Optimización de flujo en redes
Portafolio

Brenda Yaneth Sotelo Benítez
1565705

3 de junio de 2019

Introducción

En este reporte se recopilan las distintas tareas realizadas a lo largo de la unidad de aprendizaje de optimización de flujo en redes realizadas durante el período Enero-Junio 2019. Cada una de estas tareas contiene anotaciones con puntos a corregir, las cuales fueron tomadas en cuenta en la elaboración del portafolio. Se muestra cada tarea calificada y la respectiva tarea corregida si es necesario.

Tarea 1

Al realizar esta práctica y recibir retroalimentación, los principales errores encontrados fueron de ortografía, falta de espacios al citar, de ambiente matemático y errores en la bibliografía.

Representación de redes a través de la teoría de grafos

Brenda Yaneth Sotelo Benítez

12 de febrero de 2019

En esta práctica se realizó una investigación de tipos de grafos simples y multígrafos, donde para cada uno de ellos se presenta una definición,representación de un ejemplo y aplicaciones prácticas.

El lenguaje de programación que se ha utilizado para la creación y representación de los ejemplos es `Python`[1] con el uso del paquete NetwokX [2] y Matplotlib [3].

Se comenzará definiendo algunos conceptos importantes basados en [4, 5] con el objetivo de tener una estructura organizada para una clara comprensión del tema.

Conceptos básicos

Un ***grafo*** G es un par $G=(V,A)$ donde V es el conjunto de vértices o nodos y A un conjunto de aristas o arcos que se representan como pares de vértices (v_i,v_j) $v_i,v_j \in V$.

Si las aristas tienen dirección, es un grafo ***dirigido u orientado*** también llamado *digrafo*. En cada arista dirigida (v_i,v_j) el primer elemento representa el origen(o fuente) de la arista y el segundo es el destino. Si las aristas no tienen dirección se trata de un grafo ***no dirigido*** donde $(v_i, v_j)=(v_j, v_i)$ $\forall v_i, v_j \in V$. A los nodos y aristas se les puede asignar nombres o etiquetas, teniendo así un *grafo etiquetado*, mientras que asignar pesos a las aristas se denomina *grafo ponderado*.

Un ***ciclo*** es una secuencia de aristas consecutivas que empieza y termina en el mismo vértice. Un grafo que no contiene ciclos es ***acíclico***. Los ciclos de

longitud 1 se llaman *bucles o lazos*, ya que estos inician y terminan en el mismo vértice sin pasar por ningun otro.

Si un grafo cuenta con al menos un bucle , el grafo es ***reflexivo***.

En un grafo ***simple*** no existe más de una arista para cada par de vértices, por otro lado, si el grafo no es simple se le llama grafo múltiple o ***mutigrafo***.

1. Grafo simple no dirigido acíclico

Son usados para representar árboles genealógicos, organigramas, redes de distribución, carreteras(aristas) y ciudades(vértices) o bien en la química orgánica para representar compuestos. En la figura 1 se muestra un tipo de hidrocarburo saturado donde los vértices representan los atómos de carbono e hidrógeno y las aristas los enlaces entre ellos.

```
1 G=nx.Graph()
2 V={0:(0,0),1:(0.4,0),2:(0,1),3:(0.4,1),4:(0,2),5:(0.4,2)
      ,6:(0.2,5.0),7:(0.2,-3.0),8:(0.2,0),9:(0.2,1),
3     10:(0.2,2)}
4 A=[(0,1),(2,3),(4,5),(6,7),(8,9)]
5
6 nx.draw_networkx_nodes(G,V,nodelist=[0,1,2,3,4,5,6,7,8,9,10],
      node_color='black')
7 nx.draw_networkx_edges(G,V,width=1,edgelist=A)
```

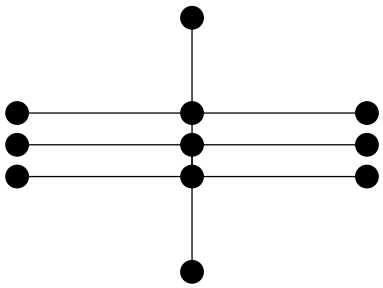


Figura 1: *Propano (C_3H_8)*

2. Grafo simple no dirigido cíclico

Entre sus aplicaciones está la representación de polígonos, redes eléctricas, en el problema del agente viajero, esquemas, representar calles en doble sentido, etc.

Un ejemplo red de ocho ordenadores se observa en la figura 2 que pueden conectarse de múltiples maneras y dan lugara un grafo no dirigido cíclico.

```
G=nx.Graph()
V={0:(0,0),1:(0,1),2:(1,0),3:(1,1),4:(0.5,0), 5:(0.5,1),6:(-0.5,1)
    ,7:(-0.5,2)}
A=[(0,1),(2,3),(4,5),(0,2),(1,5),(5,3),(1,6),(6,7)]
labels={2:'1',3:'2',4:'3',5:'4',0:'5',1:'6',6:'7',7:'8'}

nx.draw_networkx_nodes(G,V,nodelist=[0,1,2,3,4,5,6,7],node_color='b
    ')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_labels(G,V,labels, front_size=12)
```

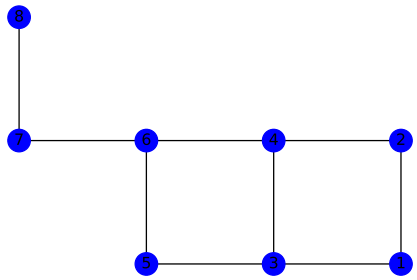


Figura 2: *Red de ordenadores*

3. Grafo simple no dirigido reflexivo

Durante la conducción es típico presentarse con situaciones en las que es necesario pasar de una avenida a otra o cambiar la dirección de la ruta. Para ello, nos vemos en la necesidad de buscar retornos ya establecidos o en su caso, buscar alguna calle para retornar y continuar con el viaje. Esta situación puede ser representada como un grafo tal y como se muestra en la figura 3 donde las aristas representan las calles y el vértice rojo el lazo o retorno.

```
G=nx.Graph()
V={0:(0,0),1:(1,0),2:(0.5,1),3:(0,1),4:(0,2)}
A=[(0,1),(1,2),(2,3),(3,4)]
labels={1:'L'}

nx.draw_networkx_nodes(G,V,nodelist=[1])
nx.draw_networkx_nodes(G,V,nodelist=[0,2,3,4],node_color='b')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_labels(G,V,labels, front_size=12)
```

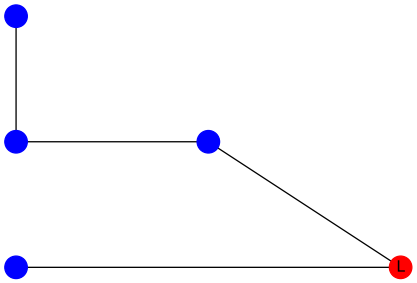


Figura 3: *Representación de calles con un retorno*

Otro ejemplo en la vida real se encuentra cuando brindamos ayuda para realizar alguna tarea o trabajo y al mismo tiempo se obtiene el beneficio de reforzar conocimiento(lazo). Pueden planterse muchos otros problemas más de esta manera pues hasta el organizar nuestras actividades del día puede ser representada por un grafo.

4. Grafo simple dirigido acíclico

En la figura 4 se plantea un modelo de conjuntos de tareas que necesitan una secuencia particular para realizarse y donde es importante que no existan ciclos para que las tareas no se repitan. Los problemas de transporte y asignación, asi como las redes neuronales pueden ser planteados de forma muy sencilla mediante este tipo de grafo.

```
G=nx.DiGraph()
V={0:(-2,0.2),1:(-1,0.2),2:(0,0.2),3:(0,0),4:(1,0.2),5:(1,0)
    ,6:(1.5,-0.2),7:(2,0.2),8:(2.5,0),9:(4,0),10:(5,0)}
A=[(0,1),(1,2),(2,3),(3,4),(3,5),(3,6),(4,7),(5,8),(8,6),(7,9)
    ,(9,10),(6,9),(1,3)]
labels={0:'1',1:'2',2:'3',3:'4',4:'5',5:'6',6:'9',7:'7',8:'8',9:'10
    ',10:'11'}

nx.draw_networkx_nodes(G,V,nodelist=[0,1,2,3,4,5,6,7,8,9,10],
    node_color='b')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_labels(G,V,labels, front_size=12)
```

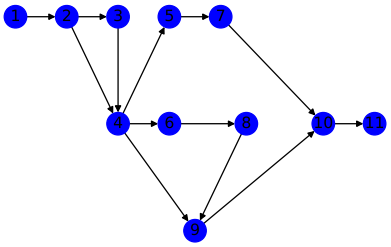


Figura 4: *Secuencia de tareas*

5. Grafo simple dirigido cíclico

El problema de VRP(ruteo de vehículos) es un problema clásico en la optimización de operaciones en el cual se tiene un depósito, un conjunto de vehículos que siguen una ruta y clientes que deben ser atendidos, donde cada uno representan las aristas y vértices respectivamente. Cada una de las rutas debe regresar al depósito. Ver figura 5.

```
G=nx.DiGraph()

V={0:(0,0),1:(1,1),2:(3,2),3:(4,1),4:(2,-1),5:(-1,2),6:(-3,1)
    ,7:(-4,-1),8:(-2,-2)}
A=[(0,1),(1,2),(2,3),(3,4),(4,0),(0,5),(5,6),(6,7),(7,8),(8,0)]
labels={0:'Centro'}

nx.draw_networkx_nodes(G,V,nodelist=[0,1,2,3,4],node_color='c')
nx.draw_networkx_nodes(G,V,nodelist=[5,6,7,8],node_color='y')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_labels(G,V,labels, front_size=12)
```

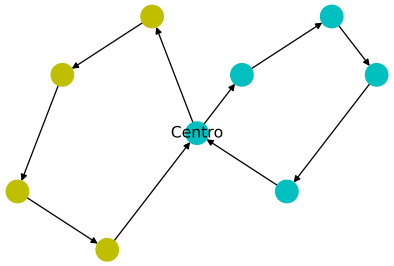


Figura 5: *VRP*

Los grafos simples dirigidos cíclicos también suelen presentarse en los diagramas de flujo.

6. Grafo simple dirigido reflexivo

Un ejemplo está dado por la representación de una tienda proveedora que vende productos a varias sucursales pequeñas y que a su vez vende para si misma. Ver figura 6. El vértice P es la tienda proveedora que tiene un lazo, mientras que los vétices etiquetados representan las tiendas.

```
G=nx.DiGraph()

V={0:(0,0),1:(1,5),2:(5,3),3:(7,0),4:(7,-3)}
A=[(0,1),(0,2),(0,3),(0,4),(3,4)]
labels={0:'P',1:'T1',2:'T2',3:'T3',4:'T4'}

nx.draw_networkx_nodes(G,V,nodelist=[1,2,3,4],node_color='c')
nx.draw_networkx_nodes(G,V,nodelist=[0],node_color='g')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_labels(G,V,labels, front_size=12)
```

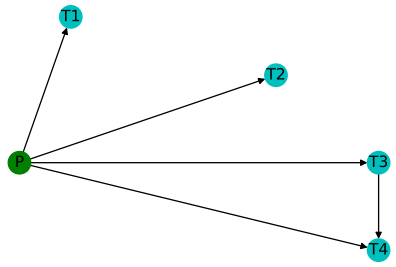


Figura 6: *Red de distribución*

7. Multigrafo no dirigido acíclico

El camino que sigue un río cuando pasa por ciertas localidades se puede representar mediante un multigrafo tal como se observa en figura 7. La arista roja representa en realidad dos aristas, dando lugar aun multigrafo. Esto debido a que un río suele dividirse cuando tiene de por medio obstáculos como tierra, piedras y se disparse en diferentes caminos beneficiando a varias comunidades.

```
1 G=nx.MultiGraph()
2
3 V={0:(0,0),1:(1,1.5),2:(2,1.5),3:(0,-3),4:(3,4), 5:(1,-3.5),6:(3,0)
      }
4 A=[(0,1),(1,2),(0,3),(2,4),(3,5),(2,6)]
5
6 nx.draw_networkx_nodes(G,V,nodelist=[0,1,2,3,4,5,6],node_color='b')
7 nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1,edge_color='c
      ')
8 nx.draw_networkx_edges(G,V,width=3,edgelist=[(0,3)],alpha=1,
      edge_color='r')
9 nx.draw_networkx_labels(G,V,front_size=12)
```

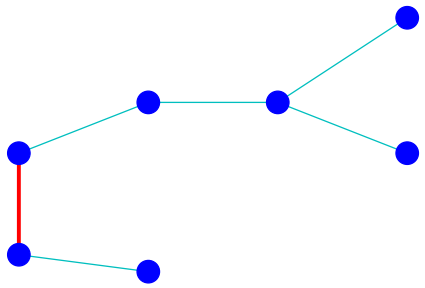


Figura 7: *Representación de un río*

8. Multigrafo no dirigido cíclico

En los aeropuertos existen diversas aerolíneas y cada una de ellas cuenta con cierta cantidad de aviones que parten a su destino y regresan al aeropuerto. Se puede plantear esta situación como un multigrafo dado que puede haber más de un avión que vaya al mismo destino, dirigido y cíclico por el hecho de regresar al origen después de hacer todas sus paradas. Ver figura 8.

```
1 G=nx.MultiGraph()
2
3 V={0:(-3,0),1:(1,0),2:(1,3),3:(-1,3)}
4 A=[(0,1),(0,2),(2,1),(3,1),(3,2)]
5
6 nx.draw_networkx_nodes(G,V,nodelist=[0,1,2,3],node_color='b')
7 nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
8 nx.draw_networkx_edges(G,V,width=3,edgelist=[(2,1)],alpha=1,
      edge_color='r')
9 nx.draw_networkx_labels(G,V,front_size=12)
```

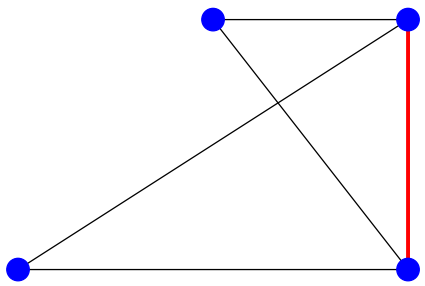


Figura 8: *Vuelos de aerolíneas*

9. Multigrafo no dirigido reflexivo

En el ámbito económico, la figura 9 presenta un multigrafo donde el vértice rojo representa a un inversionista que tiene a varios socios. Él puede decidir si invertir, que alguien invierta en él o que el mismo invierta en si mismo, lo cual esto último crearía el lazo en el grafo.

```
G=nx.MultiDiGraph()

V={0:(0,3),1:(0,0),2:(1,0.5),3:(2,2),4:(-2,2)}
A=[(0,1),(0,2),(0,3),(0,4)]
labels={0:'I', 1:'S1',2:'S2',3:'S3',4:'S4'}

nx.draw_networkx_nodes(G,V,nodelist=[1,2,3,4],node_color='b')
nx.draw_networkx_nodes(G,V,nodelist=[0],node_color='r')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_edges(G,V,width=3,edgelist=[(0,3)],alpha=1)
nx.draw_networkx_labels(G,V,labels, front_size=12)
```

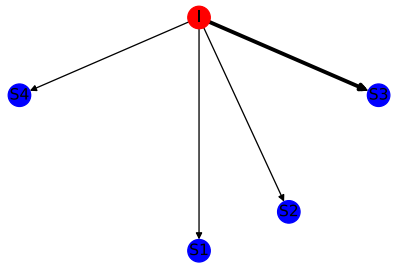


Figura 9: *Proceso de inversión*

10. Multigrafo dirigido acíclico

Un ejemplo de multigrafo dirigido cíclico se puede observar en los ductos de agua, que su representación se muestra en la figura 10 que están conectados de forma consecutiva sin formar ciclos y donde las aristas celestes son dos tuberías conectadas que llegan al mismo destino.

```
G=nx.MultiDiGraph()

V={0:(0,0),1:(1,0),2:(2,0.2),3:(3,0),4:(4,0)}
A=[(0,1),(1,2),(2,3),(3,4)]

nx.draw_networkx_nodes(G,V,nodelist=[0,1,2,3,4],node_color='b')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_edges(G,V,width=3,edgelist=[(0,1),(2,3)],alpha=1,
    edge_color='c')
```

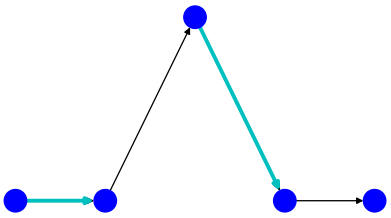


Figura 10: *Ductos de agua*

11. Multigrafo dirigido cíclico

Un ejemplo representativo está en las calles o avenidas, donde para llegar aun lugar existen varias alternativas y en su caso suele presentarse que se forme un ciclo para poder salir de una de ellas. O en el peor de los casos, toparse con calles sin salida o no saber a donde dirigirse. Las flechas resaltadas indican que entre ese par de vértices existen dos aristas o calles. Ver figura 11.

```
G=nx.MultiDiGraph()

V={0:(0,0),1:(1,0),2:(1.5,-1),3:(2,1),4:(-1,0),5:(-2,-1),6:(0,-1)}
A=[(0,6),(6,5),(5,4),(4,0),(1,0),(3,1),(2,1)]

nx.draw_networkx_nodes(G,V,nodelist=[0,1,2,3,4,5,6],node_color='b')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_edges(G,V,width=3,edgelist=[(1,0)],alpha=1)
```

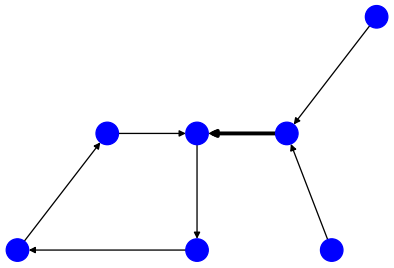


Figura 11: *Representación de avenidas*

Otra manera en la que se hace uso de este multigrafo es en la elaboración de esquemas donde se representen las tareas que cada trabajador debe realizar y considerando que una misma persona puede repetir tareas o más de una persona puede realizar la mismaz tarea.

