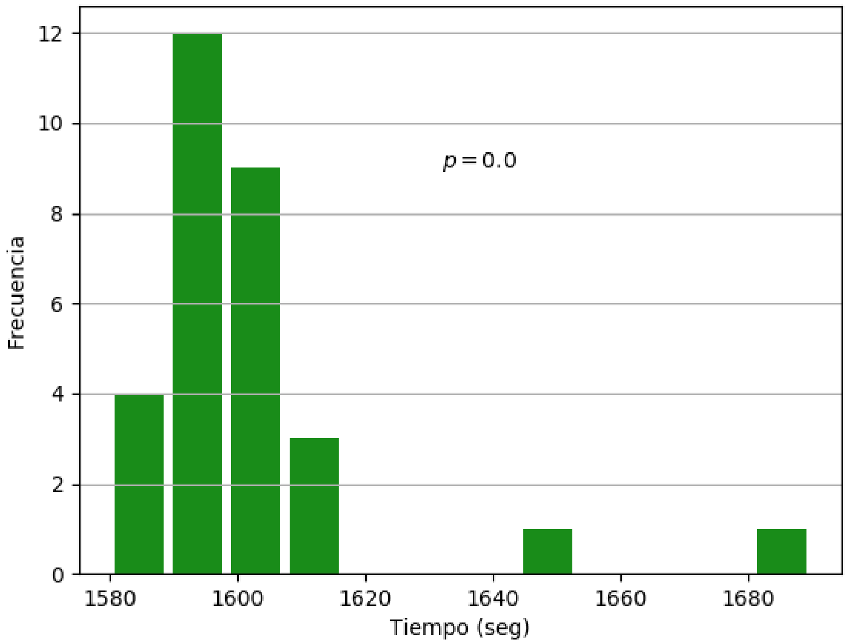


Figura 5: Histograma del tiempo de ejecución del algoritmo `max_weight_matching`.

6. Conclusiones

La figura 6 muestra una gráfica de dispersión de tiempo de ejecución contra cantidad de nodos de los grafos. Se observa como varian los tiempos dependiendo la cantidad de nodos en el grafo.

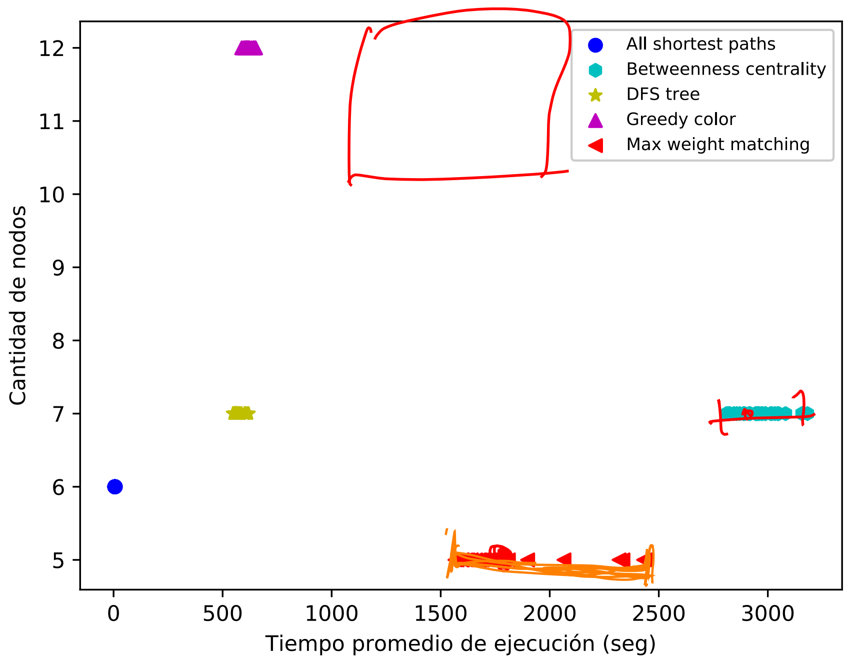


Figura 6: Gráfica de dispersión de tiempos contra cantidad de nodos del grafo.

La figura 7 muestra los resultados de los tiempos de ejecucion contra la cantidad de aristas del grafo.

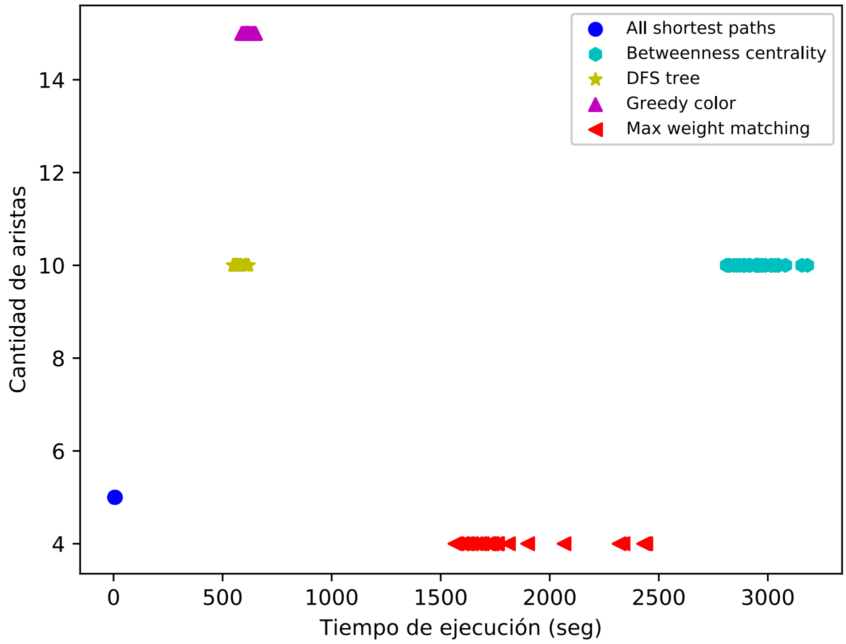


Figura 7: Gráfica de dispersión de tiempos contra cantidad de aristas del grafo.

Referencias

[1] Schaeffer E. *Optimización de flujo en redes, 2019.*
https://elisa.dyndns-web.com/teaching/opt/flow/

[2] Tomihisa Kamada, Satoru Kawai. *An Algorithm for Drawing General Undirected Graphs*
Information Processing Letters, 1988.

Optimización flujo en redes

8

5175: Ángel Moreno

Tarea # 4: Complejidad asintótica experimental

April 2, 2019

Se utiliza los siguients generados de grafo:

1. complete_graph
2. circulant_graph
3. wheel_graph

\texttt

para crear 10 grafos de dimensiones 32, 64, 128 y 256, ponderados con una distribución $\sim \mathcal{N}(10, 1)$. Para cada uno de los grafos generados se implementa los siguientes algoritmo s de flujo máximo:

1. maximum_flow
2. preflow_push
3. shortest_aumenting_path

\texttt

cinco cinco

se ejecutan con 5 diferentes fuentes y sumideros un total de 5 réplicas. Se toma el tiempo de ejecución de la implementación y se hace un análisis estadístico contra los diferentes factores cantidad de nodos, densidad del grafo, algoritmo de generación y algoritmo de flujo máximo.

1 Resultados

La figura 1 muestra la caja bigotes de generadores grafos contra tiempo de ejecución, se observa en el bloque del generador del grafo completos tiempo de ejecución son más altos que los demás algoritmo. La figura 2 muestra el gráfico de los algoritmos de flujo máximo contra el tiempo de ejecución, se observa que el tiempo de ejecución del algoritmo depende del grafo. La figura 3 muestra que el tiempo de ejecución aumenta según el orden del grafo y la figura 4 se observa que el entre mas densidad mas afecta ene le tiempo de ejecución.

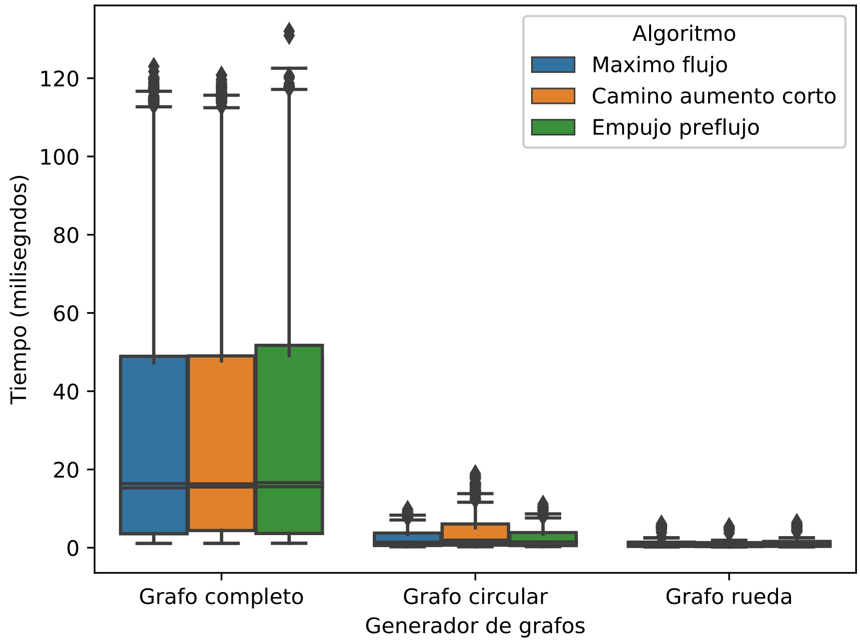


Figure 1: Efecto del geneador de grafos contra tiempo de ejecución.

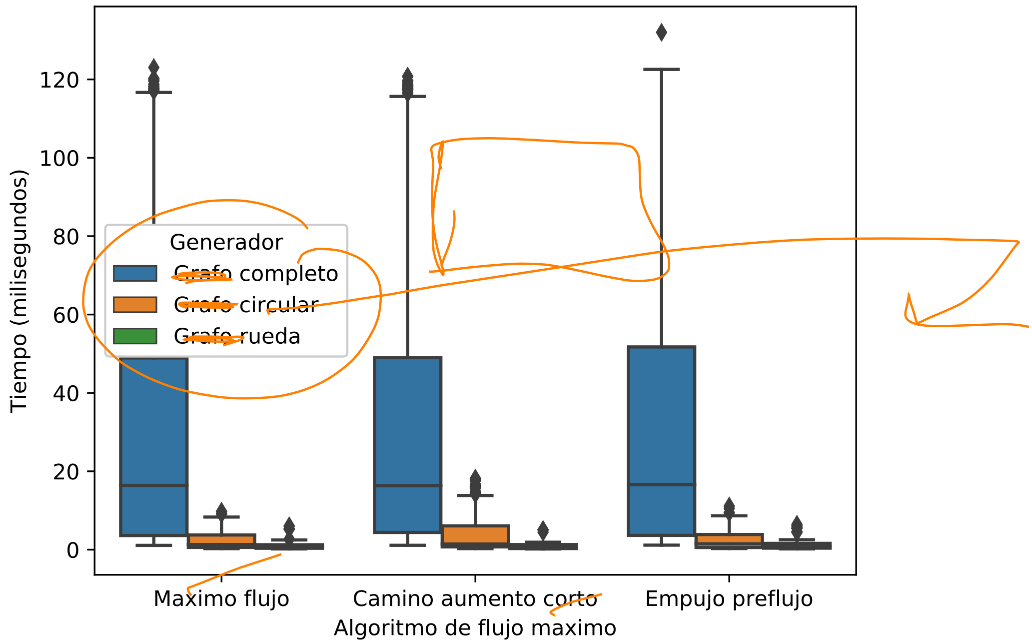


Figure 2: Efecto del algoritmo contra tiempo de ejecución.

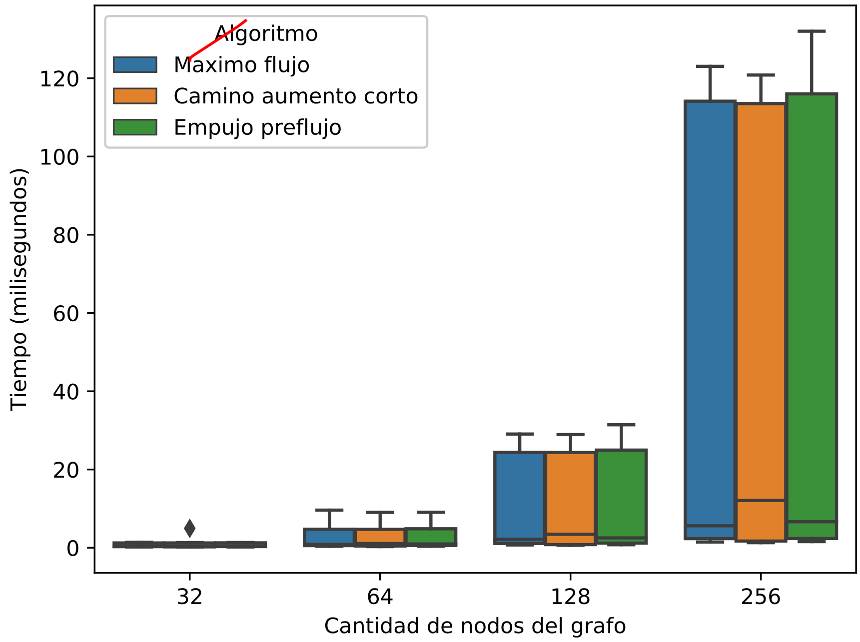
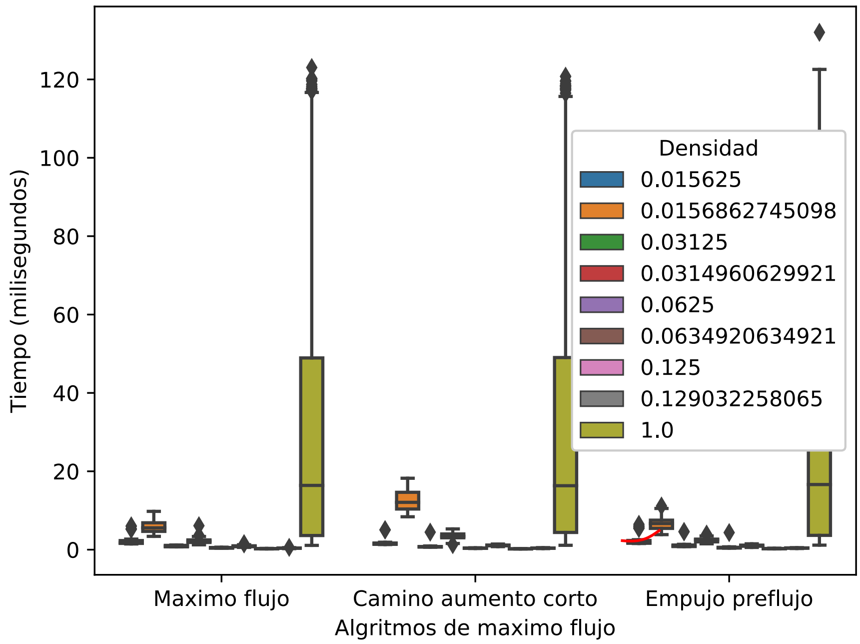


Figure 3: Efecto del orden del grafo contra el tiempo de ejecución.



0.01 – 0.03

0.03 – 0.04

0.06 – 0.07

0.1 – 0.2

1.0

Figure 4: Efecto de la densidad del grafo contra tiempo de ejecución.

$x\,0$

$\times 10^{-24}$

< 0.061

Factor	Cuadrados medios	GL	F	F_{GL}
Generador	4236.598736	2.0	56.139347	2.271267e-24
Algoritmo	103.331225	2.0	1.369246	2.545659e-01
Generador:Algoritmo	368.019017	4.0	2.438318	4.518991e-02
Orden	779151.982887	1.0	20649.150964	0.000000e+00
Generador:Orden	844.240482	2.0	11.187066	1.485138e-05
Orden:Algoritmo	49.606236	2.0	0.657334	5.183566e-01
Densidad	69967.332330	1.0	1854.280089	2.725925e-278
Generador:Densidad	399.506210	2.0	5.293873	5.101561e-03
Algoritmo:Densidad	73.938648	2.0	0.979764	3.756018e-01
Orden:Densidad	155.993981	1.0	4.134166	4.217371e-02
Residual	67239.996255	1782.0		

Table 1: Analisis de varianza de los factores.

El cuadro 1 muestra el análisis de varianza y se observa que el algoritmo, orden-algoritmo y densidad-algoritmo infuyen en el tiempo de ejecución, mientras que los demás factores no afecta el tiempo de ejecución.

References

[1] Schaeffer E. *Optimización de flujo en redes, 2019.*
`https://elisa.dyndns-web.com/teaching/opt/flow/`

[2] Kamada T. *An Algorithm for Drawing General Undirected Graphs*
`Information Processing Letters, 1988.`

[3] Saus L. *Repository of Github, 2019.*
`https://github.com/pejli`

Optimización flujo en redes

5175: Ángel Moreno

Tarea # 4: Complejidad asintótica experimental

2 de junio de 2019

Se utiliza los siguients generados de grafo:

1. `complete_graph`
2. `circulant_graph`
3. `wheel_graph`

para crear 10 grafos de dimensiones 32, 64, 128 y 256, ponderados con una distribución $\sim \mathcal{N}(10, 1)$. Para cada uno de los grafos generados se implementa los siguientes algoritmos de flujo máximo:

1. `maximum_flow`
2. `preflow_push`
3. `shortest_aumenting_path`

se ejecutan con 5 diferentes fuentes y sumideros un total de 5 réplicas. Se toma el tiempo de ejecución de la implementación y se hace un análisis estadístico contra los diferentes factores cantidad de nodos, densidad del grafo, algoritmo de generación y algoritmo de flujo máximo.

1. Resultados

La figura 1 muestra la caja bigotes de generadores grafos contra tiempo de ejecución, se observa en el bloque del generador del grafo completos tiempo de ejecución son más altos que los demás algoritmo. La figura 2 muestra el gráfico de los algoritmos de flujo máximo contra el tiempo de ejecución, se observa que el tiempo de ejecución del algoritmo depende del grafo. La figura 3 muestra que el tiempo de ejecución aumenta según el orden del grafo y la figura 4 se observa que el entre más densidad más afecta en el tiempo de ejecución.

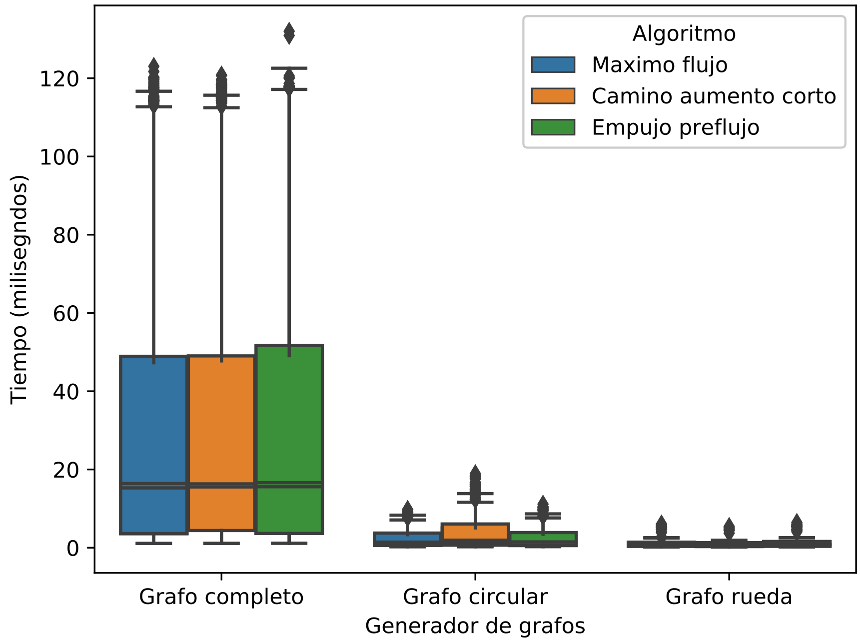


Figura 1: Efecto del geneador de grafos contra tiempo de ejecución.