12. Multigrafo dirigido reflexivo

En la figura 12 se plantea un modelo donde un hospital suministra medicinas a varias estaciones moviles, pero también se suministra el mismo.

```
G=nx.MultiDiGraph()

V={0:(0,0),1:(1,1),2:(1,-1),3:(-1.5,-1.5),4:(0,-3)}
A=[(0,1),(0,2),(0,3),(0,4)]
labels={0:'Hospital'}

nx.draw_networkx_nodes(G,V,nodelist=[1,2,3,4],node_color='pink')
nx.draw_networkx_nodes(G,V,nodelist=[0],node_color='r')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_edges(G,V,width=3,edgelist=[(0,3)],alpha=1,
    edge_color='c')
nx.draw_networkx_labels(G,V,labels, front_size=12)
```

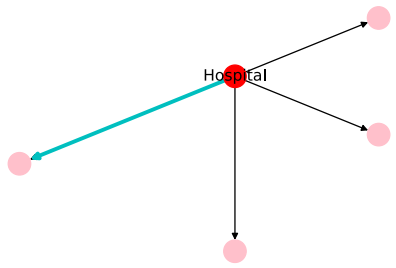


Figura 12: *Distibución de ayuda*

Referencias

[1] https://www.python.org/.

[2] https://networkx.github.io/documentation/networkx 1.10/reference/classes.html.

[3] https://matplotlib.org/.

[4] Alfredo Caicedo Barrero, Graciela Wagner de García, and Rosa María Méndez Parra. *Introducción a la Teoría de Grafos*. ELIZCOM SAS, 2010.

[5] Juana María Alonso Revenga. *Flujo en Redes y Gestión de Proyectos. Teoría y Ejercicios Resueltos.* Netbiblo, 2008.

Representación de redes a través de la teoría de grafos

Brenda Yaneth Sotelo Benítez

3 de junio de 2019

En esta práctica se realizó una investigación de tipos de grafos simples y multígrafos, donde para cada uno de ellos se presenta una definición, representación de un ejemplo y aplicaciones prácticas.

El lenguaje de programación que se ha utilizado para la creación y representación de los ejemplos es `Python` [1] con el uso del paquete Netwokx [2] y Matplotlib [3].

Se definen conceptos importantes basados en Barrero [4] y Alonso [5] con el objetivo de tener una estructura organizada para una clara comprensión del tema.

Conceptos básicos

Un ***grafo*** G es un par $G=(V,A)$ donde V es el conjunto de vértices o nodos y A un conjunto de aristas o arcos que se representan como pares de vértices (v_i,v_j), $v_i,v_j \in V$.

Si las aristas tienen dirección, es un grafo ***dirigido u orientado*** también llamado *digrafo*. En cada arista dirigida (v_i,v_j) el primer elemento representa el origen(o fuente) de la arista y el segundo es el destino. Si las aristas no tienen dirección se trata de un grafo ***no dirigido***. A los nodos y aristas se les puede asignar nombres o etiquetas, teniendo así un *grafo etiquetado*, mientras que asignar pesos a las aristas se denomina *grafo ponderado*.

Un ***ciclo*** es una secuencia de aristas consecutivas que empieza y termina en el mismo vértice. Un grafo que no contiene ciclos es ***acíclico***. Los ciclos de

longitud 1 se llaman *bucles o lazos*, ya que estos inician y terminan en el mismo vértice sin pasar por ningun otro.

Si un grafo cuenta con al menos un bucle, el grafo es ***reflexivo***.

En un grafo ***simple*** no existe más de una arista para cada par de vértices, por otro lado, si el grafo no es simple se le llama grafo múltiple o ***mutigrafo***.

1. Grafo simple no dirigido acíclico

Son usados para representar árboles genealógicos, organigramas, redes de distribución, carreteras (aristas) y ciudades (vértices) o bien en la química orgánica para representar compuestos. En la figura 1 se muestra un tipo de hidrocarburo saturado donde los vértices representan los atómos de carbono e hidrógeno y las aristas los enlaces entre ellos.

```
1 G=nx.Graph()
2 print(G)
3 V={0:(0,0),1:(0.4,0),2:(0,1),3:(0.4,1),4:(0,2),5:(0.4,2)
      ,6:(0.2,5.0),7:(0.2,-3.0),8:(0.2,0),9:(0.2,1),
4     10:(0.2,2)}
5 A=[(0,1),(2,3),(4,5),(6,7),(8,9)]
6
7 nx.draw_networkx_nodes(G,V,nodelist=[0,1,2,3,4,5,6,7,8,9,10],
      node_color='black')
```

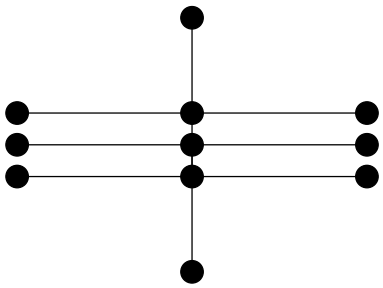


Figura 1: *Propano (C_3H_8)*

2. Grafo simple no dirigido cíclico

Entre sus aplicaciones está la representación de polígonos, redes eléctricas, en el problema del agente viajero, esquemas, representar calles en doble sentido, etc.

Un ejemplo red de ocho ordenadores se observa en la figura 2 que pueden conectarse de múltiples maneras y dan lugara un grafo no dirigido cíclico.

```
G=nx.Graph()
V={0:(0,0),1:(0,1),2:(1,0),3:(1,1),4:(0.5,0), 5:(0.5,1),6:(-0.5,1)
    ,7:(-0.5,2)}
A=[(0,1),(2,3),(4,5),(0,2),(1,5),(5,3),(1,6),(6,7)]
labels={2:'1',3:'2',4:'3',5:'4',0:'5',1:'6',6:'7',7:'8'}

nx.draw_networkx_nodes(G,V,nodelist=[0,1,2,3,4,5,6,7],node_color='b
    ')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_labels(G,V,labels, front_size=12)
```

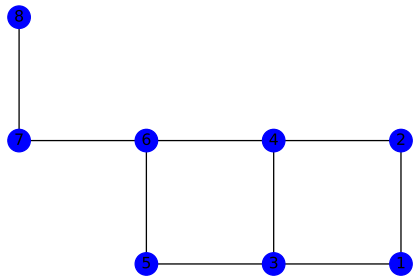


Figura 2: *Red de ordenadores*

3. Grafo simple no dirigido reflexivo

Durante la conducción es típico presentarse con situaciones en las que es necesario pasar de una avenida a otra o cambiar la dirección de la ruta. Para ello, es necesario buscar retornos ya establecidos o en su caso, buscar alguna calle para retornar y continuar con el viaje. Esta situación puede ser representada como un grafo tal y como se muestra en la figura 3 donde las aristas representan las calles y el vértice rojo el lazo o retorno.

```
G=nx.Graph()
V={0:(0,0),1:(1,0),2:(0.5,1),3:(0,1),4:(0,2)}
A=[(0,1),(1,2),(2,3),(3,4)]
labels={1:'L'}

nx.draw_networkx_nodes(G,V,nodelist=[1])
nx.draw_networkx_nodes(G,V,nodelist=[0,2,3,4],node_color='b')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_labels(G,V,labels, front_size=12)
```

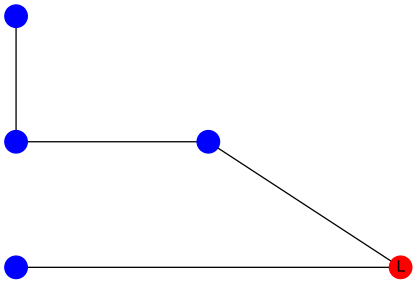


Figura 3: *Representación de calles con un retorno*

Otro ejemplo en la vida real se encuentra cuando se brinda ayuda para realizar alguna tarea o trabajo y al mismo tiempo se obtiene el beneficio de reforzar conocimiento (lazo). Pueden plantearse muchos otros problemas más de esta manera pues hasta el organizar nuestras actividades del día puede ser representada por un grafo.

4. Grafo simple dirigido acíclico

En la figura 4 se plantea un modelo de conjuntos de tareas que necesitan una secuencia particular para realizarse y donde es importante que no existan ciclos para que las tareas no se repitan. Los problemas de transporte y asignación, asi como las redes neuronales pueden ser planteados de forma muy sencilla mediante este tipo de grafo.

```
G=nx.DiGraph()
V={0:(-2,0.2),1:(-1,0.2),2:(0,0.2),3:(0,0),4:(1,0.2),5:(1,0)
    ,6:(1.5,-0.2),7:(2,0.2),8:(2.5,0),9:(4,0),10:(5,0)}
A=[(0,1),(1,2),(2,3),(3,4),(3,5),(3,6),(4,7),(5,8),(8,6),(7,9)
    ,(9,10),(6,9),(1,3)]
labels={0:'1',1:'2',2:'3',3:'4',4:'5',5:'6',6:'9',7:'7',8:'8',9:'10
    ',10:'11'}

nx.draw_networkx_nodes(G,V,nodelist=[0,1,2,3,4,5,6,7,8,9,10],
    node_color='b')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_labels(G,V,labels, front_size=12)
```

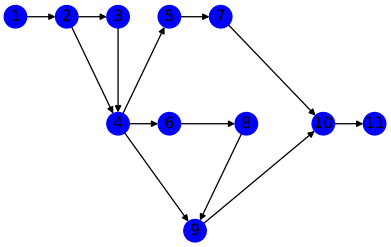


Figura 4: *Secuencia de tareas*

5. Grafo simple dirigido cíclico

El problema de VRP (ruteo de vehículos) es un problema clásico en la optimización de operaciones en el cual se tiene un depósito, un conjunto de vehículos que siguen una ruta y clientes que deben ser atendidos, donde cada uno representa las aristas y vértices respectivamente. Cada una de las rutas debe regresar al depósito. Ver figura 5.

```
G=nx.DiGraph()

V={0:(0,0),1:(1,1),2:(3,2),3:(4,1),4:(2,-1),5:(-1,2),6:(-3,1)
      ,7:(-4,-1),8:(-2,-2)}
A=[(0,1),(1,2),(2,3),(3,4),(4,0),(0,5),(5,6),(6,7),(7,8),(8,0)]
labels={0:'Centro'}

nx.draw_networkx_nodes(G,V,nodelist=[0,1,2,3,4],node_color='c')
nx.draw_networkx_nodes(G,V,nodelist=[5,6,7,8],node_color='y')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_labels(G,V,labels,  front_size=12)
```

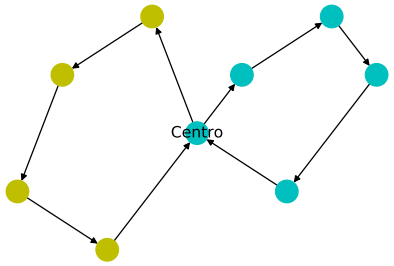


Figura 5: *VRP*

Los grafos simples dirigidos cíclicos también suelen presentarse en los diagramas de flujo.

6. Grafo simple dirigido reflexivo

Un ejemplo está dado por la representación de una tienda proveedora que vende productos a varias sucursales pequeñas y que a su vez vende para si misma. Ver figura 6. El vértice P es la tienda proveedora que tiene un lazo, mientras que los vétices etiquetados representan las tiendas.

```
G=nx.DiGraph()

V={0:(0,0),1:(1,5),2:(5,3),3:(7,0),4:(7,-3)}
A=[(0,1),(0,2),(0,3),(0,4),(3,4)]
labels={0:'P',1:'T1',2:'T2',3:'T3',4:'T4'}

nx.draw_networkx_nodes(G,V,nodelist=[1,2,3,4],node_color='c')
nx.draw_networkx_nodes(G,V,nodelist=[0],node_color='g')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_labels(G,V,labels,  front_size=12)
```

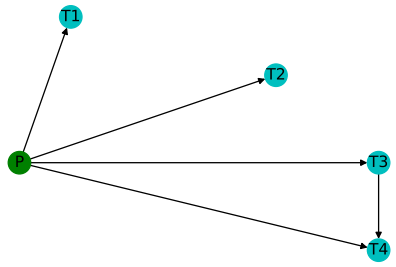


Figura 6: *Red de distribución*

7. Multigrafo no dirigido acíclico

El camino que sigue un río cuando pasa por ciertas localidades se puede representar mediante un multigrafo tal como se observa en figura 7. La arista roja representa en realidad dos aristas, dando lugar aun multigrafo. Esto debido a que un río suele dividirse cuando tiene de por medio obstáculos como tierra, piedras y se disparse en diferentes caminos beneficiando a varias comunidades.

```
1 G=nx.MultiGraph()
2
3 V={0:(0,0),1:(1,1.5),2:(2,1.5),3:(0,-3),4:(3,4), 5:(1,-3.5),6:(3,0)
      }
4 A=[(0,1),(1,2),(0,3),(2,4),(3,5),(2,6)]
5
6 nx.draw_networkx_nodes(G,V,nodelist=[0,1,2,3,4,5,6],node_color='b')
7 nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1,edge_color='c
      ')
8 nx.draw_networkx_edges(G,V,width=3,edgelist=[(0,3)],alpha=1,
      edge_color='r')
9 nx.draw_networkx_labels(G,V,front_size=12)
```

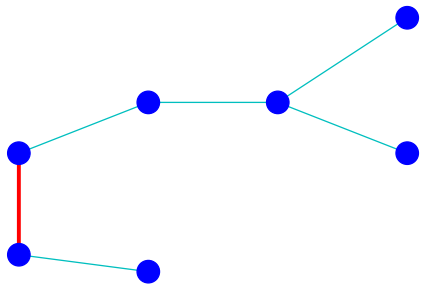


Figura 7: *Representación de un río*

8. Multigrafo no dirigido cíclico

En los aeropuertos existen diversas aerolíneas y cada una de ellas cuenta con cierta cantidad de aviones que parten a su destino y regresan al aeropuerto. Se puede plantear esta situación como un multigrafo, dado que puede haber más de un avión que vaya al mismo destino, sea dirigido y cíclico por el hecho de regresar al origen después de hacer todas sus paradas. Ver figura 8.

```
1 G=nx.MultiGraph()
2
3 V={0:(-3,0),1:(1,0),2:(1,3),3:(-1,3)}
4 A=[(0,1),(0,2),(2,1),(3,1),(3,2)]
5
6 nx.draw_networkx_nodes(G,V,nodelist=[0,1,2,3],node_color='b')
7 nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
8 nx.draw_networkx_edges(G,V,width=3,edgelist=[(2,1)],alpha=1,
      edge_color='r')
9 nx.draw_networkx_labels(G,V,front_size=12)
```

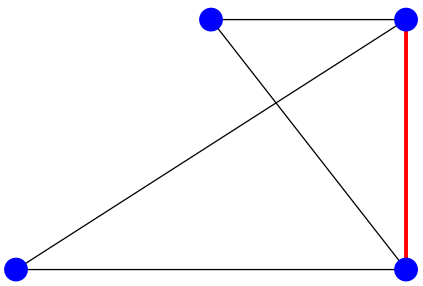


Figura 8: *Vuelos de aerolíneas*

9. Multigrafo no dirigido reflexivo

En el ámbito económico, la figura 9 presenta un multigrafo donde el vértice rojo representa a un inversionista que tiene a varios socios. Él puede decidir si invertir, que inviertan en él o invertir en si mismo, lo cual esto último crearía el lazo en el grafo.

```
G=nx.MultiDiGraph()

V={0:(0,3),1:(0,0),2:(1,0.5),3:(2,2),4:(-2,2)}
A=[(0,1),(0,2),(0,3),(0,4)]
labels={0:'I', 1:'S1',2:'S2',3:'S3',4:'S4'}

nx.draw_networkx_nodes(G,V,nodelist=[1,2,3,4],node_color='b')
nx.draw_networkx_nodes(G,V,nodelist=[0],node_color='r')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_edges(G,V,width=3,edgelist=[(0,3)],alpha=1)
nx.draw_networkx_labels(G,V,labels, front_size=12)
```

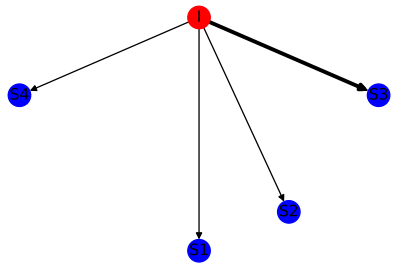


Figura 9: *Proceso de inversión*

10. Multigrafo dirigido acíclico

Un ejemplo de multigrafo dirigido cíclico se puede observar en los ductos de agua, que su representación se muestra en la figura 10 que están conectados de forma consecutiva sin formar ciclos y donde las aristas celestes son dos tuberías conectadas que llegan al mismo destino.

```
1 G=nx.MultiDiGraph()
2
3 V={0:(0,0),1:(1,0),2:(2,0.2),3:(3,0),4:(4,0)}
4 A=[(0,1),(1,2),(2,3),(3,4)]
5
6 nx.draw_networkx_nodes(G,V,nodelist=[0,1,2,3,4],node_color='b')
7 nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
8 nx.draw_networkx_edges(G,V,width=3,edgelist=[(0,1),(2,3)],alpha=1,
      edge_color='c')
```

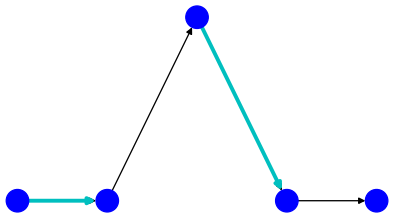


Figura 10: *Ductos de agua*

11. Multigrafo dirigido cíclico

Un ejemplo representativo está en las calles o avenidas, donde para llegar aun lugar existen varias alternativas y en su caso suele presentarse que se forme un ciclo para poder salir de una de ellas. En el peor de los casos, toparse con calles sin salida o no saber a donde dirigirse. Las flecha resaltada indica que entre ese par de vértices existen dos aristas o calles. Ver figura 11.

```
G=nx.MultiDiGraph()

V={0:(0,0),1:(1,0),2:(1.5,-1),3:(2,1),4:(-1,0),5:(-2,-1),6:(0,-1)}
A=[(0,6),(6,5),(5,4),(4,0),(1,0),(3,1),(2,1)]

nx.draw_networkx_nodes(G,V,nodelist=[0,1,2,3,4,5,6],node_color='b')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_edges(G,V,width=3,edgelist=[(1,0)],alpha=1)
```

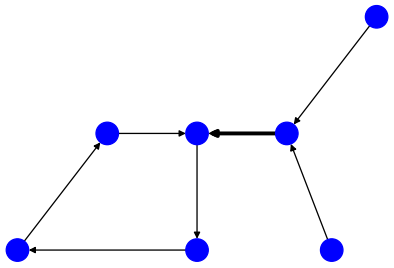


Figura 11: *Representación de avenidas*

Otra manera en la que se hace uso de este multigrafo es en la elaboración de esquemas donde se representen las tareas que cada trabajador debe realizar y considerando que una misma persona puede repetir tareas o más de una persona puede realizar la mismas tarea.