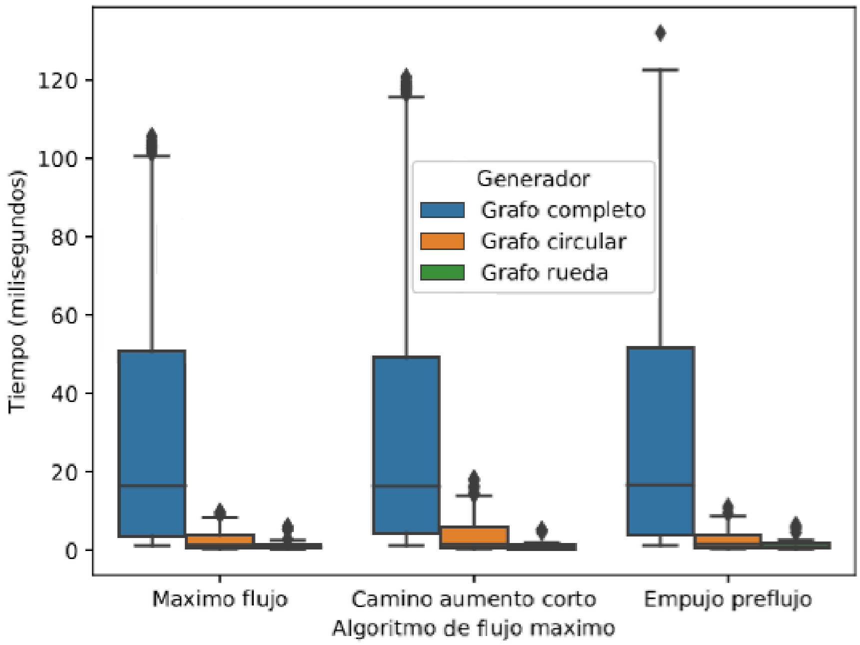


Figura 2: Efecto del algoritmo contra tiempo de ejecución.

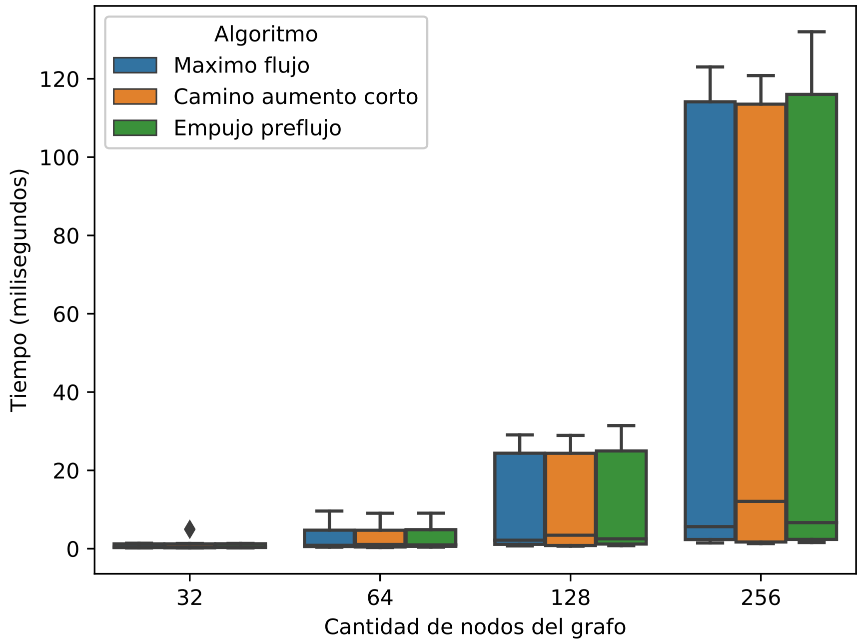


Figura 3: Efecto del orden del grafo contra el tiempo de ejecución.

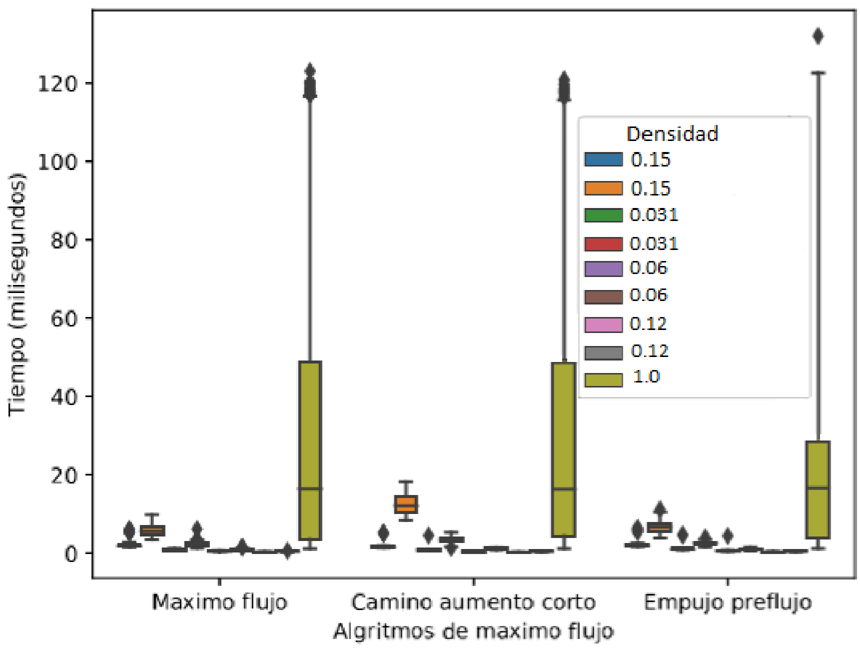


Figura 4: Efecto de la densidad del grafo contra tiempo de ejecución.

Factor	Cuadrados medios	GL	F	F_{GL}
Generador	4236.59	2	56.13	0
Algoritmo	103.33	2	1.36	0.254
Generador:Algoritmo	368.01	4	2.43	0.045
Orden	779151.98	1	20649.15	0
Generador:Orden	844.24	2	11.18	1.48e-05
Orden:Algoritmo	49.60	2	0.65	0.518
Densidad	69967.33	1	1854.28	0
Generador:Densidad	399.50	2	5.29	0.0051
Algoritmo:Densidad	73.93	2	0.97	0.375
Orden:Densidad	155.99	1	4.13	0.042
Residual	67239.99	1782		

Cuadro 1: Analisis de varianza de los factores.

El cuadro 1 muestra el análisis de varianza y se observa que el algoritmo, orden-algoritmo y densidad-algoritmo infuyen en el tiempo de ejecución, mientras que los demás factores no afecta el tiempo de ejecución.

Referencias

[1] Schaeffer E. *Optimización de flujo en redes, 2019.*
https://elisa.dyndns-web.com/teaching/opt/flow/

[2] Kamada T. *An Algorithm for Drawing General Undirected Graphs*
Information Processing Letters, 1988.

[3] Saus L. *Repository of Github, 2019.*
https://github.com/pejli

Optimización flujo en redes

3

5175: Ángel Moreno

Tarea # 5: Caracterización estructural de instancias

April 30, 2019

Se utiliza el generador de grafo `wheel_graph` el consiste en generar un grafo donde tenga un nodo conectados a todos y los demas conectados en forma en circular, este tipo de grafo se uiliza comunmente cuando de un deposito o fuente se desea traladar o proporcionar algun material, energia, agua, etc. Las siguientes figuras muestran algunos ejemplos de instacias de un grafo con 16 nodos variendo la fuente y el sumidero para encontrar el flujo maximo utlizando el algoritmo `maximum_flow`.

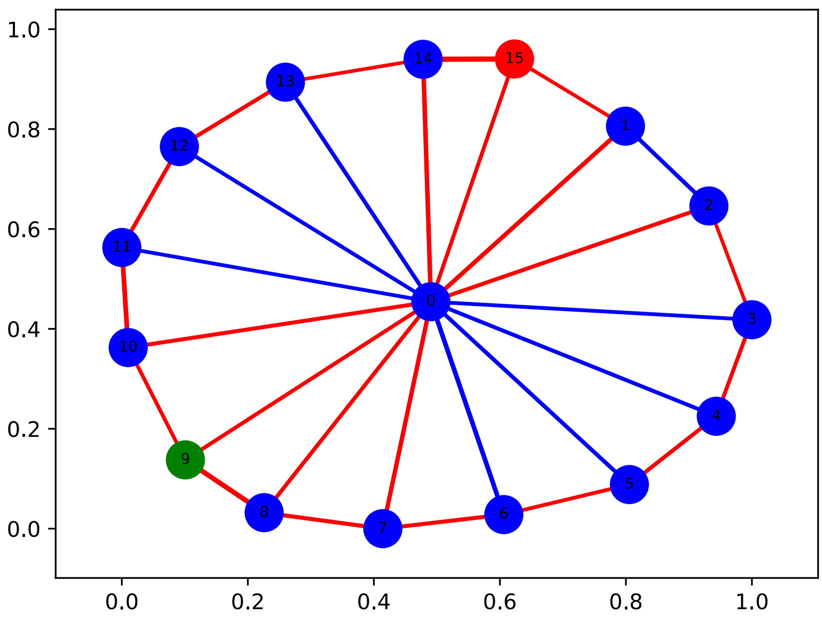


Figure 1: Grafo con fuente 9 y sumidero 15.

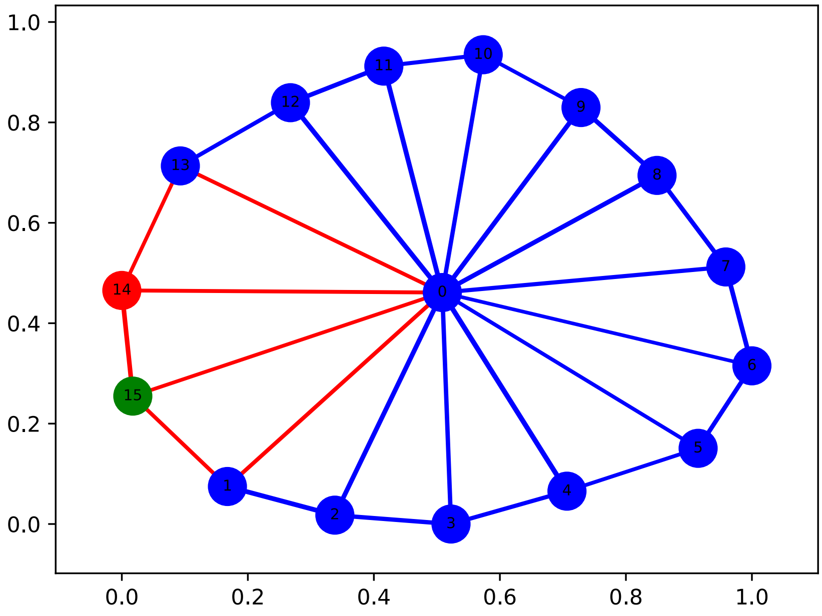


Figure 2: Grafo con fuente 15 y sumidero 14.

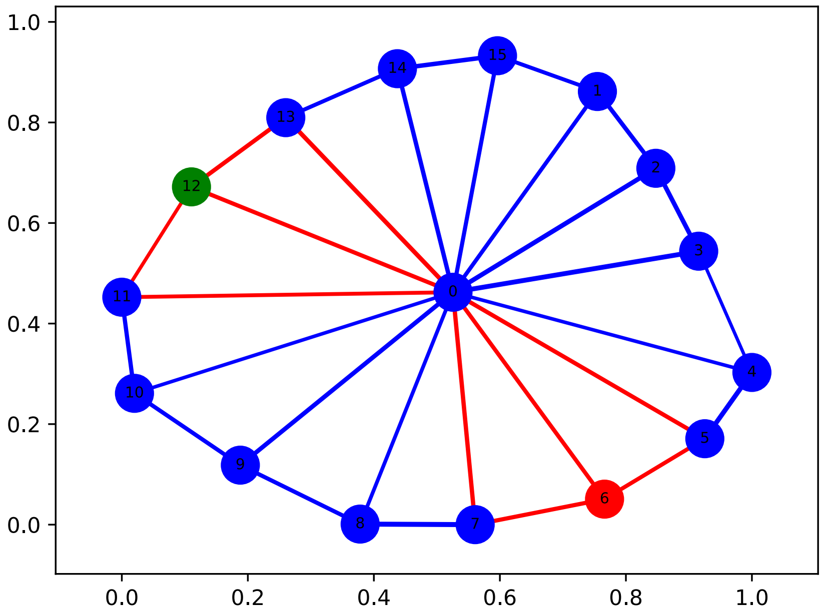


Figure 3: Grafo con fuente 12 y sumidero 6.

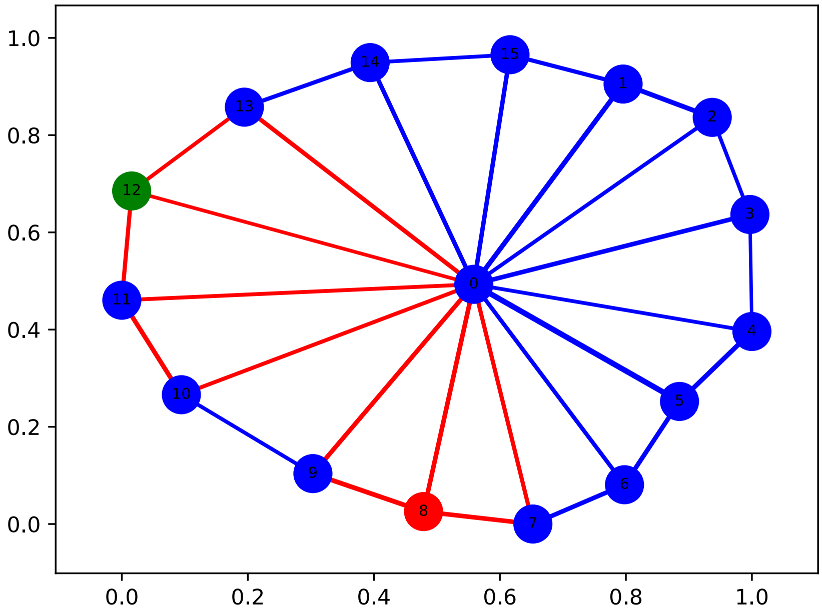


Figure 4: Grafo con fuente 12 y sumidero 8.

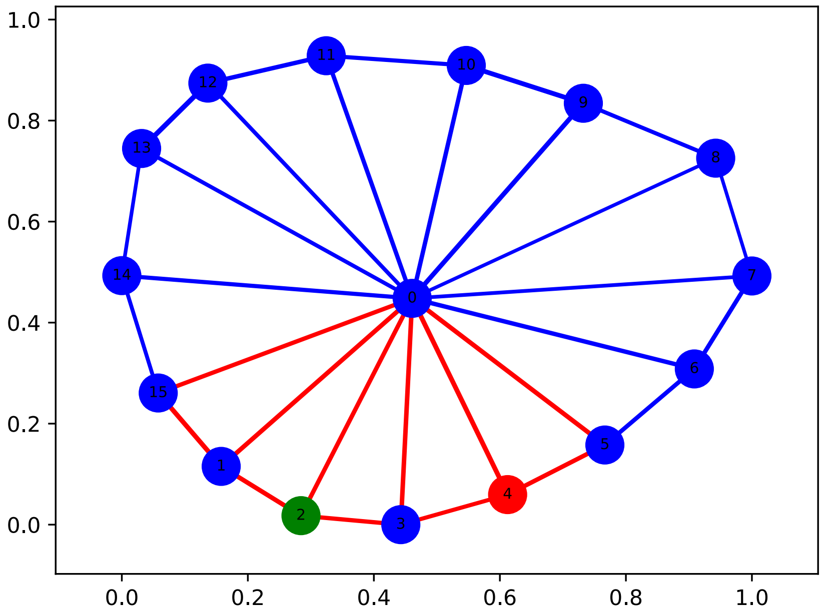


Figure 5: Grafo con fuente 2 y sumidero 4.

1 Resultados

Se ejecuta el algoritmo de flujo máximo un total de 30 réplicas para medir su tiempo de ejecución, recordadndo que los distintos grafos tienen diferentes fuentes y sumideros. La figura

6 muestra los resultados, se observa qe los tiempos se comportan similares para los grafos del 2 al 5, muestras que le grafo 1 demoro mas tiempo que los demas.

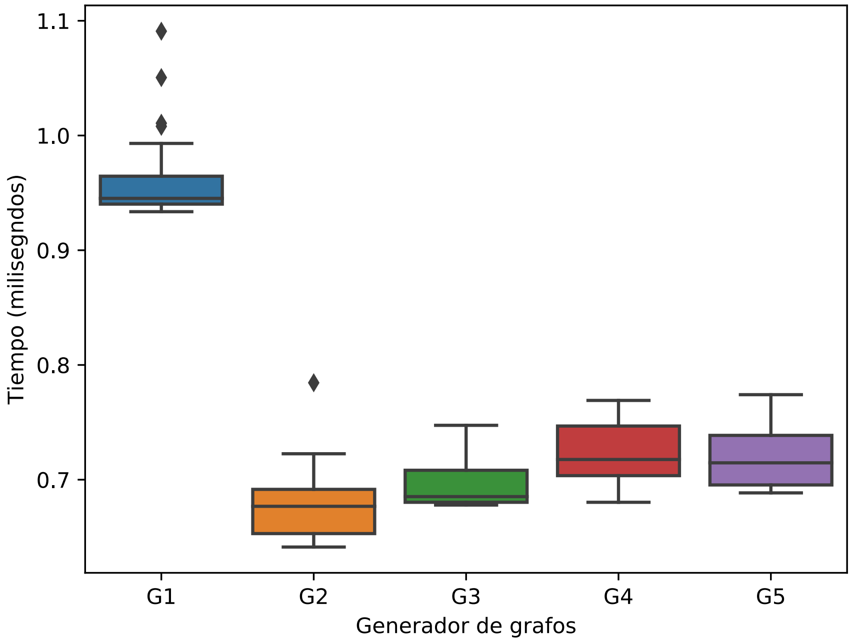
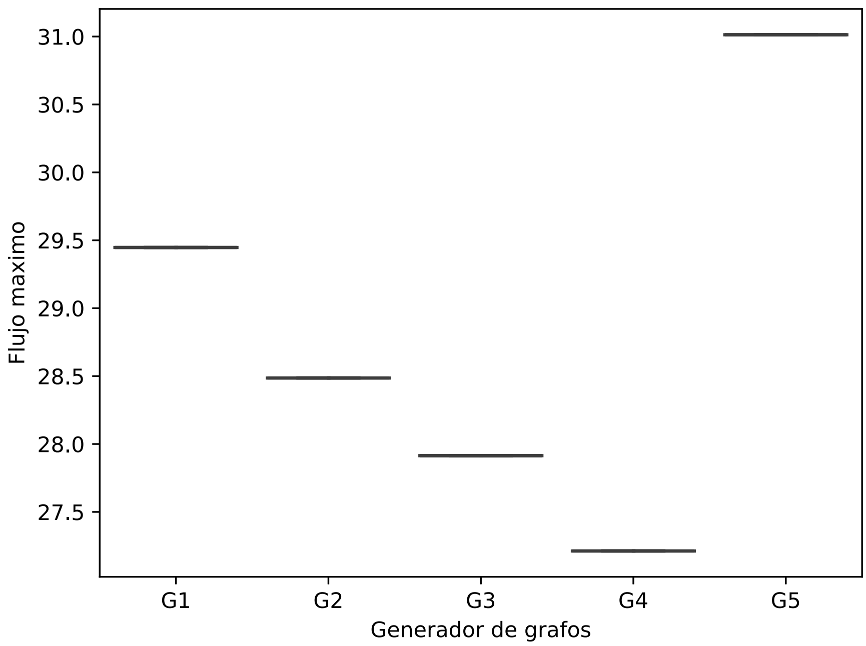


Figure 6: Efecto de las distintas instacias contra tiempo de ejecución.