12. Multigrafo dirigido reflexivo

En la figura 12 se plantea un modelo donde un hospital suministra medicinas a varias estaciones móviles, pero también se suministra el mismo.

```
G=nx.MultiDiGraph()

V={0:(0,0),1:(1,1),2:(1,-1),3:(-1.5,-1.5),4:(0,-3)}
A=[(0,1),(0,2),(0,3),(0,4)]
labels={0:'Hospital'}

nx.draw_networkx_nodes(G,V,nodelist=[1,2,3,4],node_color='pink')
nx.draw_networkx_nodes(G,V,nodelist=[0],node_color='r')
nx.draw_networkx_edges(G,V,width=1,edgelist=A,alpha=1)
nx.draw_networkx_edges(G,V,width=3,edgelist=[(0,3)],alpha=1,
    edge_color='c')
nx.draw_networkx_labels(G,V,labels, front_size=12)
```

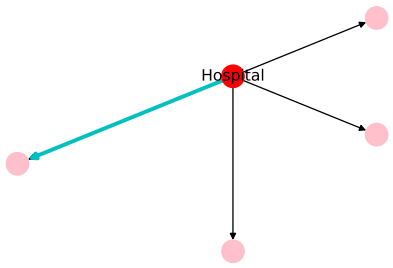


Figura 12: *Distibución de ayuda*

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/networkx-1.10/reference/drawing.html`.

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

[4] Graciela Wagner y Parra Rosa María Méndez Barrero, Alfredo Caicedo y de García. *Introducción a la Teoría de Grafos*. ELIZCOM SAS, 2010.

[5] Revenga Juana. *Flujo en Redes y Gestión de Proyectos. Teoría y Ejercicios Resueltos.* Netbiblo, 2008.

Tarea 2

En esta práctica se realizaron pequeñas correcciones ortográficas y la correcta forma de citar. Aún así se muestra la versión corregida de la tarea.

Visualización de grafos

Brenda Yaneth Sotelo Benítez
5705

26 de febrero de 2019

Este trabajo busca visualizar grafos utilizando un diferente algoritmo de acomodo (inglés: layout) para cada caso, con la implementación de la librería Networkx [1] y Matplotlib [2] de `Python` [3] para la generación de grafos y guardar imágenes en el formato *eps* respectivamente. El código empleado se obtuvo consultando documentación oficial [4] y los ejemplos mostrados se encuentran en [5]. Se utilizan arcos rojos para representar cuando un nodo tiene múltiples aristas y un nodo cuadrado verde para representar un lazo.

1. Grafo simple no dirigido acíclico

El algoritmo de acomodo utilizado para este ejemplo es el `kamada_kawai_layout` que toma en cuenta las distancias cuando posiciona un nodo. Se observa que el resultado de aplicarlo es bueno debido a que la cantidad de cruces es mínima, se mantiene la idea del ejemplo original y no es necesario hacer iteraciones. El resultado de la visualización se muestra en la figura 1.

```
1  G=nx.Graph()
2
3  A=[(1,2),(2,3),(3,4),(4,5),(2,6),(2,7),(3,8),(3,9),(4,10),(4,11)]
4  G.add_edges_from(A)
5
6  pos = nx.kamada_kawai_layout(G)
7
8  nx.draw_networkx_nodes(G,pos,nodelist=[1,2,3,4,5,6,7,8,9,10,11],
       node_color='gray', node_size=500)
9  nx.draw_networkx_edges(G,pos,width=3,edgelist=A, )
10
11 plt.axis('off')
12 plt.savefig("GNA.eps")
```

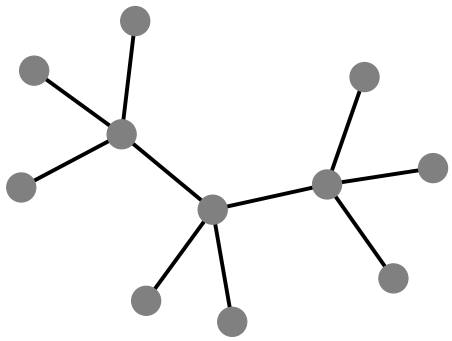


Figura 1: *Propano (C_3H_8)*

2. Grafo simple no dirigido cíclico

En la figura 2 se puede apreciar el resultado de aplicar el algoritmo de `fruchterman_reingold_layout` que al igual que `kamada_kawai_layout` son dirigidos por fuerzas, con la característica especial de la distribución de los nodos de forma homogénea. Se fijo un total de mil iteraciones para que el grafo se ajustara sin perder la forma debido a que con la ejecución de menos iteraciones el resultado es malo.

```
1  G=nx.Graph()
2
3  A=[(0,1),(1,2),(2,3),(3,4),(4,5),(0,5),(1,4),(0,6),(6,7)]
4
5  G.add_edges_from(A)
6  labels={2:'1',3:'2',4:'3',5:'4',0:'5',1:'6',6:'7',7:'8'}
7  pos= nx.fruchterman_reingold_layout(G, iterations=1000)
8
9  nx.draw_networkx_nodes(G,pos,nodelist=[0,1,2,3,4,5,6,7],node_color=
       'b', node_size=300)
10 nx.draw_networkx_edges(G,pos,width=1,edgelist=A,alpha=1)
11 nx.draw_networkx_labels(G,pos,labels, font_size=12, font_color='w'
       )
12
13 plt.axis('off')
```

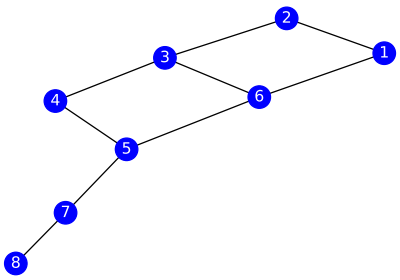


Figura 2: *Red de computadoras*

3. Grafo simple no dirigido reflexivo

El algoritmo `shell_layout` utilizado para la visualización del ejemplo de calles con retorno, tal y como se muestra en la figura 3 tiene buenos resultados a comparación del ejemplo original y de los otros diseños de acomodo utilizados, destacando que posiciona los nodos en círculos concéntricos y tiene la facilidad de elegir el centro del diseño. A continuación se muestra parte del código realizado donde también se hace uso de formas diferentes en los nodos y aristas.

```
1  G=nx.Graph()
2  A=[(0,1),(1,2),(2,3),(3,4)]
3  G.add_edges_from(A)
4  labels={1:'R'}
5  pos=nx.shell_layout(G)
6
7  nx.draw_networkx_nodes(G,pos,nodelist=[1],node_shape='s',
       node_color='g',node_size=500)
8  nx.draw_networkx_nodes(G,pos,nodelist=[3,4],node_color='b',
       node_size=80)
9  nx.draw_networkx_nodes(G,pos,nodelist=[0,2],node_color='b',
       node_size=80)
10 nx.draw_networkx_edges(G,pos,width=3,edgelist=[(2,3),(3,4)],alpha
       =1,style='dashed')
11 nx.draw_networkx_edges(G,pos,width=3,edgelist=[(0,1),(1,2)],alpha
       =1,style='dashed')
12 nx.draw_networkx_labels(G,pos,labels, front_size=12)
13 nx.draw_networkx_edge_labels(G, pos,font_family='sans-serif',
       edge_labels={(0,1):'Barragan',(1,2):'Gonzalitos',(2,3):'Ruiz_
       Cortines',(3,4):'Rangel_Frias'})
14
15 plt.axis('off')
16 plt.savefig("GNR.eps")
```

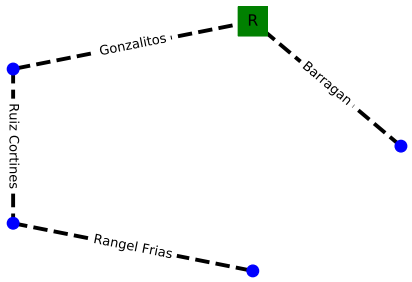


Figura 3: *Representación de calles con un retorno*

4. Grafo simple dirigido acíclico

El algoritmo de acomodo `circular_layout` como su nombre lo indica, tiene la característica de acomodar los nodos en círculo. Debido a esto, el ejemplo de secuencia de tareas que se muestra en la figura 4 tiene un buen ajuste en la representación e incluso mejor que el original. Este diseño es útil para crear grafos circulares con un número mayor de nodos.

```
1  G=nx.DiGraph()
2
3  A=[(0,1),(1,2),(2,3),(3,4),(3,5),(3,6),(4,7),(5,8),(8,6),(7,9)
       ,(9,10),(6,9),(1,3)]
4  labels={0:'1',1:'2',2:'3',3:'4',4:'5',5:'6',6:'9',7:'7',8:'8',9:'10
       ',10:'11'}
5  G.add_edges_from(A)
6
7  pos =  nx.circular_layout(G)
8
9  nx.draw_networkx_nodes(G,pos,nodelist=[0,1,2,3,4,5,6,7,8,9,10],
       node_color='m', node_size=450, alpha=0.8)
10 nx.draw_networkx_edges(G,pos,width=1.2,edgelist=A,alpha=1,arrowsize
       =20)
11 nx.draw_networkx_labels(G,pos,labels, front_size=12)
12
13 plt.axis('off')
14 plt.savefig("GDA.eps")
```

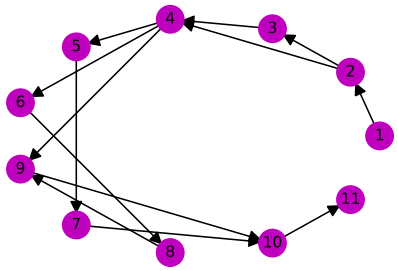


Figura 4: *Secuencia de tareas*

5. Grafo simple dirigido cíclico

Para el diseño del ejemplo de ruteo de vehículos se hizo uso del algoritmo `spring_layout` que puede recibir hasta once parámetros y utiliza el algoritmo de Fruchterman-Reingold. El resultado de cien iteraciones se muestra en la figura 5 dado que a menor número de iteraciones los nodos están muy dispersos y hay cruces de aristas y a mayor número de iteraciones los nodos están más cercanos, es por eso que se tomó una cantidad media de iteraciones.

```
1  G=nx.DiGraph()
2  A=[(0,1),(1,2),(2,3),(3,4),(4,0),(0,5),(5,6),(6,7),(7,8),(8,0)]
3  G.add_edges_from(A)
4  labels={0:'D'}
5  pos=nx.spring_layout(G, iterations=100)
6  nx.draw_networkx_nodes(G,pos,nodelist=[0],node_shape='s',node_color
       ='b', node_size=300)
7  nx.draw_networkx_nodes(G,pos,nodelist=[1,2,3,4],node_color='pink')
8  nx.draw_networkx_nodes(G,pos,nodelist=[5,6,7,8],node_color='c')
9  nx.draw_networkx_edges(G,pos,width=2,edgelist=A,alpha=1,arrowsize
       =25)
10 nx.draw_networkx_labels(G,pos,labels,front_size=12,font_color='w')
11
12 plt.axis('off')
13 plt.savefig("GDC.eps")
```

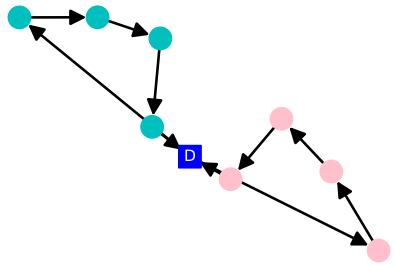


Figura 5: *VRP*

6. Grafo simple dirigido reflexivo

En la figura 6 se muestra el resultado de aplicar el algoritmo de acomodo `random_layout` que coloca los vértices al azar en un cuadrado al ejemplo de representación de una tienda proveedora que vende productos a varias sucursales pequeñas y que a su vez vende para si misma. Este algoritmo fue difícil de decidir donde aplicarlo ya que presenta una gran cantidad de cruces en las aristas por lo que se ejecutó el código varias veces para obtener la representacón que más se ajustara.

```
G=nx.DiGraph()
A=[(0,1),(0,2),(0,3),(0,4),(3,4)]
labels={0:'P',1:'T1',2:'T2',3:'T3',4:'T4'}
G.add_edges_from(A)
pos=nx.random_layout(G)

nx.draw_networkx_nodes(G,pos,nodelist=[1,2,3,4],node_color='c',
    node_size=500)
nx.draw_networkx_nodes(G,pos,nodelist=[0],node_color='g', node_size
    =1000,node_shape='s')
nx.draw_networkx_edges(G,pos,width=1,edgelist=A,alpha=1,arrowsize
    =20)
nx.draw_networkx_labels(G,pos,labels, front_size=12,font_color='w')

plt.axis('off')
plt.savefig("GDR.eps")
```

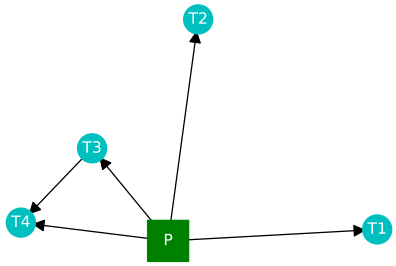


Figura 6: *Red de distribución*

7. Multigrafo no dirigido acíclico

El camino que sigue un río cuando pasa por ciertas localidades se puede representar mediante un multigrafo tal como se observa en figura 7. Esto debido a que un río suele dividirse cuando tiene de por medio obstáculos como tierra, piedras y se disparse en diferentes caminos beneficiando a varias comunidades. El algoritmo de acomodo que se ajusta a esta representación es `nx_pydot.pydot_layout` que se muestra en la figura 7.

```
1  G=nx.MultiGraph()
2  G.add_nodes_from([0,1,2,3,4,5,6])
3
4  pos = nx.nx_pydot.pydot_layout(G)
5
6  A=[(0,1),(1,2),(0,3),(2,4),(3,5),(2,6)]
7
8  nx.draw_networkx_nodes(G,pos,nodelist=[0,1,2,3,4,5,6],node_color='b
       ')
9  nx.draw_networkx_edges(G,pos,width=1,edgelist=A,alpha=1,edge_color=
       'c')
10 nx.draw_networkx_edges(G,pos,width=3,edgelist=[(0,3)],alpha=1,
       edge_color='r')
11 nx.draw_networkx_labels(G,pos,front_size=12)
12
13 plt.axis('off')
14 plt.savefig("MNA.eps")
15 plt.show()
```