Esto se debe a las fuentes y sumideros escogidos, el cual en el primer grafo los vertices de fuente y sumideres estoy muy lejos esto implica mayor tiempo de ejecución, mientras los demas se ejecutan en menos tiempo. Se recomienda utilizar fuentes y sumideros tan alejados para menor tiempo de ejecución.

La figura 7 muestra el flujo maximo obtenido para cada grafo.



lol

Figure 7: Efecto de las distintas instacias contra tiempo de flujo máximo.

References

[1] SCHAEFFER E. *Optimización de flujo en redes, 2019.*
`https://elisa.dyndns-web.com/teaching/opt/flow/`

[2] KAMADA T. *An Algorithm for Drawing General Undirected Graphs*
`Information Processing Letters, 1988.`

[3] SAUS L. *Repository of Github, 2019.*
`https://github.com/pejli`

Optimización flujo en redes

5175: Ángel Moreno

Tarea # 5: Caracterización estructural de instancias

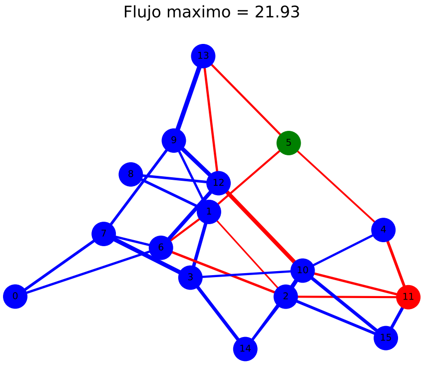
4 de junio de 2019

Resumen

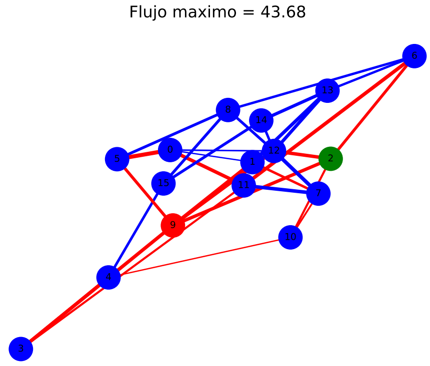
En esta tarea se ejecuta un experimento para determinar si el tiempo de ejecucion del experimento o el calculo del flujo maximo de un grafo se ven afectados por las caracteristicas de distribucion de grado, coeficiente de agrupamiento, centralidad de cercania, centralidad de carga, excentricidad y rango de pagina. Se utiliza el generdor de grafos `watts_strogatz_graph` de la libreria `networkx` de `python`, el algoritmo de acomodo `spring_layout` y el algoritmo de flujo maximo `maximum_flow`.

Grafos y sus caracteristicas

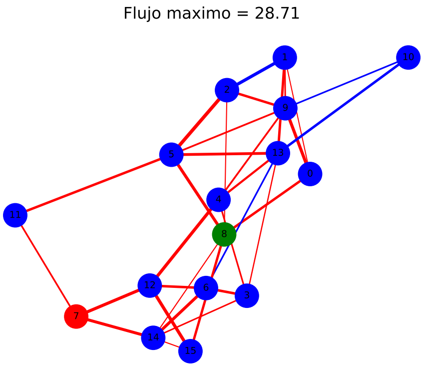
Segundo [4] el generdor de grafo `watts_strogatz_graph` fue diseñado para redes del mundo pequeño donde el grafo tiene n nodos y cada nodo se una con k vecinos con un probabilidad p. El algoritmo de acomodo `spring_layout` usa el algoritmo dirigido por fuerza de Fruchterman-Reingold [5], en este se consideran la fuerza que hay entre dos nodos. El algoritmo para el flujo maximo que se utiliza es el `maximum_flow`. La figura 1 muestra cinco grafos generados con las descripciones antes mencionadas ademas con el flujo maximo calculado.



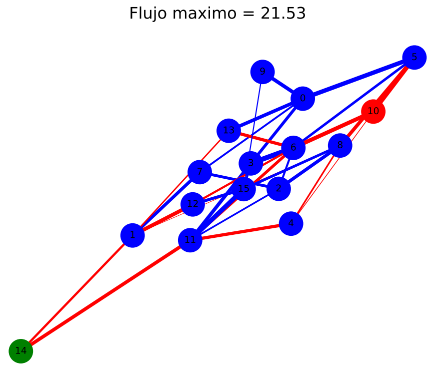
(a) Grafo 1



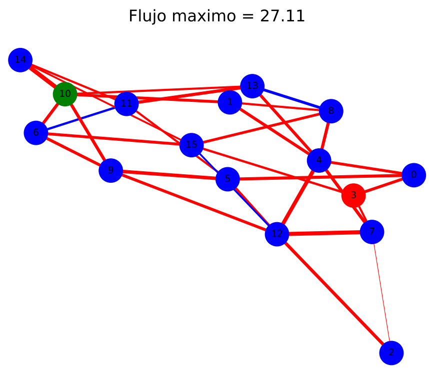
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4

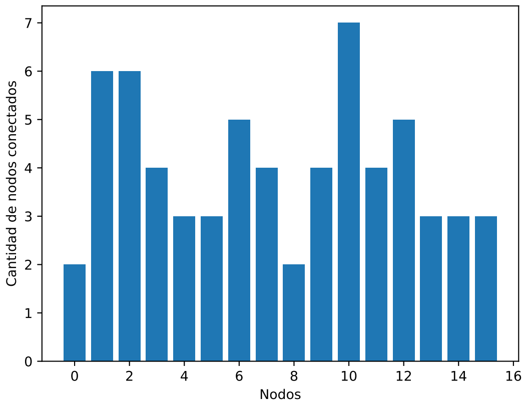


(e) Grafo 5

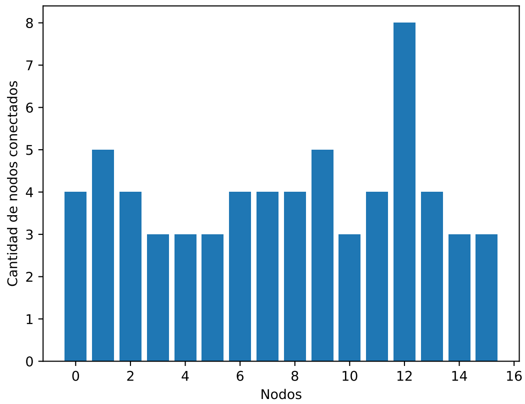
Figura 1: Grafica de las 5 instacias con el flujo maximo trazado.

Distribucin de grado

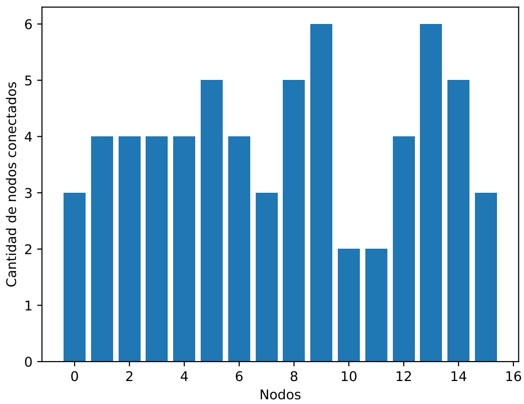
Esta caracteristica es calculada con la funcion `networkx.Graph.degree` el cual regresa por nodo cuantos nodos estan conectados.



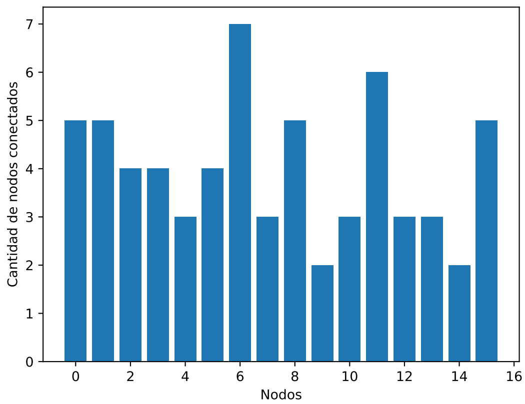
(a) Grafo 1



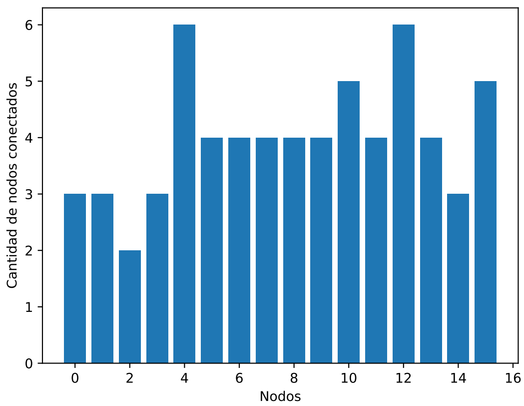
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

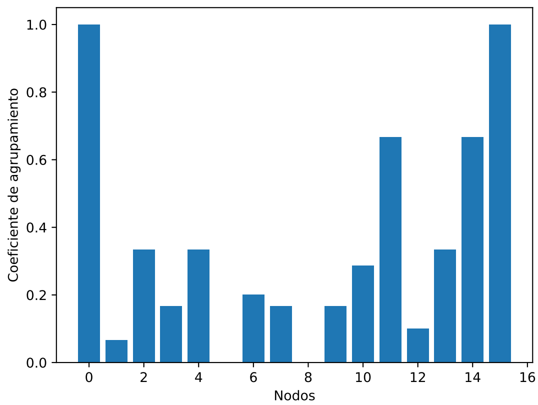
Figura 2: Histograma de la distribucion de grado de cada grafo.