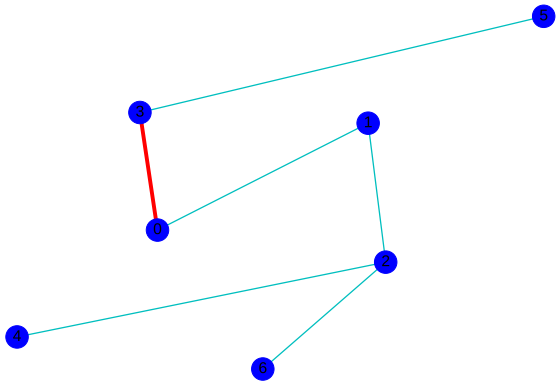


Figura 7: *Representación de un río*

8. Multigrafo no dirigido cíclico

Para la representación del ejemplo de los aeropuertos donde existen diversas aerolíneas y cada una de ellas cuenta con cierta cantidad de aviones que parten a su destino y regresan al aeropuerto, en la figura 8 se muestra el resultado de aplicar el algoritmo `bipartite_layout` que coloca los nodos en dos líneas rectas y se obtiene un buen resultado en comparación con el original. Es de utilidad para representar árboles.

```
1  G=nx.MultiGraph()
2  A=[(1,2),(2,3),(3,4),(3,1),(3,2),(2,4),(1,3)]
3  G.add_edges_from(A)
4  labels={1:'MÃ©xico', 3: 'Nayarit', 4: 'CancÃºn' }
5  labels1={2:'Monterrey'}
6  pos = nx.bipartite_layout(G,{1,4}, scale=0.2)
7
8  nx.draw_networkx_nodes(G,pos,nodelist=[1,2,3,4],node_color='b',
       node_shape='p')
9  nx.draw_networkx_edges(G,pos,width=1,edgelist=A,alpha=1)
10 nx.draw_networkx_edges(G,pos,width=3,edgelist=[(2,3)],alpha=1,
       edge_color='r',size=0.1)
11 pos1=pos
12 for i in pos:
13     pos1[i][1]=pos1[i][1]+0.04
14 nx.draw_networkx_labels(G,pos,labels, font_size=12)
15 for i in pos:
16     pos1[i][1]=pos1[i][1]-0.08
17 nx.draw_networkx_labels(G,pos,labels1, font_size=12)
```

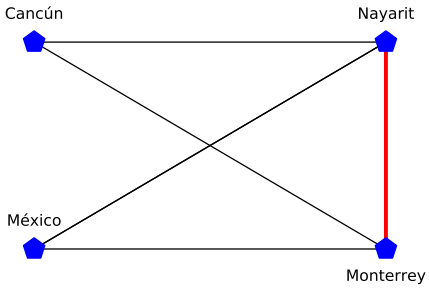


Figura 8: *Vuelos de aerolíneas*

9. Multigrafo no dirigido reflexivo

En la figura 9 se muestra el algoritmo `forceatlas2_layout` para el ejemplo de un inversionista que tiene a varios socios. Él puede decidir si invertir, que inviertan en él o invertir en si mismo, lo cual esto último crearía el lazo en el grafo.

```
G = nx.MultiDiGraph()
G.add_nodes_from(range(6))
labels={1:'I', 2:'S1',3:'S2',4:'S3',5:'S4'}
forceatlas2 = ForceAtlas2(
                          outboundAttractionDistribution=True,
                          linLogMode=False,
                          adjustSizes=False,
                          edgeWeightInfluence=1.0,

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

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

                          verbose=True)

pos = forceatlas2.forceatlas2_networkx_layout(G, pos=None,
    iterations=100)



nx.draw_networkx_nodes(G, pos, nodelist=[2,3,4,5], node_size=400,
    with_labels=True, node_color="m")
nx.draw_networkx_nodes(G, pos, nodelist=[1], node_size=400,
```

```
    with_labels=True, node_color="g")
28 nx.draw_networkx_edges(G, pos, edgelist=[(1,5)], edge_color="r")
29 nx.draw_networkx_edges(G, pos, edgelist=[(1,2),(1,3),(1,4)],
    edge_color="k")
30 nx.draw_networkx_labels(G,pos,labels, front_size=12, font_color='w'
    )
31
32 plt.axis('off')
33 plt.savefig('MNR.eps')
```

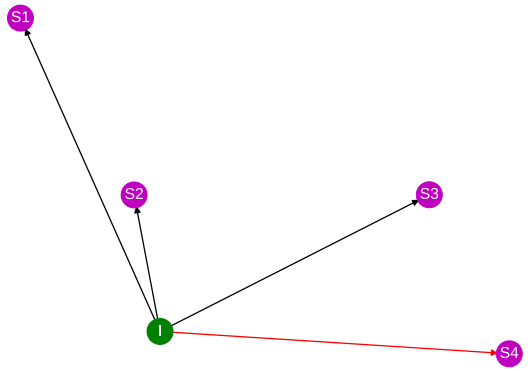


Figura 9: *Proceso de inversión*

10. Multigrafo dirigido acíclico

El algoritmo de acomodo utilizado para el ejemplo de los ductos de agua conectados de forma consecutiva sin formar ciclos es el `spectral_layout` que coloca los nodos utilizando vectores propios. De la misma forma que `random_layout` se presentó la dificultad de analizar en donde aplicarlo ya que los nodos y aristas se sobredibujaban y el grafo perdía forma.

```
1
2 A=[(1,2),(2,3),(3,4),(4,5)]
3 G.add_edges_from(A)
4
5 pos =nx.spectral_layout(G)
6
7 nx.draw_networkx_nodes(G,pos,nodelist=[1,2,3,4,5],node_color='b',
      node_size=300)
8 nx.draw_networkx_edges(G,pos,width=2,edgelist=A,alpha=1, edge_color
      ='c')
9 nx.draw_networkx_edges(G,pos,width=3,edgelist=[(1,2),(3,4)],alpha
      =1,edge_color='r')
10 nx.draw_networkx_edge_labels(G, pos,edge_labels={(1,2):'Ductos 1 y
      2',(2,3):'Ducto 3',(3,4):'Ductos 4 y 5',(4,5):'Ducto 6'})
11
12 plt.axis('off')
13 plt.savefig("MDA.eps")
```

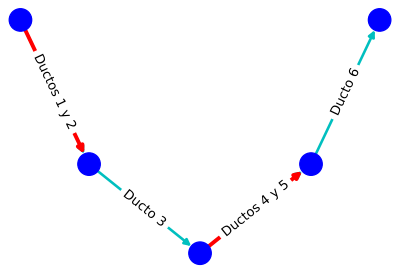


Figura 10: *Ductos de agua*

11. Multigrafo dirigido cíclico

En la figura 11 se aplica el algoritmo `shell_layout` en el problema representativo de calles o avenidas, donde para llegar a un lugar existen varias alternativas. El diseño muestra una buena visualización.

```
G=nx.MultiDiGraph()

A=[(1,2),(2,4),(4,3),(3,1),(5,2),(6,5),(7,5)]
G.add_edges_from(A)

pos =nx.shell_layout(G)

nx.draw_networkx_nodes(G,pos,nodelist=[1,2,3,4,5,6,7],node_color='y
    ', node_size=400,node_shape='^')
nx.draw_networkx_edges(G,pos,width=1,alpha=1,arrowsize=20)
nx.draw_networkx_edges(G,pos,width=3,edgelist=[(5,2)],alpha=1,
    edge_color='r', arrowsize=20)

plt.axis('off')
plt.savefig("MDC.eps")
```

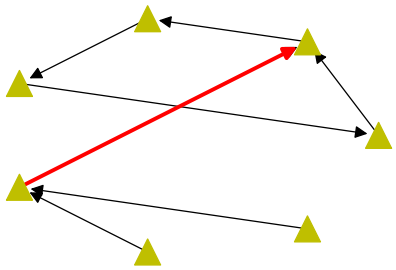


Figura 11: *Representación de avenidas*

12. Multigrafo dirigido reflexivo

En la figura 12 se plantea un modelo donde un hospital suministra medicinas a varias estaciones móviles, pero también se suministra el mismo. El algoritmo utilizado para la representación es el `nx_pydot.graphviz_layout`.

```
G=nx.DiGraph()
G.add_nodes_from(range(5))

V = nx.nx_pydot.graphviz_layout(G)

labels={0:'Hospital'}

nx.draw_networkx_nodes(G,V,nodelist=[1,2,3,4],node_color='pink')
nx.draw_networkx_nodes(G,V,nodelist=[0],node_color='r')
nx.draw_networkx_edges(G,V,width=1,edgelist=[(0,1),(0,2),(0,3)
    ,(0,4)],alpha=1)
nx.draw_networkx_edges(G,V,width=3,edgelist=[(0,3)],alpha=1,
    edge_color='c')
nx.draw_networkx_labels(G,V,labels, front_size=12)

plt.axis('off')
plt.savefig("MDR.eps")
```

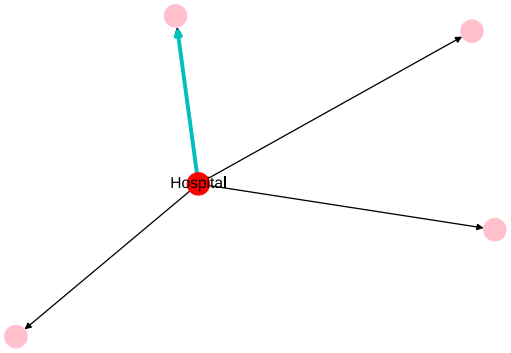


Figura 12: *Distribución de ayuda*

Referencias

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

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

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

[4] NetworkX developers con última actualización el 19 de Septiembre 2018. https://networkx.github.io/documentation/networkx-1.10/reference/drawing.html.

[5] Sotelo B. Repositorio optimización flujo en redes. https://github.com/BrendaSotelo/Flujo_Redes_BSotelo.

Visualización de grafos

Brenda Yaneth Sotelo Benítez
5705

3 de junio de 2019

Este trabajo busca visualizar grafos utilizando un diferente algoritmo de acomodo para cada caso, con la implementación de la librería Networkx [1] y Matplotlib [2] de `Python` [3] para la generación de grafos y guardar imágenes en el formato *eps* respectivamente. El código empleado se obtuvo consultando documentación oficial [4] y los ejemplos mostrados se encuentran en Sotelo [5]. Se utilizan arcos rojos para representar cuando un nodo tiene múltiples aristas y un nodo cuadrado verde para representar un lazo.

1. Grafo simple no dirigido acíclico

El algoritmo de acomodo utilizado para este ejemplo es el `kamada_kawai_layout` que toma en cuenta las distancias cuando posiciona un nodo. Se observa que el resultado de aplicarlo es bueno debido a que la cantidad de cruces es mínima, se mantiene la idea del ejemplo original y no es necesario hacer iteraciones. El resultado de la visualización se muestra en la figura 1.

```
1  G=nx.Graph()
2
3  A=[(1,2),(2,3),(3,4),(4,5),(2,6),(2,7),(3,8),(3,9),(4,10),(4,11)]
4  G.add_edges_from(A)
5
6  pos = nx.kamada_kawai_layout(G)
7
8  nx.draw_networkx_nodes(G,pos,nodelist=[1,2,3,4,5,6,7,8,9,10,11],
       node_color='gray', node_size=500)
9  nx.draw_networkx_edges(G,pos,width=3,edgelist=A, )
10
11 plt.axis('off')
12 plt.savefig("GNA.eps")
```

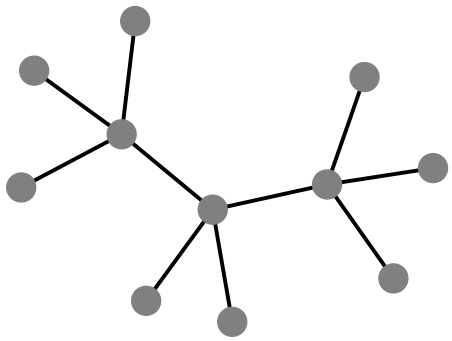


Figura 1: *Propano (C_3H_8)*

2. Grafo simple no dirigido cíclico

En la figura 2 se puede apreciar el resultado de aplicar el algoritmo de `fruchterman_reingold_layout` que al igual que `kamada_kawai_layout` son dirigidos por fuerzas, con la característica especial de la distribución de los nodos de forma homogénea. Se fijo un total de mil iteraciones para que el grafo se ajustara sin perder la forma debido a que con la ejecución de menos iteraciones el resultado es malo.

```
1  G=nx.Graph()
2
3  A=[(0,1),(1,2),(2,3),(3,4),(4,5),(0,5),(1,4),(0,6),(6,7)]
4
5  G.add_edges_from(A)
6  labels={2:'1',3:'2',4:'3',5:'4',0:'5',1:'6',6:'7',7:'8'}
7  pos= nx.fruchterman_reingold_layout(G, iterations=1000)
8
9  nx.draw_networkx_nodes(G,pos,nodelist=[0,1,2,3,4,5,6,7],node_color=
       'b', node_size=300)
10 nx.draw_networkx_edges(G,pos,width=1,edgelist=A,alpha=1)
11 nx.draw_networkx_labels(G,pos,labels, front_size=12, font_color='w'
       )
12
13 plt.axis('off')
```

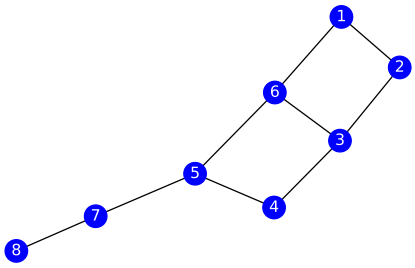


Figura 2: *Red de computadoras*

3. Grafo simple no dirigido reflexivo

El algoritmo `shell_layout` utilizado para la visualización del ejemplo de calles con retorno, tal y como se muestra en la figura 3 tiene buenos resultados a comparación del ejemplo original y de los otros diseños de acomodo utilizados, destacando que posiciona los nodos en círculos concéntricos y tiene la facilidad de elegir el centro del diseño. A continuación se muestra parte del código realizado donde también se hace uso de formas diferentes en los nodos y aristas.

```
1  G=nx.Graph()
2  A=[(0,1),(1,2),(2,3),(3,4)]
3  G.add_edges_from(A)
4  labels={1:'R'}
5  pos=nx.shell_layout(G)
6
7  nx.draw_networkx_nodes(G,pos,nodelist=[1],node_shape='s',
       node_color='g',node_size=500)
8  nx.draw_networkx_nodes(G,pos,nodelist=[3,4],node_color='b',
       node_size=80)
9  nx.draw_networkx_nodes(G,pos,nodelist=[0,2],node_color='b',
       node_size=80)
10 nx.draw_networkx_edges(G,pos,width=3,edgelist=[(2,3),(3,4)],alpha
       =1,style='dashed')
11 nx.draw_networkx_edges(G,pos,width=3,edgelist=[(0,1),(1,2)],alpha
       =1,style='dashed')
12 nx.draw_networkx_labels(G,pos,labels, front_size=12)
13 nx.draw_networkx_edge_labels(G, pos,font_family='sans-serif',
       edge_labels={(0,1):'Barragan',(1,2):'Gonzalitos',(2,3):'Ruiz_
       Cortines',(3,4):'Rangel_Frias'})
14
15 plt.axis('off')
16 plt.savefig("GNR.eps")
```

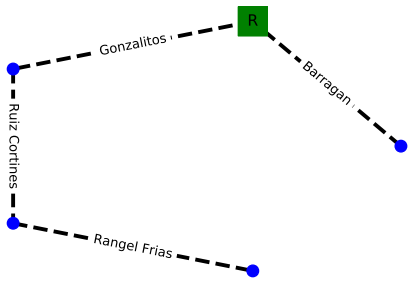


Figura 3: *Representación de calles con un retorno*

4. Grafo simple dirigido acíclico

El algoritmo de acomodo `circular_layout` como su nombre lo indica, tiene la característica de acomodar los nodos en círculo. Debido a esto, el ejemplo de secuencia de tareas que se muestra en la figura 4 tiene un buen ajuste en la representación e incluso mejor que el original. Este diseño es útil para crear grafos circulares con un número mayor de nodos.

```
1  G=nx.DiGraph()
2
3  A=[(0,1),(1,2),(2,3),(3,4),(3,5),(3,6),(4,7),(5,8),(8,6),(7,9)
       ,(9,10),(6,9),(1,3)]
4  labels={0:'1',1:'2',2:'3',3:'4',4:'5',5:'6',6:'9',7:'7',8:'8',9:'10
       ',10:'11'}
5  G.add_edges_from(A)
6
7  pos =  nx.circular_layout(G)
8
9  nx.draw_networkx_nodes(G,pos,nodelist=[0,1,2,3,4,5,6,7,8,9,10],
       node_color='m', node_size=450, alpha=0.8)
10 nx.draw_networkx_edges(G,pos,width=1.2,edgelist=A,alpha=1,arrowsize
       =20)
11 nx.draw_networkx_labels(G,pos,labels, front_size=12)
12
13 plt.axis('off')
14 plt.savefig("GDA.eps")
```

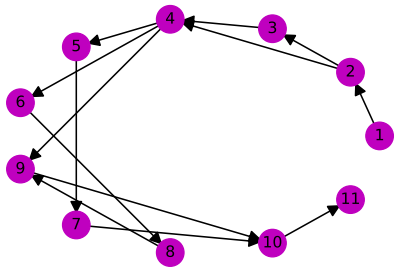


Figura 4: *Secuencia de tareas*

5. Grafo simple dirigido cíclico

Para el diseño del ejemplo de ruteo de vehículos se hizo uso del algoritmo `spring_layout` que puede recibir hasta once parámetros y utiliza el algoritmo de Fruchterman-Reingold. El resultado de cien iteraciones se muestra en la figura 5 dado que a menor número de iteraciones los nodos están muy dispersos y hay cruces de aristas y a mayor número de iteraciones los nodos están más cercanos, es por eso que se tomó una cantidad media de iteraciones.

```
1  G=nx.DiGraph()
2  A=[(0,1),(1,2),(2,3),(3,4),(4,0),(0,5),(5,6),(6,7),(7,8),(8,0)]
3  G.add_edges_from(A)
4  labels={0:'D'}
5  pos=nx.spring_layout(G, iterations=100)
6  nx.draw_networkx_nodes(G,pos,nodelist=[0],node_shape='s',node_color
       ='b', node_size=300)
7  nx.draw_networkx_nodes(G,pos,nodelist=[1,2,3,4],node_color='pink')
8  nx.draw_networkx_nodes(G,pos,nodelist=[5,6,7,8],node_color='c')
9  nx.draw_networkx_edges(G,pos,width=2,edgelist=A,alpha=1,arrowsize
       =25)
10 nx.draw_networkx_labels(G,pos,labels,front_size=12,font_color='w')
11
12 plt.axis('off')
13 plt.savefig("GDC.eps")
```

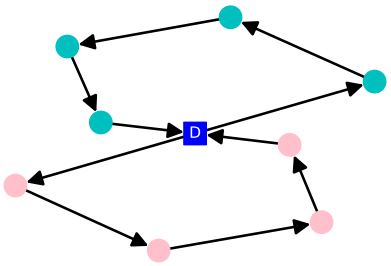


Figura 5: *VRP*

6. Grafo simple dirigido reflexivo

En la figura 6 se muestra el resultado de aplicar el algoritmo de acomodo `random_layout` que coloca los vértices al azar en un cuadrado al ejemplo de representación de una tienda proveedora que vende productos a varias sucursales pequeñas y que a su vez vende para si misma. Este algoritmo fue difícil de decidir donde aplicarlo ya que presenta una gran cantidad de cruces en las aristas por lo que se ejecutó el código varias veces para obtener la representacón que más se ajustara.

```
G=nx.DiGraph()
A=[(0,1),(0,2),(0,3),(0,4),(3,4)]
labels={0:'P',1:'T1',2:'T2',3:'T3',4:'T4'}
G.add_edges_from(A)
pos=nx.random_layout(G)

nx.draw_networkx_nodes(G,pos,nodelist=[1,2,3,4],node_color='c',
    node_size=500)
nx.draw_networkx_nodes(G,pos,nodelist=[0],node_color='g', node_size
    =1000,node_shape='s')
nx.draw_networkx_edges(G,pos,width=1,edgelist=A,alpha=1,arrowsize
    =20)
nx.draw_networkx_labels(G,pos,labels, front_size=12,font_color='w')

plt.axis('off')
plt.savefig("GDR.eps")
```

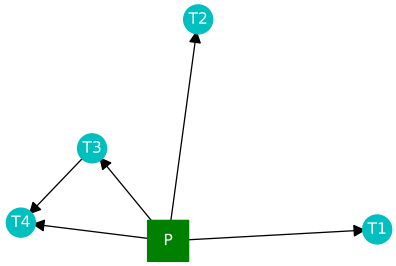


Figura 6: *Red de distribución*

7. Multigrafo no dirigido acíclico

El camino que sigue un río cuando pasa por ciertas localidades se puede representar mediante un multigrafo tal como se observa en figura 7. Esto debido a que un río suele dividirse cuando tiene de por medio obstáculos como tierra, piedras y se disparse en diferentes caminos beneficiando a varias comunidades. El algoritmo de acomodo que se ajusta a esta representación es `nx_pydot.pydot_layout` que se muestra en la figura 7.

```
1  G=nx.MultiGraph()
2  G.add_nodes_from([0,1,2,3,4,5,6])
3
4  pos = nx.nx_pydot.pydot_layout(G)
5
6  A=[(0,1),(1,2),(0,3),(2,4),(3,5),(2,6)]
7
8  nx.draw_networkx_nodes(G,pos,nodelist=[0,1,2,3,4,5,6],node_color='b
       ')
9  nx.draw_networkx_edges(G,pos,width=1,edgelist=A,alpha=1,edge_color=
       'c')
10 nx.draw_networkx_edges(G,pos,width=3,edgelist=[(0,3)],alpha=1,
       edge_color='r')
11 nx.draw_networkx_labels(G,pos,front_size=12)
12
13 plt.axis('off')
14 plt.savefig("MNA.eps")
15 plt.show()
```

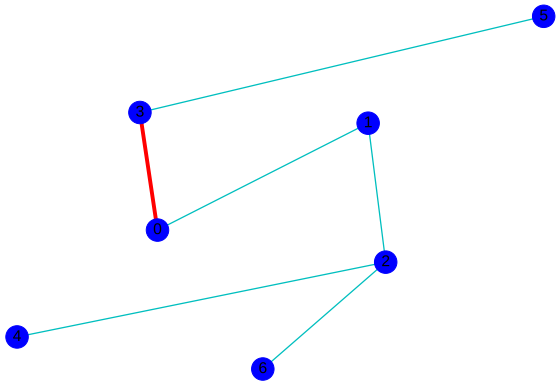


Figura 7: *Representación de un río*

8. Multigrafo no dirigido cíclico

Para la representación del ejemplo de los aeropuertos donde existen diversas aerolíneas y cada una de ellas cuenta con cierta cantidad de aviones que parten a su destino y regresan al aeropuerto, en la figura 8 se muestra el resultado de aplicar el algoritmo `bipartite_layout` que coloca los nodos en dos líneas rectas y se obtiene un buen resultado en comparación con el original. Es de utilidad para representar árboles.

```
1  G=nx.MultiGraph()
2  A=[(1,2),(2,3),(3,4),(3,1),(3,2),(2,4),(1,3)]
3  G.add_edges_from(A)
4  labels={1:'MÃ©xico', 3: 'Nayarit', 4: 'CancÃºn' }
5  labels1={2:'Monterrey'}
6  pos = nx.bipartite_layout(G,{1,4}, scale=0.2)
7
8  nx.draw_networkx_nodes(G,pos,nodelist=[1,2,3,4],node_color='b',
       node_shape='p')
9  nx.draw_networkx_edges(G,pos,width=1,edgelist=A,alpha=1)
10 nx.draw_networkx_edges(G,pos,width=3,edgelist=[(2,3)],alpha=1,
       edge_color='r',size=0.1)
11 pos1=pos
12 for i in pos:
13     pos1[i][1]=pos1[i][1]+0.04
14 nx.draw_networkx_labels(G,pos,labels, font_size=12)
15 for i in pos:
16     pos1[i][1]=pos1[i][1]-0.08
17 nx.draw_networkx_labels(G,pos,labels1, font_size=12)
```

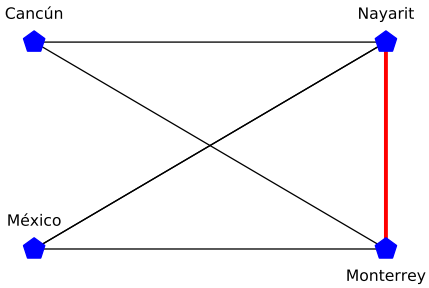


Figura 8: *Vuelos de aerolíneas*

9. Multigrafo no dirigido reflexivo

En la figura 9 se muestra el algoritmo `forceatlas2_layout` para el ejemplo de un inversionista que tiene a varios socios. Él puede decidir si invertir, que inviertan en él o invertir en si mismo, lo cual esto último crearía el lazo en el grafo.

```
G = nx.MultiDiGraph()
G.add_nodes_from(range(6))
labels={1:'I', 2:'S1',3:'S2',4:'S3',5:'S4'}
forceatlas2 = ForceAtlas2(
                          outboundAttractionDistribution=True,
                          linLogMode=False,
                          adjustSizes=False,
                          edgeWeightInfluence=1.0,

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

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

                          verbose=True)

pos = forceatlas2.forceatlas2_networkx_layout(G, pos=None,
    iterations=100)



nx.draw_networkx_nodes(G, pos, nodelist=[2,3,4,5], node_size=400,
    with_labels=True, node_color="m")
nx.draw_networkx_nodes(G, pos, nodelist=[1], node_size=400,
```

```
    with_labels=True, node_color="g")
nx.draw_networkx_edges(G, pos, edgelist=[(1,5)], edge_color="r")
nx.draw_networkx_edges(G, pos, edgelist=[(1,2),(1,3),(1,4)],
    edge_color="k")
nx.draw_networkx_labels(G,pos,labels, front_size=12, font_color='w'
    )

plt.axis('off')
plt.savefig('MNR.eps')
```

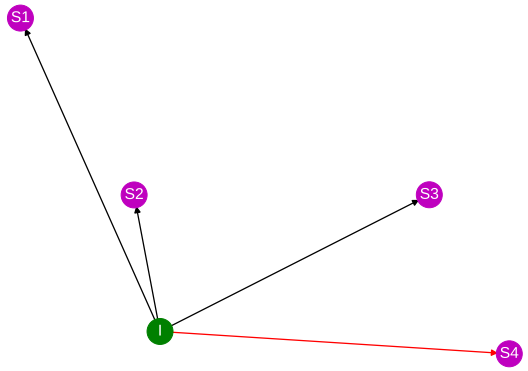


Figura 9: *Proceso de inversión*

10. Multigrafo dirigido acíclico

El algoritmo de acomodo utilizado para el ejemplo de los ductos de agua conectados de forma consecutiva sin formar ciclos es el `spectral_layout` que coloca los nodos utilizando vectores propios. De la misma forma que con `random_layout` se presentó la dificultad de analizar en donde aplicarlo ya que los nodos y aristas se sobredibujaban y el grafo perdía forma.

```
1
2 A=[(1,2),(2,3),(3,4),(4,5)]
3 G.add_edges_from(A)
4
5 pos =nx.spectral_layout(G)
6
7 nx.draw_networkx_nodes(G,pos,nodelist=[1,2,3,4,5],node_color='b',
      node_size=300)
8 nx.draw_networkx_edges(G,pos,width=2,edgelist=A,alpha=1, edge_color
      ='c')
9 nx.draw_networkx_edges(G,pos,width=3,edgelist=[(1,2),(3,4)],alpha
      =1,edge_color='r')
10 nx.draw_networkx_edge_labels(G, pos,edge_labels={(1,2):'Ductos 1 y
      2',(2,3):'Ducto 3',(3,4):'Ductos 4 y 5',(4,5):'Ducto 6'})
11
12 plt.axis('off')
13 plt.savefig("MDA.eps")
```

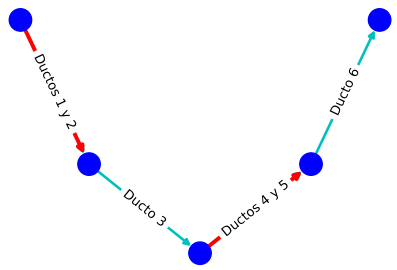


Figura 10: *Ductos de agua*

11. Multigrafo dirigido cíclico

En la figura 11 se aplica el algoritmo `shell_layout` en el problema representativo de calles o avenidas, donde para llegar a un lugar existen varias alternativas. El diseño muestra una buena visualización.

```
G=nx.MultiDiGraph()

A=[(1,2),(2,4),(4,3),(3,1),(5,2),(6,5),(7,5)]
G.add_edges_from(A)

pos =nx.shell_layout(G)

nx.draw_networkx_nodes(G,pos,nodelist=[1,2,3,4,5,6,7],node_color='y
    ', node_size=400,node_shape='^')
nx.draw_networkx_edges(G,pos,width=1,alpha=1,arrowsize=20)
nx.draw_networkx_edges(G,pos,width=3,edgelist=[(5,2)],alpha=1,
    edge_color='r', arrowsize=20)

plt.axis('off')
plt.savefig("MDC.eps")
```

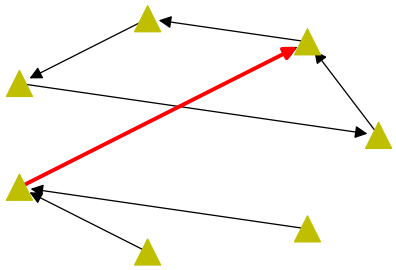


Figura 11: *Representación de avenidas*

12. Multigrafo dirigido reflexivo

En la figura 12 se plantea un modelo donde un hospital suministra medicinas a varias estaciones móviles, pero también se suministra el mismo. El algoritmo utilizado para la representación es el `nx_pydot.graphviz_layout`.

```
1  G=nx.DiGraph()
2  G.add_nodes_from(range(5))
3
4  V = nx.nx_pydot.graphviz_layout(G)
5
6  labels={0:'Hospital'}
7
8  nx.draw_networkx_nodes(G,V, nodelist=[1,2,3,4], node_color='pink')
9  nx.draw_networkx_nodes(G,V, nodelist=[0], node_color='r')
10 nx.draw_networkx_edges(G,V, width=1, edgelist=[(0,1),(0,2),(0,3)
       ,(0,4)], alpha=1)
11 nx.draw_networkx_edges(G,V, width=3, edgelist=[(0,3)], alpha=1,
       edge_color='c')
12 nx.draw_networkx_labels(G,V, labels ,  front_size=12)
13
14 plt.axis('off')
15 plt.savefig("MDR.eps")
```

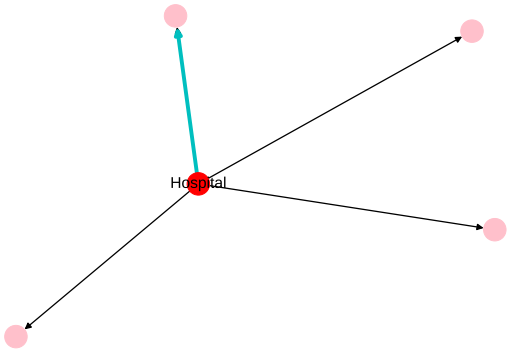


Figura 12: *Distribución de ayuda*

Referencias

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

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

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

[4] NetworkX developers con última actualización el 19 de Septiembre 2018. https://networkx.github.io/documentation/networkx-1.10/reference/drawing.html.

[5] Sotelo B. Repositorio optimización flujo en redes. https://github.com/BrendaSotelo/Flujo_Redes_BSotelo.

Tarea 3

En esta práctica se realizaron pequeñas correcciones ortográficas y en las tablas, donde se ajusto la orientación de los valores a la derecha y que tuvieran todos la misma cantidad de decimales. Aún así se muestra la versión corregida de la tarea.

Medición de tiempo de ejecución

Brenda Yaneth Sotelo Benítez
5705

19 de marzo de 2019

En este trabajo se realiza la medición de tiempo de ejecución de cinco algoritmos con cinco grafos distintos, con la implementación de la librería Networkx [1] y Matplotlib [2] de `Python` [3] para la generación de grafos y guardar imágenes en el formato *eps* respectivamente. El código empleado se obtuvo consultando documentación oficial [4] y los grafos utilizados se encuentran en el repositorio de Sotelo [5] que fueron modificados para cumplir con los requerimientos que cada algoritmo tiene sobre sus datos de entrada.

Cada ejecución se repite una cantidad suficiente de veces para que el tiempo total de ejecución del conjunto de réplicas sea mayor a 5 segundos, posteriormente se repite la medición del conjunto de réplicas 30 veces en total.

Se presenta una breve descripción de cada algoritmo seleccionado, visualización de los grafos, el histograma de comparación de tiempos promedio para cada grafo, fragmentos relevantes de código y dos gráficas de dispersión.

Las pruebas se han realizado en una PC con procesador Intel Core i3 2.00 GHz y 8.00 GB de RAM.

Algoritmo 1

El algoritmo `maximum_flow` recibe como parámetros principales un grafo simple G dirigido o no dirigido, capacidad de los arcos, que en dado caso de no ser especificada se considera que tienen capacidad infinita, un nodo fuente s y un nodo sumidero t para el flujo.

Consiste en encontrar la cantidad máxima de flujo que puede circular desde s hasta t, por lo que devuelve como resultado el valor total de flujo máximo y el valor del flujo que pasó por cada arco.

La tabla 1 muestra la media y desviación estándar del tiempo de ejecución del conjunto de réplicas para cada grafo y en la figura 1 el histograma correspondiente al tipo de grafo y su tiempo promedio.

Grafo	Media	Desviación
1	5.5907	0.4611
2	7.5716	0.5386
3	12.8474	0.6974
4	9.6711	0.6482
5	5.0180	0.4612

Tabla 1: *Reporte de la media y desviación*

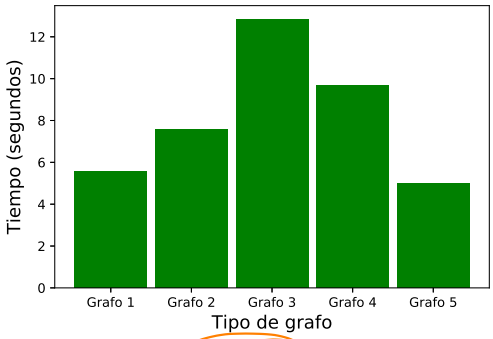
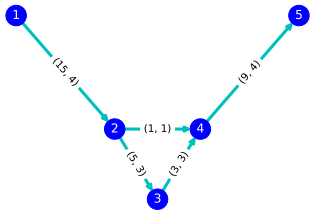


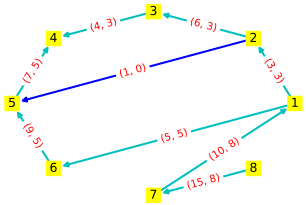
Figura 1: *Tiempo promedio para maximum_flow*

Se presenta a continuación en la figura 2 la solución para cada uno de los grafos donde en cada uno de ellos se muestra la ruta, la capacidad y el flujo que pasa a través de los arcos.