Coeficiente de agrupamiento

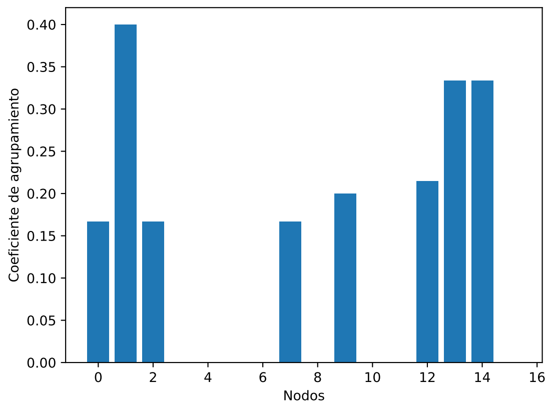
La forma para cacular esta medida esta dada por el promedio geometrico del peso de los subgrafos a cada nodo

$$c_u = \frac{2T(u)}{deg(u)(deg(u)-1)}$$

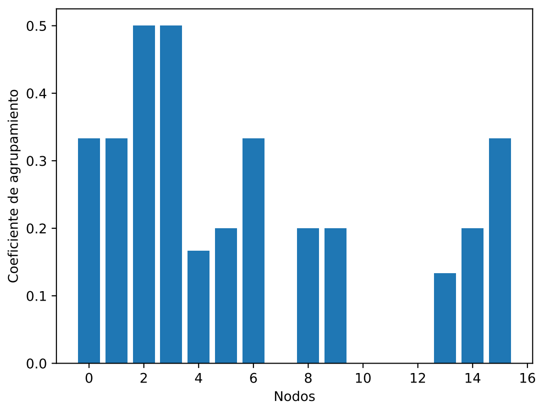
donde $T(u)$ es el numero de triangulos formados con el nodo u y $deg(u)$ es el grado de u.



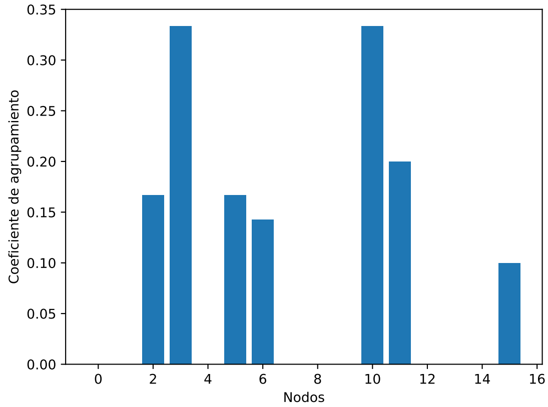
(a) Grafo 1



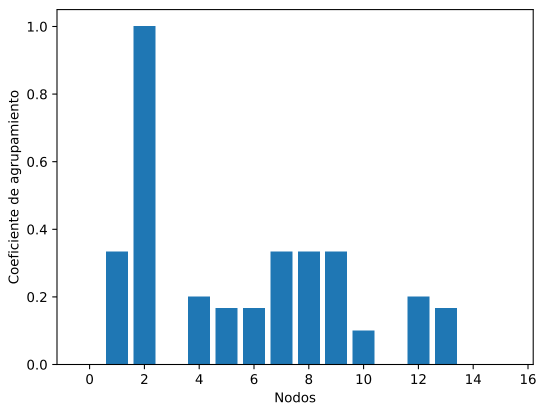
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

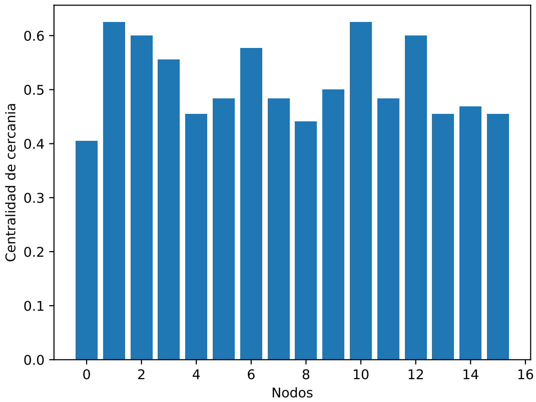
Figura 3: Histograma del coeficiente de agrupamiento de cada grafo.

Centralidad de cercanía

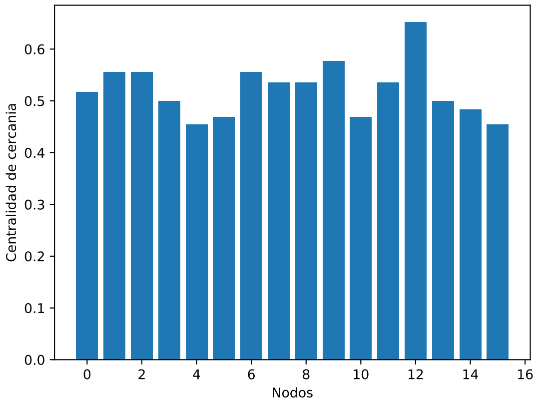
La centralidad de cercania de cada nodo es el cociente de la suma de las distancias del camino más corto desde nodo que se desa calcular esta caracteristica hasta todos los demas nodos. La formula es

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(u,v)}$$

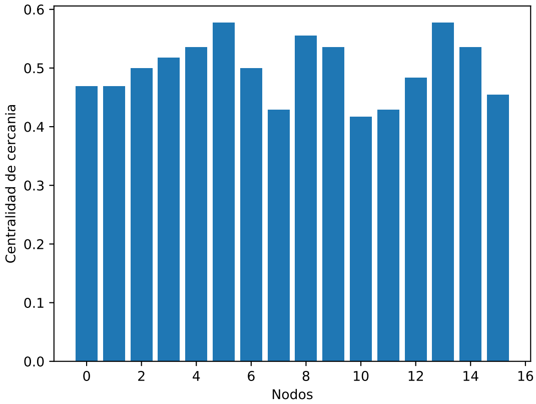
donde $d(u,v)$ es el camino camino mas corto entre u y v, y n es el numero de nodos en el grafo.



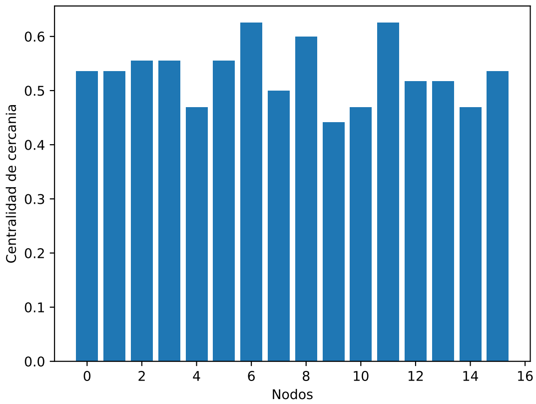
(a) Grafo 1



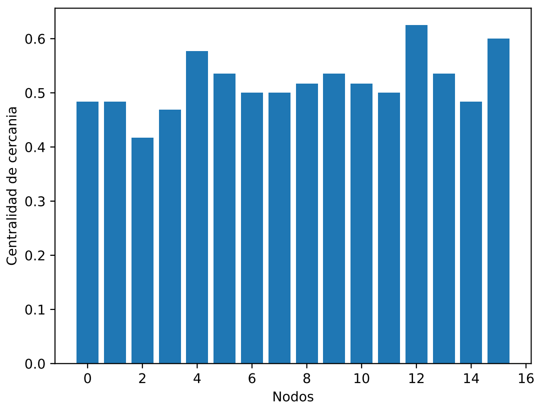
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4

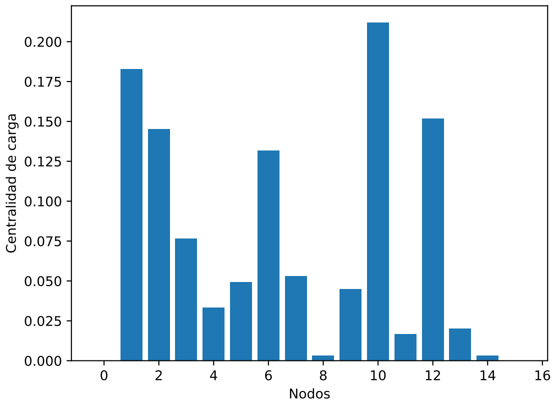


(e) Grafo 5

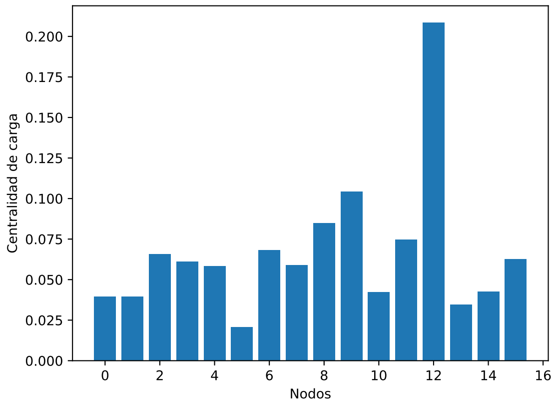
Figura 4: Histograma de la centralidad de cercanía de cada grafo.