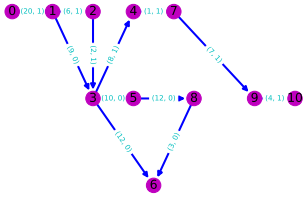
```
1 valor_flujo , flujo_max =nx.maximum_flow(G,1,5,capacity = 'peso')
```



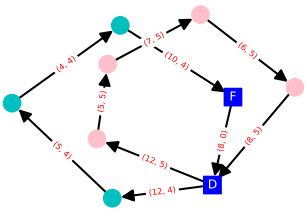
(a) Grafo 1



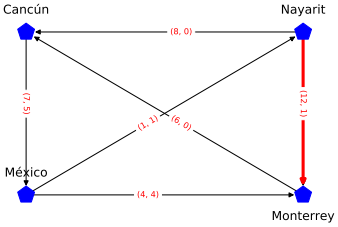
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 2: *Solución de maximum_flow*

Algoritmo 2

El algoritmo `all_shortest_paths` recibe como parámetros principales un grafo simple G dirigido o no dirigido, peso en los arcos, que en dado caso de no ser especificada se toma como peso, distancia o costo igual a 1 , un nodo inicial *source* y nodo final *target* para la ruta, asi como el método, es decir, el algoritmo que se utiliza para calcular las longitudes de la ruta: *dijkstra* o *bellman-ford*.

Consiste en buscar el camino más corto entre dos nodos, por lo que devuelve como resultado todas las rutas más cortas en el grafo.

La tabla 2 muestra la media y desviación estándar del tiempo de ejecución del conjunto de réplicas para cada grafo y en la figura 3 el histograma correspondiente al tipo de grafo y su tiempo promedio.

Grafo	Media	Desviación
1	6.0375	0.4586
2	10.5003	0.7112
3	14.9692	1.4172
4	10.6432	0.7121
5	9.3038	1.0171

Tabla 2: *Reporte de la media y desviación*

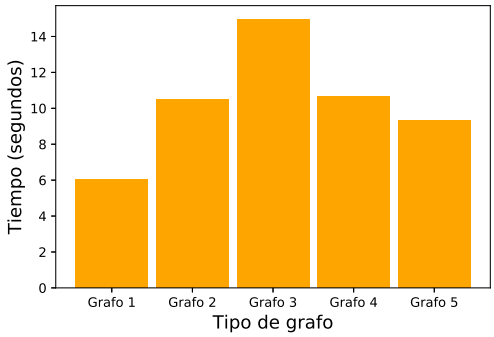
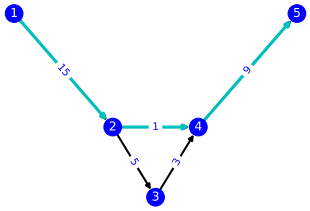


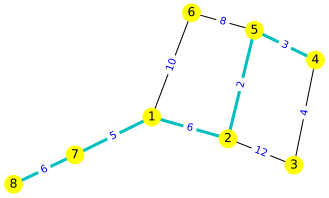
Figura 3: *Tiempo promedio para all_shortest_paths*

Se presenta a continuación en la figura 4 la solución para cada uno de los grafos, en cada uno de ellos se muestra la ruta más corta y el peso de cada uno de los arcos.

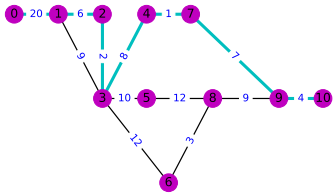
```
1 R = [p for p in nx.all_shortest_paths(G, source=1, target=5,weight=
      'peso')]
```



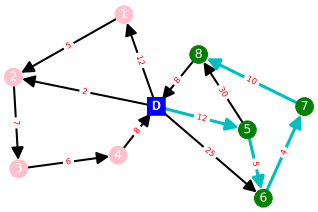
(a) Grafo 1



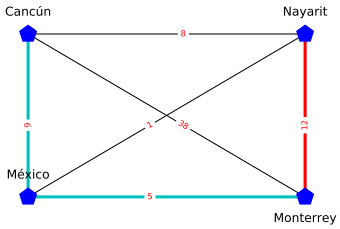
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 4: *Solución de all_shortest_paths*

Algoritmo 3

El algoritmo `minimum_spanning_tree` recibe como parámetros principales un grafo simple G dirigido o no dirigido, peso en los arcos, asi como el método, es decir, el algoritmo que se utiliza para encontrar un árbol de expansión mínima: *kruskal*, *bprim* o *boruvka*. El algoritmo predeterminado es *kruskal*.

Devuelve como resultado un árbol o bosque de expansión mínima en un grafo.

La tabla 3 muestra la media y desviación estándar del tiempo de ejecución del conjunto de réplicas para cada grafo y en la figura 5 el histograma correspondiente al tipo de grafo y su tiempo promedio.

Grafo	Media	Desviación
1	6.3368	0.352
2	7.2527	0.4135
3	9.9281	0.6668
4	8.3248	0.4654
5	4.3935	0.318

Tabla 3: *Reporte de la media y desviación*

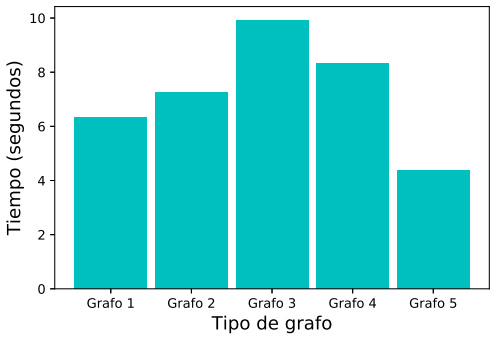
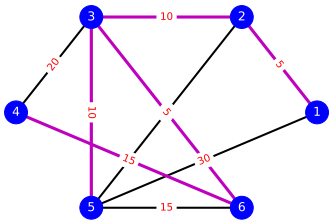


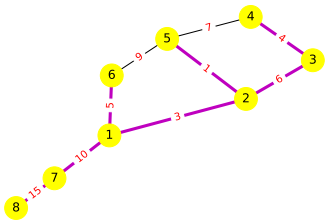
Figura 5: *Tiempo promedio para minimum_spanning_tree*

Se presenta a continuación en la figura 6 la solución para cada uno de los grafos, en cada uno de ellos se muestra el camino que sigue el árbol de expansión.

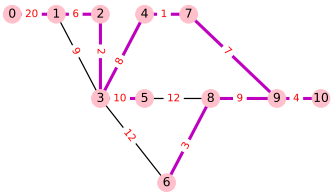
```
1 Tm = nx.minimum_spanning_tree(G, algorithm='kruskal')
```



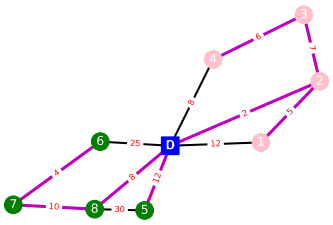
(a) Grafo 1



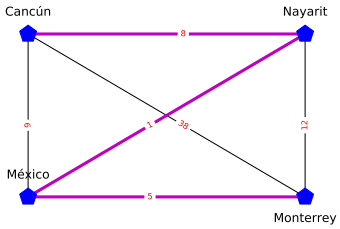
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 6: *Solución de minimum_spanning_tree*

Algoritmo 4

El algoritmo `greedy_color` recibe como parámetros principales un grafo simple G dirigido o no dirigido, una estrategia para el orden de coloración de los nodos y un algoritmo de intercambio.

Consiste en colorear un grafo con la menor cantidad de colores posible, donde para cada vecino de un nodo no puede tener el mismo color que el mismo.

La tabla 4 muestra la media y desviación estándar del tiempo de ejecución del conjunto de réplicas para cada grafo y en la figura 7 el histograma correspondiente al tipo de grafo y su tiempo promedio.

Grafo	Media	Desviación
1	3.4836	0.6221
2	5.3441	2.0836
3	6.2121	1.1141
4	5.331	1.4626
5	4.0002	1.6867

Tabla 4: *Reporte de la media y desviación*

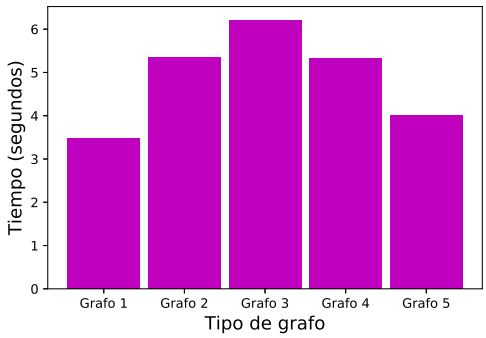
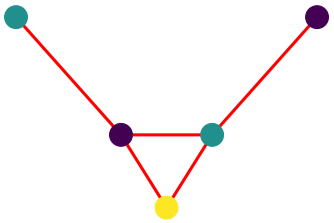


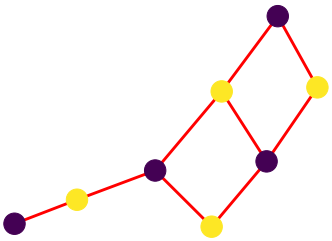
Figura 7: *Tiempo promedio para greedy_color*

Se presenta a continuación en la figura 8 la solución para cada uno de los grafos.

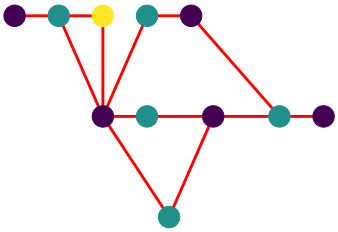
```
1        d = nx.coloring.greedy_color(G, strategy='largest_first')
```



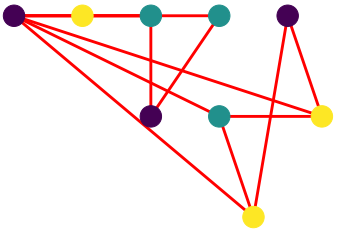
(a) Grafo 1



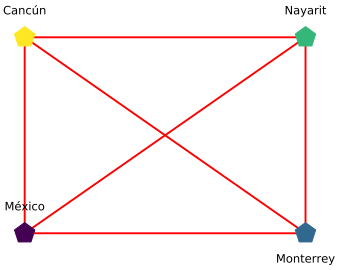
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 8: *Solución de greedy_color*

Algoritmo 5

El algoritmo `dfs_tree` recibe como parámetros principales un grafo simple G dirigido, un nodo fuente opcional para el inicio de la búsqueda en profundidad y la especificación del límite de búsqueda a profundidad.

Consiste en recorrer todos los nodos de un grafo comenzando de un nodo fuente y visitando a sus nodos adyacentes de tal manera que va formando un árbol a profundidad . Devuelve como resultado un árbol dirigido.

La tabla 5 muestra la media y desviación estándar del tiempo de ejecución del conjunto de réplicas para cada grafo y en la figura 9 el histograma correspondiente al tipo de grafo y su tiempo promedio.

Grafo	Media	Desviación
1	6.7802	0.4576
2	12.0012	1.4879
3	15.2492	0.6522
4	12.4167	1.4608
5	6.2011	0.5338

Tabla 5: *Reporte de la media y desviación*

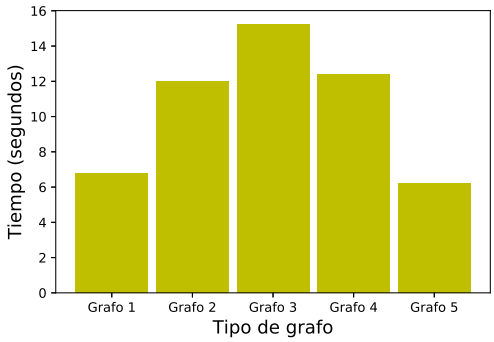
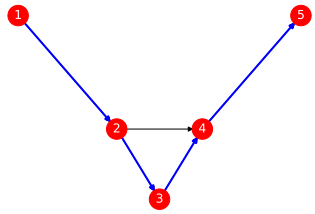


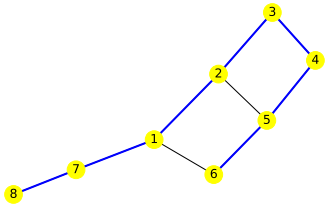
Figura 9: *Tiempo promedio para dfs_tree*

Se presenta a continuación en la figura 10 la solución para cada uno de los grafos, en cada uno de ellos se muestra el camino que sigue el algoritmo.

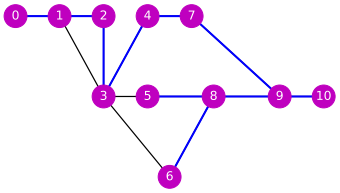
```
1        Tm = nx.dfs_tree(G, source=1)
```



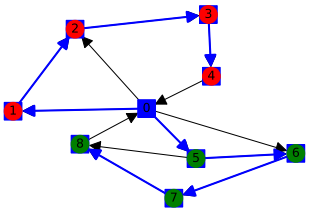
(a) Grafo 1



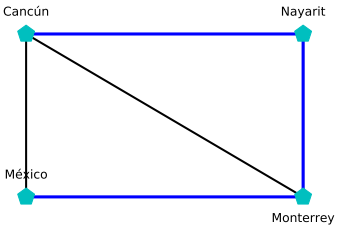
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 10: *Solución de dfs_tree*

En la figura 11 y figura 12 se muestra la gráfica de dispersión de cantidad de nodos y arcos contra el tiempo promedio de ejecución de cada uno de los grafos con los cinco tipos de algoritmos respectivamente. Cada forma de la gráfica representa un tipo de algoritmo.

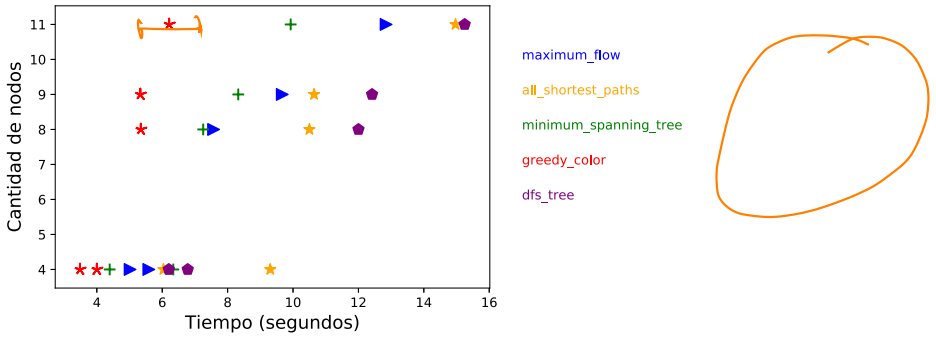


Figura 11: *Cantidad de nodos contra tiempos*

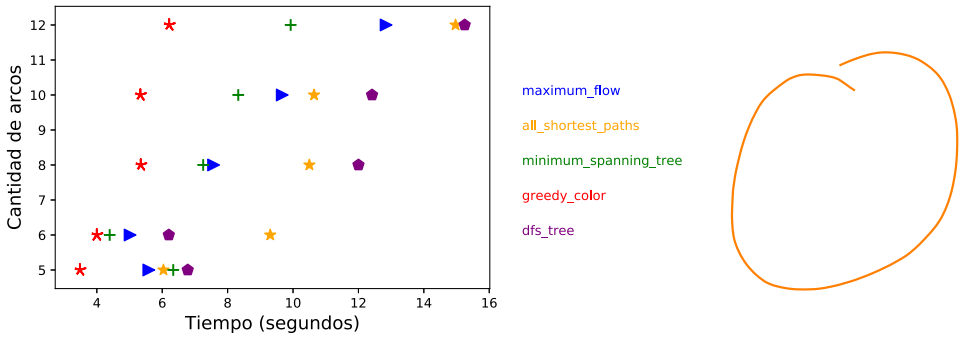


Figura 12: *Cantidad de arcos contra tiempos*

Referencias

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

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

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

[4] NetworkX developers con última actualización el 19 de Septiembre 2018. `https://networkx.github.io/documentation/networkx-1.10/reference/drawing.html`.

[5] Sotelo B. Repositorio optimización flujo en redes. `https://github.com/BrendaSotelo/Flujo_Redes_BSotelo`.

Complejidad asintótica experimental

Brenda Yaneth Sotelo Benítez
5705

3 de junio de 2019

En este trabajo se realiza la medición de tiempo de ejecución de cinco algoritmos con cinco grafos distintos, con la implementación de la librería Networkx [1] y Matplotlib [2] de `Python` [3] para la generación de grafos y guardar imágenes en el formato *eps* respectivamente. El código empleado se obtuvo consultando documentación oficial [4] y los grafos utilizados se encuentran en el repositorio de Sotelo [5] que fueron modificados para cumplir con los requerimientos que cada algoritmo tiene sobre sus datos de entrada.

Cada ejecución se repite una cantidad suficiente de veces para que el tiempo total de ejecución del conjunto de réplicas sea mayor a 5 segundos, posteriormente se repite la medición del conjunto de réplicas 30 veces en total.

Se presenta una breve descripción de cada algoritmo seleccionado, visualización de los grafos, el histograma de comparación de tiempos promedio para cada grafo, fragmentos relevantes de código y dos gráficas de dispersión.

Las pruebas se han realizado en una PC con procesador Intel Core i3 2.00 GHz y 8.00 GB de RAM.

Algoritmo 1

El algoritmo `maximum_flow` recibe como parámetros principales un grafo simple G dirigido o no dirigido, capacidad de los arcos, que en dado caso de no ser especificada se considera que tienen capacidad infinita, un nodo fuente s y un nodo sumidero t para el flujo.

Consiste en encontrar la cantidad máxima de flujo que puede circular desde s hasta t, por lo que devuelve como resultado el valor total de flujo máximo y el valor del flujo que pasó por cada arco.

La tabla 1 muestra la media y desviación estándar del tiempo de ejecución del conjunto de réplicas para cada grafo y en la figura 1 el histograma correspondiente al tipo de grafo y su tiempo promedio.

Grafo	**Media**	**Desviación**
1	5.5907	0.4611
2	7.5716	0.5386
3	12.8474	0.6974
4	9.6711	0.6482
5	5.0180	0.4612

Tabla 1: *Reporte de la media y desviación*

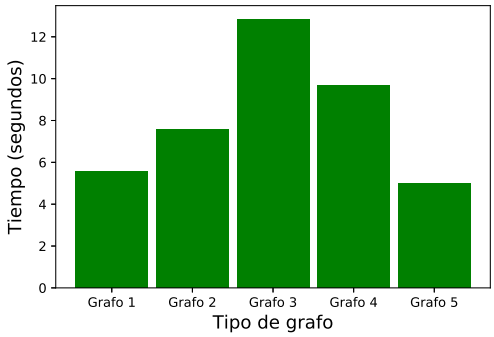
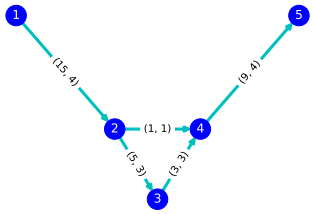


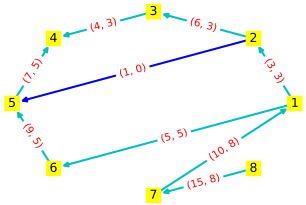
Figura 1: *Tiempo promedio para maximum_flow*

Se presenta a continuación en la figura 2 la solución para cada uno de los grafos donde en cada uno de ellos se muestra la ruta, la capacidad y el flujo que pasa a través de los arcos.

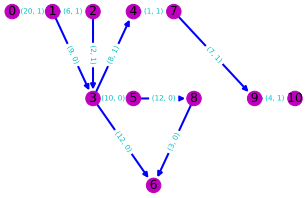
```
1 valor_flujo, flujo_max =nx.maximum_flow(G,1,5,capacity = 'peso')
```



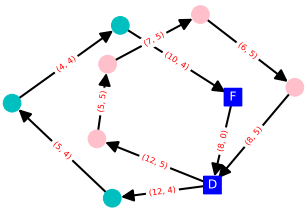
(a) Grafo 1



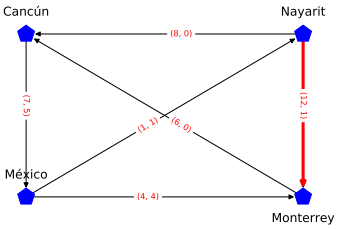
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 2: *Solución de maximum_flow*

Algoritmo 2

El algoritmo `all_shortest_paths` recibe como parámetros principales un grafo simple G dirigido o no dirigido, peso en los arcos, que en dado caso de no ser especificada se toma como peso, distancia o costo igual a 1 , un nodo inicial *source* y nodo final *target* para la ruta, asi como el método, es decir, el algoritmo que se utiliza para calcular las longitudes de la ruta: *dijkstra* o *bellman-ford*.

Consiste en buscar el camino más corto entre dos nodos, por lo que devuelve como resultado todas las rutas más cortas en el grafo.

La tabla 2 muestra la media y desviación estándar del tiempo de ejecución del conjunto de réplicas para cada grafo y en la figura 3 el histograma correspondiente al tipo de grafo y su tiempo promedio.

Grafo	Media	Desviación
1	6.0375	0.4586
2	10.5003	0.7112
3	14.9692	1.4172
4	10.6432	0.7121
5	9.3038	1.0171

Tabla 2: *Reporte de la media y desviación*

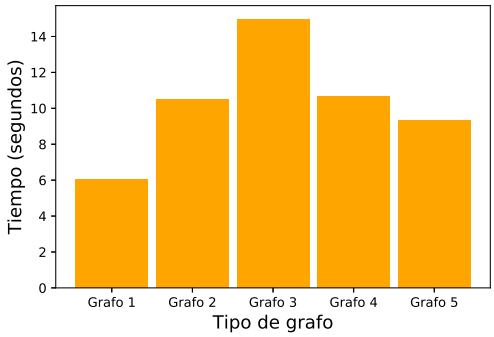
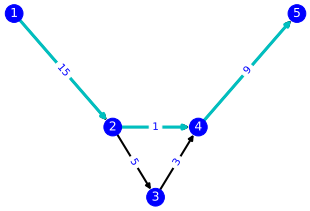


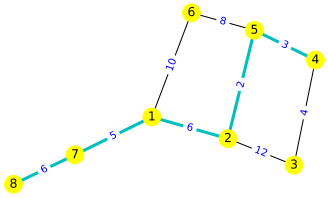
Figura 3: *Tiempo promedio para all_shortest_paths*

Se presenta a continuación en la figura 4 la solución para cada uno de los grafos, en cada uno de ellos se muestra la ruta más corta y el peso de cada uno de los arcos.

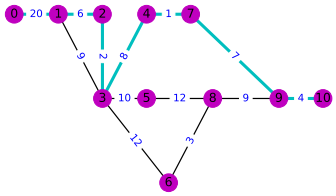
```
1 R = [p for p in nx.all_shortest_paths(G, source=1, target=5,weight=
      'peso')]
```



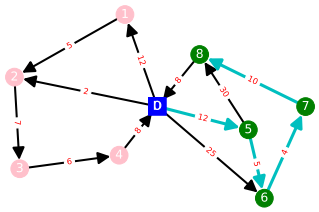
(a) Grafo 1



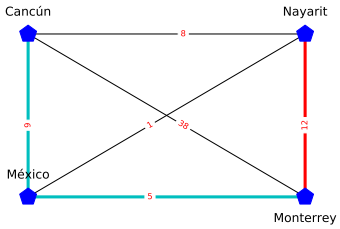
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 4: *Solución de all_shortest_paths*

Algoritmo 3

El algoritmo `minimum_spanning_tree` recibe como parámetros principales un grafo simple G dirigido o no dirigido, peso en los arcos, asi como el método, es decir, el algoritmo que se utiliza para encontrar un árbol de expansión mínima: *kruskal*, *bprim* o *boruvka*. El algoritmo predeterminado es *kruskal*.

Devuelve como resultado un árbol o bosque de expansión mínima en un grafo.

La tabla 3 muestra la media y desviación estándar del tiempo de ejecución del conjunto de réplicas para cada grafo y en la figura 5 el histograma correspondiente al tipo de grafo y su tiempo promedio.

Grafo	Media	Desviación
1	6.3368	0.352
2	7.2527	0.4135
3	9.9281	0.6668
4	8.3248	0.4654
5	4.3935	0.3181

Tabla 3: *Reporte de la media y desviación*

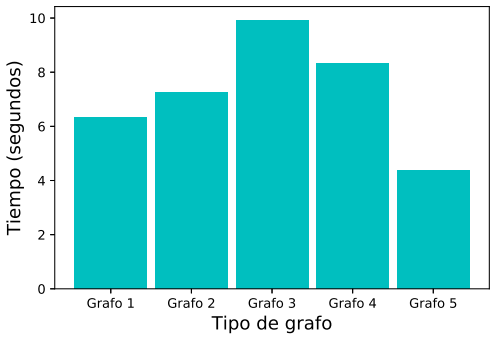
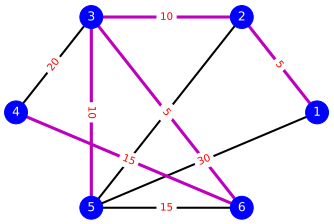


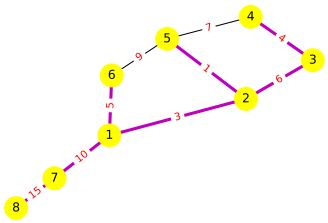
Figura 5: *Tiempo promedio para minimum_spanning_tree*

Se presenta a continuación en la figura 6 la solución para cada uno de los grafos, en cada uno de ellos se muestra el camino que sigue el árbol de expansión.

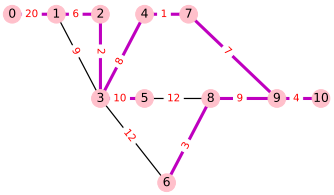
```
1 Tm = nx.minimum_spanning_tree(G, algorithm='kruskal')
```



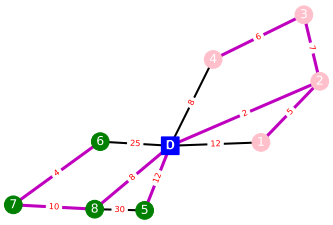
(a) Grafo 1



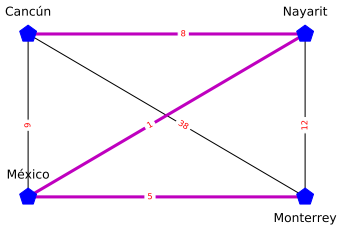
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 6: *Solución de minimum_spanning_tree*

Algoritmo 4

El algoritmo `greedy_color` recibe como parámetros principales un grafo simple G dirigido o no dirigido, una estrategia para el orden de coloración de los nodos y un algoritmo de intercambio.

Consiste en colorear un grafo con la menor cantidad de colores posible, donde para cada vecino de un nodo no puede tener el mismo color que el mismo.

La tabla 4 muestra la media y desviación estándar del tiempo de ejecución del conjunto de réplicas para cada grafo y en la figura 7 el histograma correspondiente al tipo de grafo y su tiempo promedio.

Grafo	Media	Desviación
1	3.4836	0.6221
2	5.3441	2.0836
3	6.2121	1.1141
4	5.3312	1.4626
5	4.0002	1.6867

Tabla 4: *Reporte de la media y desviación*

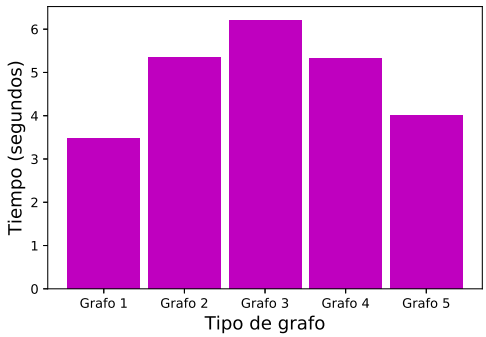
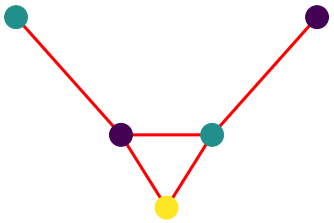


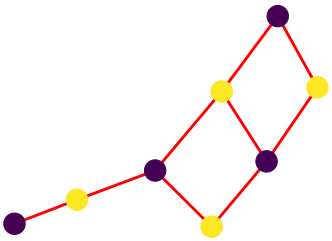
Figura 7: *Tiempo promedio para greedy_color*

Se presenta a continuación en la figura 8 la solución para cada uno de los grafos.

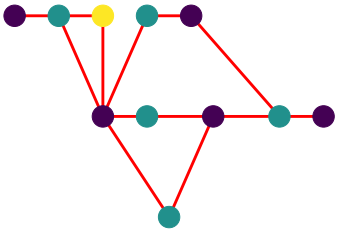
```
1        d = nx.coloring.greedy_color(G, strategy='largest_first')
```



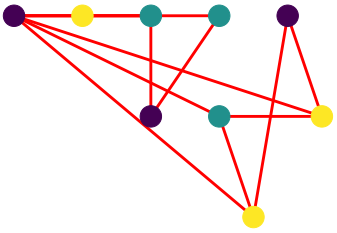
(a) Grafo 1



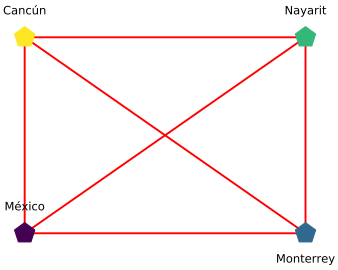
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 8: *Solución de greedy_color*

Algoritmo 5

El algoritmo `dfs_tree` recibe como parámetros principales un grafo simple G dirigido, un nodo fuente opcional para el inicio de la búsqueda en profundidad y la especificación del límite de búsqueda a profundidad.

Consiste en recorrer todos los nodos de un grafo comenzando de un nodo fuente y visitando a sus nodos adyacentes de tal manera que va formando un árbol a profundidad. Devuelve como resultado un árbol dirigido.

La tabla 5 muestra la media y desviación estándar del tiempo de ejecución del conjunto de réplicas para cada grafo y en la figura 9 el histograma correspondiente al tipo de grafo y su tiempo promedio.

Grafo	Media	Desviación
1	6.7802	0.4576
2	12.0012	1.4879
3	15.2492	0.6522
4	12.4167	1.4608
5	6.2011	0.5338

Tabla 5: *Reporte de la media y desviación*

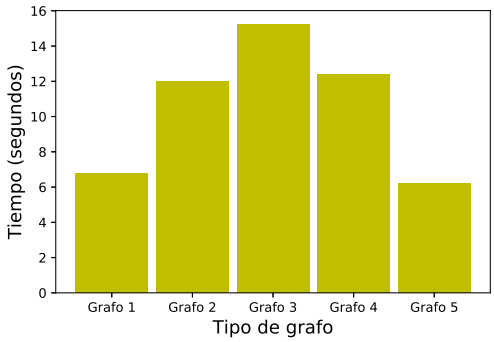
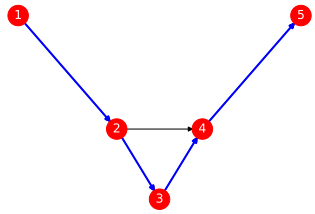


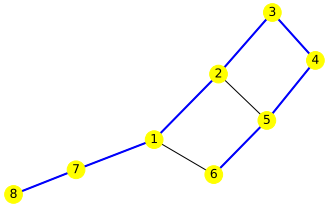
Figura 9: *Tiempo promedio para dfs_tree*

Se presenta a continuación en la figura 10 la solución para cada uno de los grafos, en cada uno de ellos se muestra el camino que sigue el algoritmo.

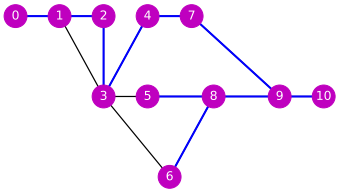
```
1        Tm = nx.dfs_tree(G, source=1)
```



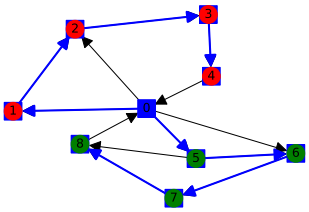
(a) Grafo 1



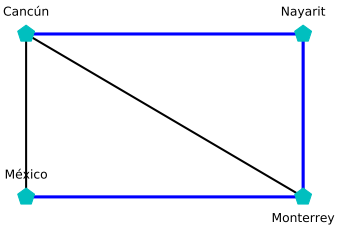
(b) Grafo 2



(c) Grafo 3



(d) Grafo 4



(e) Grafo 5

Figura 10: *Solución de dfs_tree*

En la figura 11 y figura 12 se muestra la gráfica de dispersión de cantidad de nodos y arcos contra el tiempo promedio de ejecución de cada uno de los grafos con los cinco tipos de algoritmos respectivamente. Cada forma de la gráfica representa un tipo de algoritmo.

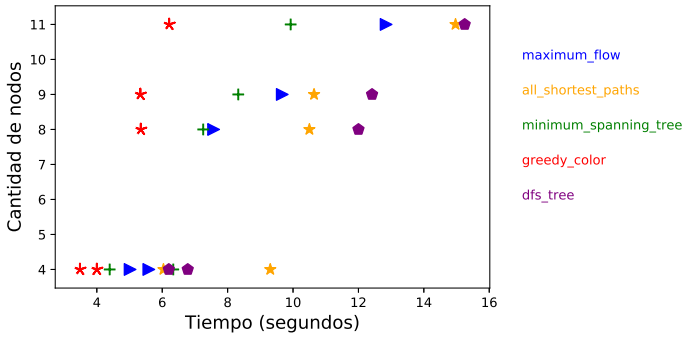


Figura 11: *Cantidad de nodos contra tiempos*

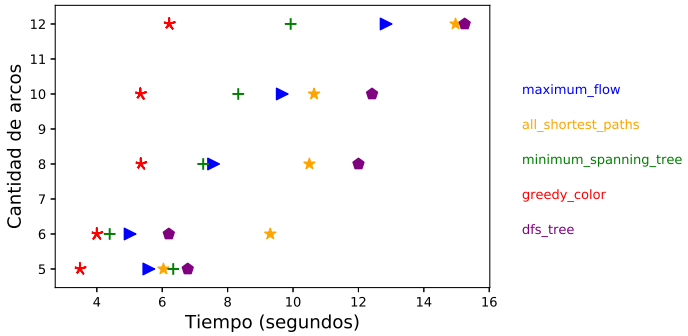


Figura 12: *Cantidad de arcos contra tiempos*

Referencias

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

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

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

[4] NetworkX developers con última actualización el 19 de Septiembre 2018. `https://networkx.github.io/documentation/networkx-1.10/reference/drawing.html`.