Centralidad de carga

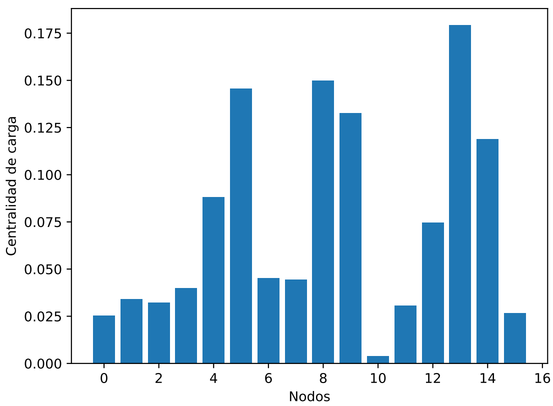
La centralidad de carga de un nodo es la fracción de todas las rutas más cortas que pasan a través de ese nodo.



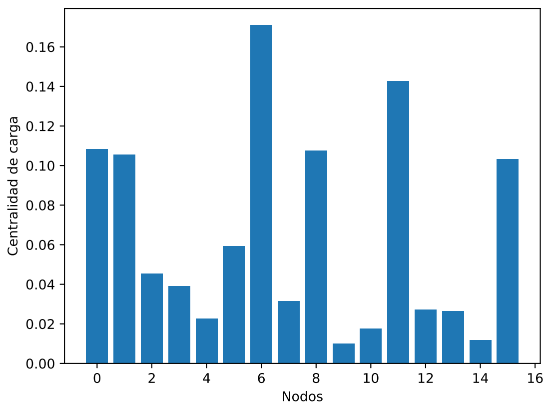
(a) Grafo 1



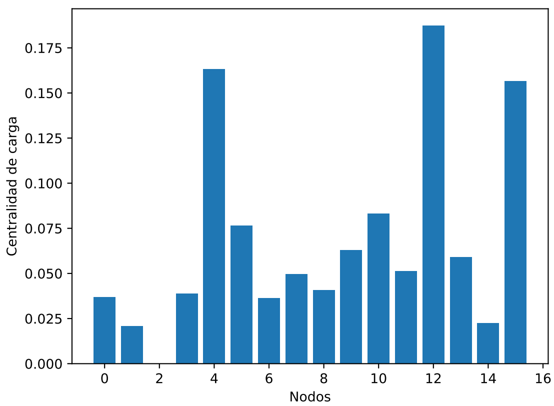
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4

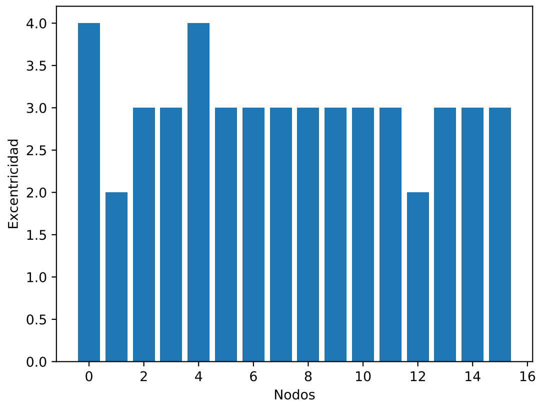


(e) Grafo 5

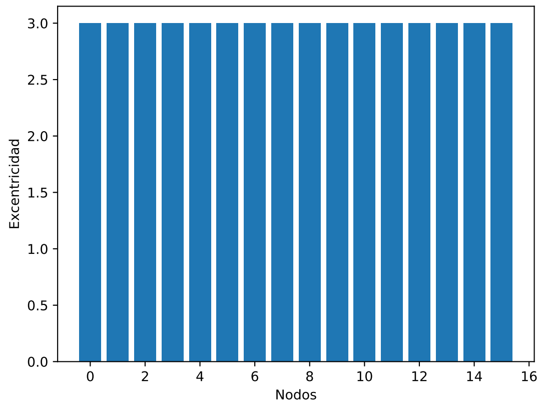
Figura 5: Histograma de la centralidad de carga de cada grafo.

Excentricidad

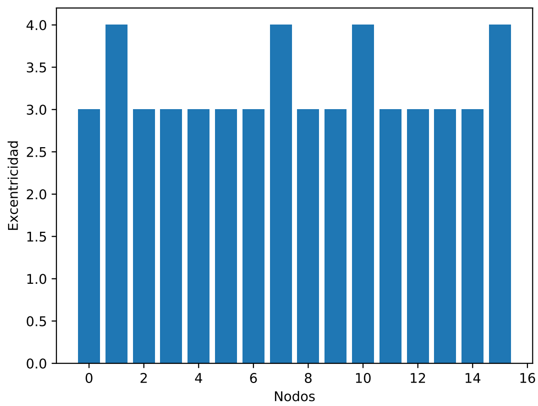
La excentricidad de un nodo u es la distancia máxima de el a todos los demás nodos en G.



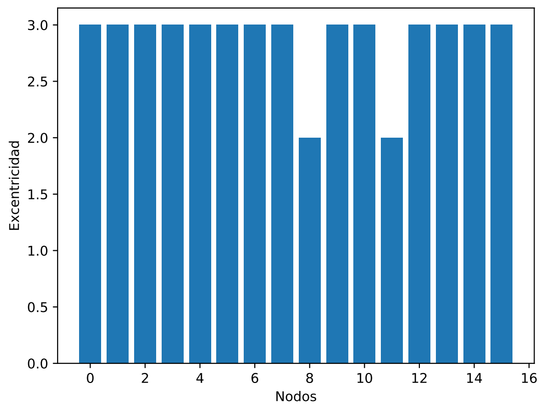
(a) Grafo 1



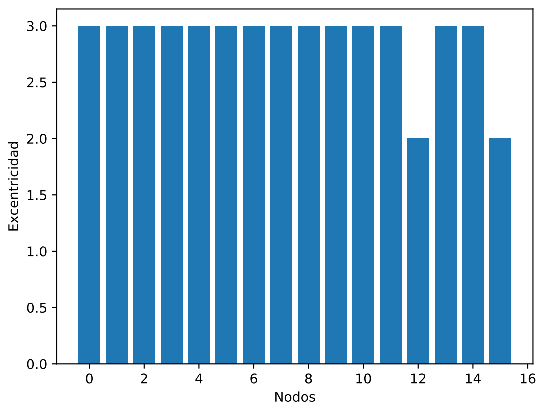
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4

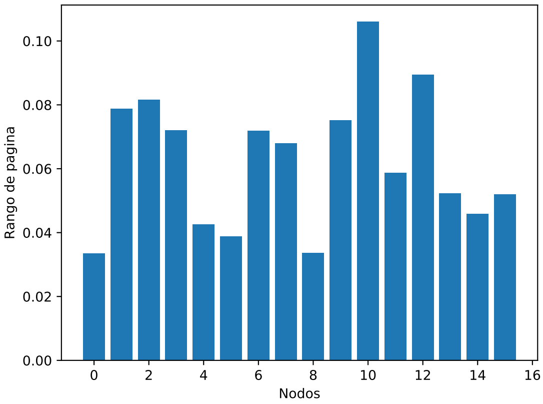


(e) Grafo 5

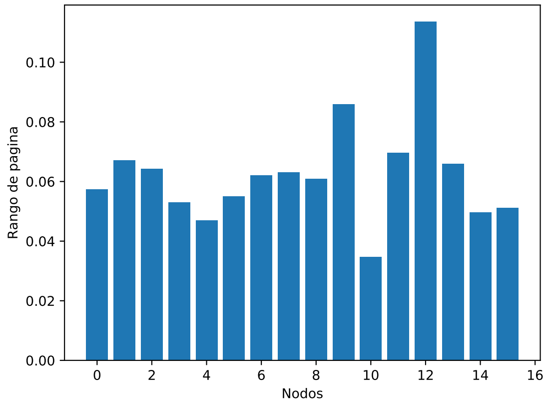
Figura 6: Histograma de la excentricidad de cada grafo.

Rango de pagina

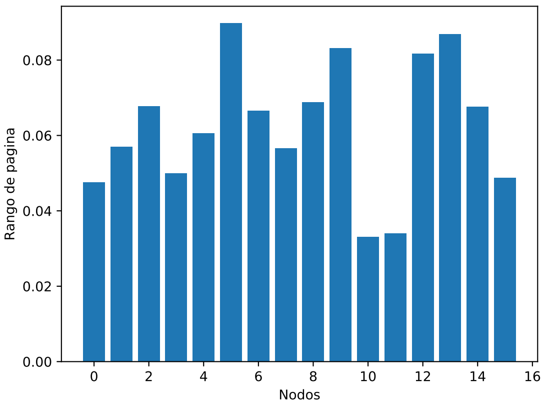
El rango de pagina calcula una clasificación de los nodos en el gráfico G en función de la estructura de las aristas entrantes. Originalmente fue diseñado como un algoritmo para clasificar páginas web.



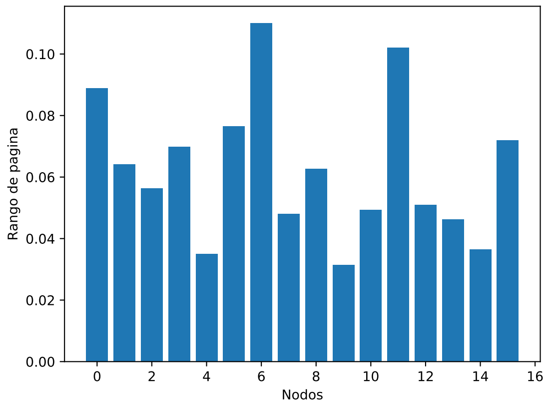
(a) Grafo 1



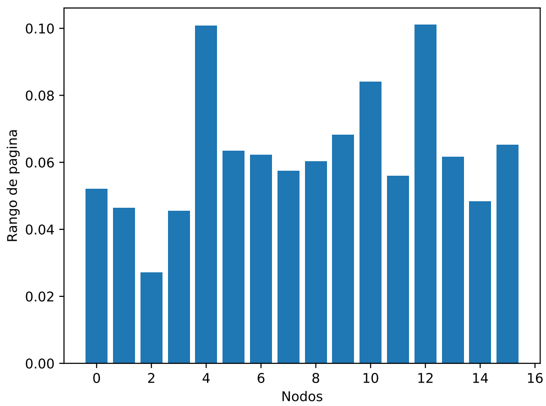
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 7: Histograma del rango de pagina de cada grafo.

1. Resultados

Se ejecuta el algoritmo de flujo máximo un total de 30 réplicas para medir su tiempo de ejecución, recordadndo que los distintos grafos tienen diferentes fuentes y sumideros. La figura muestra los resultados, se observa qe los tiempos se comportan similares para los grafos del 2 al 5, muestras que le grafo 1 demoro mas tiempo que los demas.

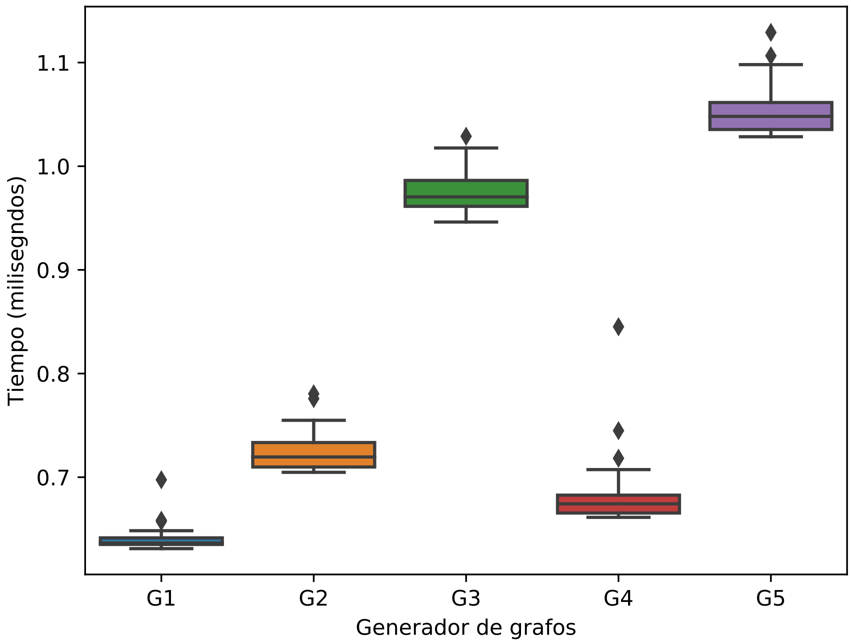


Figura 8: Resultados de tiempo de ejecucion contra grafo.

Esto se debe a las fuentes y sumideros escogidos, el cual en el primer grafo los vertices de fuente y sumideres estoy muy lejos esto implica mayor tiempo de ejecución, mientras los demas se ejecutan en menos tiempo. Se recomienda utilizar fuentes y sumideros tan alejados para menor tiempo de ejecución. Los cuadros 1 y 2 muestras los resultados del ANOVA de las caracteristas antes mencionadas.

	coef	**std err**	**t**	**P>** $\|t\|$	**[0.025**	**0.975]**
Intercept	-0.0284	0.057	-0.495	0.623	-0.143	0.086
Degree	-0.0120	0.015	-0.799	0.428	-0.042	0.018
Clustering	0.0399	0.024	1.663	0.102	-0.008	0.088
Closcentrality	0.1070	0.154	0.694	0.490	-0.201	0.415
Loadcentr	0.1281	0.329	0.389	0.698	-0.531	0.787
Eccentr	0.0094	0.016	0.577	0.566	-0.023	0.042
Pagerank	0.4874	0.794	0.614	0.542	-1.102	2.077
Degree:Clustering	0.0010	0.003	0.320	0.750	-0.005	0.007
Degree:Closcentrality	-0.0005	0.019	-0.025	0.980	-0.038	0.037
Degree:Loadcentr	0.0079	0.021	0.385	0.702	-0.033	0.049
Degree:Eccentr	0.0039	0.003	1.281	0.205	-0.002	0.010
Degree:Pagerank	0.0139	0.035	0.393	0.696	-0.057	0.085
Clustering:Closcentrality	-0.0532	0.042	-1.263	0.211	-0.138	0.031
Clustering:Loadcentr	-0.1557	0.071	-2.182	0.033	-0.299	-0.013
Clustering:Eccentr	-0.0065	0.003	-1.983	0.052	-0.013	6.19e-05
Clustering:Pagerank	0.0891	0.149	0.597	0.553	-0.210	0.388
Closcentrality:Loadcentr	0.2489	0.456	0.546	0.587	-0.664	1.162
Closcentrality:Eccentr	-0.0312	0.048	-0.652	0.517	-0.127	0.065
Closcentrality:Pagerank	-0.4397	1.154	-0.381	0.705	-2.749	1.870
Loadcentr:Eccentr	-0.0651	0.056	-1.159	0.251	-0.178	0.047
Loadcentr:Pagerank	-1.5144	1.066	-1.421	0.161	-3.648	0.619
Eccentr:Pagerank	-0.0765	0.156	-0.492	0.625	-0.388	0.235

Cuadro 1: ANOVA de tiempo contra cada uno de las caracteristicas.

	coef	**std err**	**t**	**P>** $|t|$	**[0.025**	**0.975]**
Intercept	652.6700	241.679	2.701	0.009	168.896	1136.444
Degree	-107.9745	67.775	-1.593	0.117	-243.642	27.693
Clustering	221.5764	183.587	1.207	0.232	-145.913	589.066
Closcentrality	-1484.4203	619.850	-2.395	0.020	-2725.186	-243.655
Loadcentr	2911.2114	1485.885	1.959	0.055	-63.110	5885.533
Eccentr	-123.5724	59.121	-2.090	0.041	-241.917	-5.228
Pagerank	1180.1544	3824.884	0.309	0.759	-6476.182	8836.491
Degree:Clustering	16.3447	21.682	0.754	0.454	-27.057	59.747
Degree:Closcentrality	248.9036	113.112	2.201	0.032	22.485	475.322
Degree:Loadcentr	-81.1284	96.433	-0.841	0.404	-274.160	111.904
Degree:Eccentr	1.6299	10.522	0.155	0.877	-19.432	22.692
Degree:Pagerank	-350.1165	286.652	-1.221	0.227	-923.912	223.679
Clustering:Closcentrality	-516.3110	413.216	-1.249	0.217	-1343.453	310.831
Clustering:Loadcentr	37.2367	316.199	0.118	0.907	-595.704	670.177
Clustering:Eccentr	-14.7959	16.200	-0.913	0.365	-47.223	17.631
Clustering:Pagerank	233.3437	959.092	0.243	0.809	-1686.487	2153.175
Closcentrality:Loadcentr	-3670.6632	2579.760	-1.423	0.160	-8834.614	1493.288
Closcentrality:Eccentr	282.1975	159.498	1.769	0.082	-37.073	601.468
Closcentrality:Pagerank	-687.4145	6758.639	-0.102	0.919	-1.42e+04	1.28e+04
Loadcentr:Eccentr	-346.5293	205.428	-1.687	0.097	-757.738	64.679
Loadcentr:Pagerank	6346.9764	6061.663	1.047	0.299	-5786.761	1.85e+04
Eccentr:Pagerank	54.8826	493.857	0.111	0.912	-933.680	1043.445

Cuadro 2: ANOVA de flujo maximo contra las caracteristicas.

Referencias

[1] Schaeffer E. *Optimización de flujo en redes, 2019.*
`https://elisa.dyndns-web.com/teaching/opt/flow/`

[2] Kamada T. *An Algorithm for Drawing General Undirected Graphs*
`Information Processing Letters, 1988.`

[3] Saus L. *Repository of Github, 2019.*
`https://github.com/pejli`

[4] Watts, Duncan J. Strogatz, Steven H. *Collective dynamics of 'small-world' networks, 1998*
`Magazine: Nature`

[5] Fruchterman, T. M. J., & Reingold, E. M. (1991). *Graph Drawing by Force-Directed Placement.*
`Software: Practice and Experience, 21(11).`

Optimización flujo en redes

5175: Ángel Moreno
Tarea # 6

4 de junio de 2019

Seleccion de proyecto

Sea P el conjunto de los posibles proyectos. Cada proyecto $v \in P$ tiene asociado un ingreso p_v. Algunas combinaciones de proyectos generan beneficio y otros generan costo. Algunos proyectos no pueden empezar hasta que que otros proyectos esten terminados. Sea D es el conjunto de dependencia que es un grafo aciclico, donde la arista $i \to j$ indica que el proyecto i depende de j. El problema consiste en encontrar un subconjunto de proyectos donde el ingreso sea maximo, en dado caso si el ingreso es negativo la solucion es no hacer nada.

Definimos un nuevo gráfico G agregando un vértice de origen s y un vértice de destino t del conjunto D, con la arista $s \to t$ para cada trabajo rentable (con $p_j > 0$), y con la arista $i \to t$ un costo de trabajo (con $p_i < 0$). Asignamos capacidades de borde de la siguiente manera:

- $c(s \to j) = p_j$ para todo trabajo j rentable
- $c(i \to t) = -p_i$ para todo costo de trabajo i
- $c(i \to j) = \infty$ para todo arista dependiente $i \to j$

Para todo subconjunto A de proyectos, se definen tres funciones:

$$cost(A) := \sum_{i \in A: p_i < 0} -p_i = \sum_{i \ inA} c(i \to j)$$

$$benefit(A) := \sum_{j \in A: p_i > 0} p_j = \sum_{j \in A} c(s \to j)$$

$$profit(A) := \sum_{i \in A} p_i = benefit(A) - cost(A)$$