[5] Sotelo B. Repositorio optimización flujo en redes. `https://github.com/BrendaSotelo/Flujo_Redes_BSotelo`.

Tarea 4

En esta práctica se realizaron correcciones ortográficas y de ausencia de código.

Complejidad asintótica experimental

Brenda Yaneth Sotelo Benítez
5705

2 de abril de 2019

En este trabajo se realiza la medición de tiempo de ejecución de tres métodos de generación de grafos de orden 8, 16, 32, 64 a base 2 con tres implementaciones de algoritmos de flujo máximo. Se generaron diez grafos distintos de cada orden con la asignación de pesos positivos normalmente distribuidos a las aristas con media $\mu = 10$ y desviación estándar $\sigma = 3$ para poder ser utilizados en las instancias del problema de flujo máximo.

Los algoritmos seleccionados se ejecutan con cinco diferentes pares de fuente-sumidero y se obtuvieron por medio de la librería Networkx [1] y Matplotlib [2] de `Python` [3] para la generación de grafos y guardar imágenes en el formato *eps* respectivamente. Además se hace uso de la librería Numpy [4] para la suma de tiempos, calculo de tiempos promedios y desviaciones estándar y Pandas [5] para el tratamiento de datos. El código empleado se obtuvo consultando documentación oficial [6].

Se presenta una breve descripción de los algoritmos seleccionados, visualización de los grafos, fragmentos relevantes de código y una sección de resultados donde se muestra un análisis de varianza para saber si los efectos entre el algoritmo generador, algoritmo de flujo máximo, orden y la densidad del grafo con el tiempo de ejecución son estadísticamente significativos.

Las pruebas se han realizado en una PC con procesador Intel Core i3 2.00 GHz y 8.00 GB de RAM.

Métodos generadores de grafos

La elección de los siguientes métodos se debe a que la visualización de cada uno es interesante, sobre todo los que generan grafos n-dimensionales, por la aplicación práctica que tienen, simplicidad y variedad.

1. `Complete_graph`. Tiene como parámetros el número de nodos que se desea y el tipo de grafo, que toma por defecto un grafo simple no dirigido.

   La figura 1 muestra una visualización de este generador.

```
1 G = nx.complete_graph(30)
```

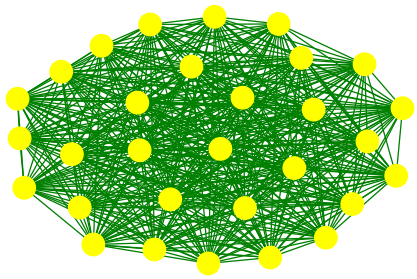


Figura 1: *Complete graph*

2. `Wheel_graph`. El grafo de la rueda tiene un nodo central que esta conectado a un ciclo de (n-1) nodos. Recibe como parámetros el número de nodos, el tipo de grafo que se desea crear, el cual tiene como opción predeterminada un grafo simple no dirigido. Las etiquetas de los nodos van de 0 a n-1.

   Se presenta la visualización del generador en la figura 2.

```
1 G = nx.wheel_graph(100)
```

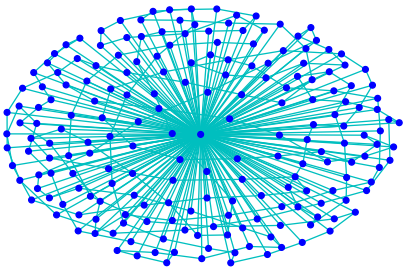


Figura 2: *Wheel graph*

3. `Grid_graph`. Recibe como principal parámetro la cantidad de nodos que puede ser una lista, tupla o un entero y devuelve un grafo de cuadrícula n-dimensional.

   La figura 3 muestra una visualización de este generador.

```
1 G = nx.grid_graph(dim=[8,8])
```

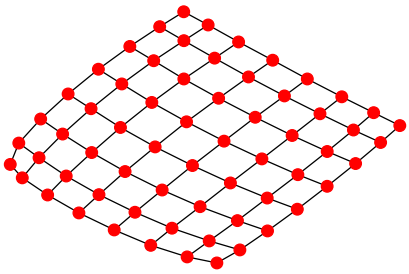


Figura 3: *Grid graph*

4. `Circular_ladder_graph`. Devuelve un grafo de escalera circular, que consiste en dos ciclos de n nodos en donde cada uno de los n pares de nodos se unen por una arista. Toma como dato de entrada la cantidad de nodos y las etiquetas de estos son los números enteros de 0 a n-1.

   La figura 4 muestra una visualización de este generador.

```
1 G = nx.circular_ladder_graph(20)
```

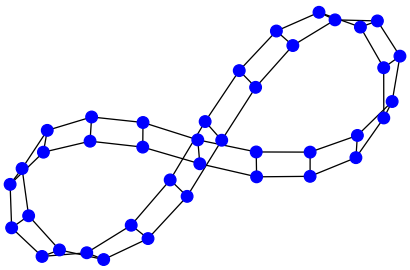


Figura 4: *Circular ladder graph*

Algoritmos de flujo máximo

La elección de los siguientes algoritmos se debe a la variedad que hay para calcular el valor del flujo de un grafo, que reciben parámetros similares y retornan diferentes datos.

1. `Maximum_flow_value`. Encuentra y devuelve el valor del flujo máximo donde cada arista debe tener una capacidad, que en caso de no tenerla toma por defecto una capacidad infinita, recibe un nodo fuente y un nodo sumidero y una función para calcular el flujo máximo. El algoritmo no es compatible con multigrafos.

```
flow_value = nx.maximum_flow_value(G, 1, 6)
```

2. `Minimum_cut`. Recibe como parámetros principales la capacidad y si este dato se omite se considera que tiene una capacidad infinita , nodo fuente y sumidero para el flujo y una función para calcular el flujo máximo. Devuelve el valor del corte mínimo y la partición de los nodos que definen el corte mínimo.

```
cut_value, partition = nx.minimum_cut(G, 1, 6)
```

3. `Minimum_cut_value`. Este algoritmo devuelve solamente el valor que tiene el corte mínimo. Tiene como datos de entrada el nodo fuente y nodo sumidero, una función para calcular el flujo máximo y la capacidad de las aristas que por defecto se considera que tiene una capacidad infinita.

```
cut_value = nx.minimum_cut_value(G, 1, 6)
```

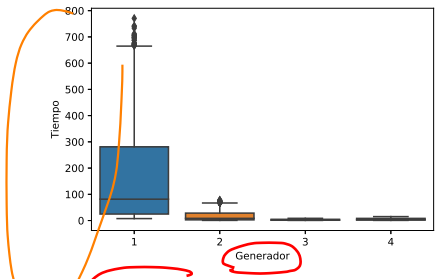
Resultados

Los resultados del análisis de varianza de los efectos simples se muestran en el cuadro 1, donde los factores que presentan una relación significativa con el tiempo de ejecución son la densidad, generador de grafo y orden, es decir, que tienen un p-valor menor a 0.05 (nivel de significancia) y para los que la hipótesis nula de que las medias son iguales se rechaza. Mientras que el factor algoritmo no es significativo con el tiempo de ejecución.

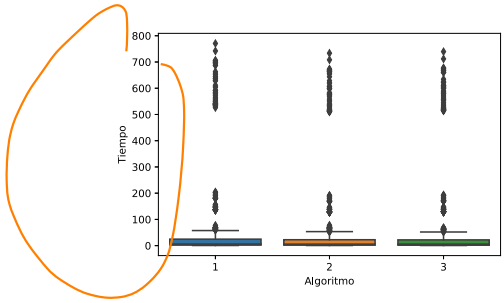
Cuadro 1: *Efectos simples*

Factor	**P-valor**
Algoritmo	0.316
Densidad	0.000
Generador	0.000
Orden	0.000

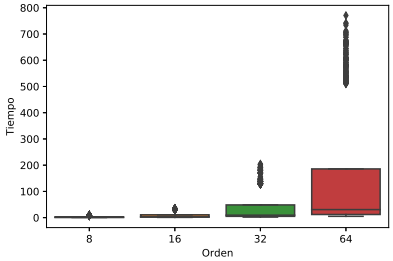
Por otro lado, en la figura 5 se muestran los diagramas de caja y bigotes para cada nivel de los factores donde se observa que el generador 1 es el que mayor tiempo consume a comparación del generador 2, los tres tipos de algoritmos parecen tener tiempos de ejecución similares, conforme va creciendo el orden de los grafos el tiempo de ejecución crece también y que a menor densidad el tiempo aumenta.



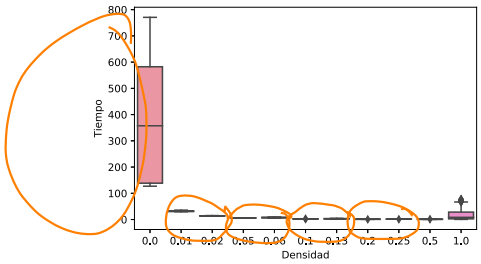
(a) Generador de grafo contra tiempo



(b) Algoritmo contra tiempo



(c) Orden del grafo contra tiempo



(d) Densidad del grafo contra tiempo

Figura 5: *Diagramas de caja y bigotes de los factores.*

Además, en el cuadro 2 se muestran los efectos de interacción para cada pareja de factor de los factores mencionados en el cuadro 1, donde se puede concluir que las interacciones generador-densidad, generador-orden y orden-densidad tienen relación significativa, es decir, que tienen un p-valor menor a 0.05 (nivel de significancia), por lo que la combinación que se tome de cada uno de los factores importa para el valor del tiempo de ejecución. Mientras que las interacciones

algoritmo-densidad, algoritmo-orden y generador-orden no tienen relación significativa con el tiempo dado que el p-valor es mayor a 0.05, por lo que no importa que combinación se tenga de los factores.

Cuadro 2: *Efectos de interacción*

Factor	P-valor
Algoritmo-Densidad	0.785
Algoritmo-Orden	0.417
Generador-Algoritmo	0.311
Generador-Densidad	0.000
Generador-Orden	0.000
Orden-Densidad	0.000

Referencias

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

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

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

[4] NumPy developers Versión 1.16.2. `https://networkx.github.io`.

[5] Augspurger T. and the pandas core team Versión 0.24.2. `https://pandas.pydata.org/`.

[6] NetworkX developers con última actualización el 19 de Septiembre 2018. `https://networkx.github.io/documentation/networkx-1.10/reference/drawing.html`.

Complejidad asintótica experimental

Brenda Yaneth Sotelo Benítez
5705

3 de junio de 2019

En este trabajo se realiza la medición de tiempo de ejecución de tres métodos de generación de grafos de orden 8, 16, 32, 64 a base 2 con tres implementaciones de algoritmos de flujo máximo. Se generaron diez grafos distintos de cada orden con la asignación de pesos positivos normalmente distribuidos a las aristas con media $\mu = 10$ y desviación estándar $\sigma = 3$ para poder ser utilizados en las instancias del problema de flujo máximo.

Los algoritmos seleccionados se ejecutan con cinco diferentes pares de fuente-sumidero y se obtuvieron por medio de la librería Networkx [1] y Matplotlib [2] de `Python` [3] para la generación de grafos y guardar imágenes en el formato *eps* respectivamente. Además se hace uso de la librería Numpy [4] para la suma de tiempos, cálculo de tiempos promedios y desviaciones estándar y Pandas [5] para el tratamiento de datos. El código empleado se obtuvo consultando documentación oficial [6].

Se presenta una breve descripción de los algoritmos seleccionados, visualización de los grafos, fragmentos relevantes de código y una sección de resultados donde se muestra un análisis de varianza para saber si los efectos entre el algoritmo generador, algoritmo de flujo máximo, orden y la densidad del grafo con el tiempo de ejecución son estadísticamente significativos.

Las pruebas se han realizado en una PC con procesador Intel Core i3 2.00 GHz y 8.00 GB de RAM.

Métodos generadores de grafos

La elección de los siguientes métodos se debe a que la visualización de cada uno es interesante, sobre todo los que generan grafos n-dimensionales, por la aplicación práctica que tienen, simplicidad y variedad.

1. `Complete_graph`. Tiene como parámetros el número de nodos que se desea y el tipo de grafo, que toma por defecto un grafo simple no dirigido.

   La figura 1 muestra una visualización de este generador.

```
G = nx.complete_graph(30)
nx.draw_networkx(G, node_color='yellow', edge_color='g',
    with_labels=False, alpha=0.8)
```

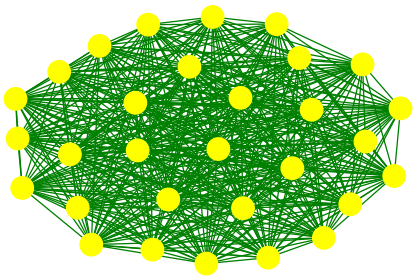


Figura 1: *Complete graph*

2. `Wheel_graph`. El grafo de la rueda tiene un nodo central que está conectado a un ciclo de $(n-1)$ nodos. Recibe como parámetros el número de nodos, el tipo de grafo que se desea crear, el cual tiene como opción predeterminada un grafo simple no dirigido. Las etiquetas de los nodos van de 0 a $n-1$.

   Se presenta la visualización del generador en la figura 2.

```
G = nx.wheel_graph(10)
nx.draw_networkx(G, node_color='b', edge_color='c', with_labels
    =False, node_size=20)
```

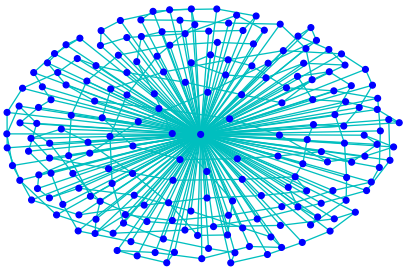


Figura 2: *Wheel graph*

3. `Grid_graph`. Recibe como principal parámetro la cantidad de nodos que puede ser una lista, tupla o un entero y devuelve un grafo de cuadrícula n-dimensional.

   La figura 3 muestra una visualización de este generador.

```
1 G = nx.grid_graph(dim=[8,8])
2 nx.draw_networkx(G, node_color='r', edge_color='k', with_labels
      =False, node_size=80)
```

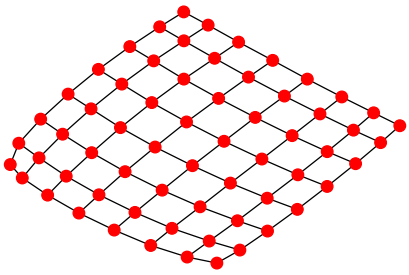


Figura 3: *Grid graph*

4. `Circular_ladder_graph`. Devuelve un grafo de escalera circular, que consiste en dos ciclos de n nodos en donde cada uno de los n pares de nodos se unen por una arista. Toma como dato de entrada la cantidad de nodos y las etiquetas de estos son los números enteros de 0 a $n-1$.

   La figura 4 muestra una visualización de este generador.

```
G = nx.circular_ladder_graph(20)
nx.draw_networkx(G, node_color='b', edge_color='k', with_labels
    =False, node_size=80)
```

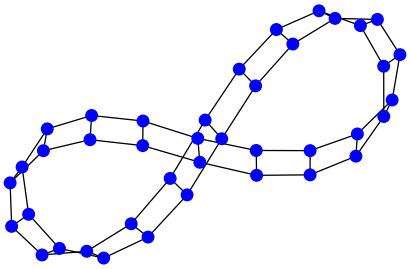


Figura 4: *Circular ladder graph*

Algoritmos de flujo máximo

La elección de los siguientes algoritmos se debe a la variedad que hay para calcular el valor del flujo de un grafo, que reciben parámetros similares y retornan diferentes datos.

1. `Maximum_flow_value`. Encuentra y devuelve el valor del flujo máximo donde cada arista debe tener una capacidad, que en caso de no tenerla toma por defecto una capacidad infinita, recibe un nodo fuente y un nodo sumidero y una función para calcular el flujo máximo. El algoritmo no es compatible con multigrafos.

```
G = nx.DiGraph()
G.add_edge(1, 2, capacity=5)
G.add_edge(2, 3, capacity=10)
G.add_edge(3, 4, capacity=20)
G.add_edge(1, 5, capacity=30)
G.add_edge(5, 6, capacity=15)
G.add_edge(6, 4, capacity=15)
G.add_edge(2, 5, capacity=20)
G.add_edge(5, 3, capacity=10)
G.add_edge(3, 6, capacity=5)

flow_value = nx.maximum_flow_value(G, 1, 6)
```

2. `Minimum_cut`. Recibe como parámetros principales la capacidad y si este dato se omite se considera que tiene una capacidad infinita , nodo fuente y sumidero para el flujo y una función para calcular el flujo máximo. Devuelve el valor del corte mínimo y la partición de los nodos que definen el corte mínimo.

```
G = nx.DiGraph()
G.add_edge(1, 2, capacity=5)
G.add_edge(2, 3, capacity=10)
G.add_edge(3, 4, capacity=20)
G.add_edge(1, 5, capacity=30)
G.add_edge(5, 6, capacity=15)
G.add_edge(6, 4, capacity=15)
G.add_edge(2, 5, capacity=20)
G.add_edge(5, 3, capacity=10)
G.add_edge(3, 6, capacity=5)

cut_value, partition = nx.minimum_cut(G, 1, 6)
```

3. `Minimum_cut_value`. Este algoritmo devuelve solamente el valor que tiene el corte mínimo. Tiene como datos de entrada el nodo fuente y nodo sumidero, una función para calcular el flujo máximo y la capacidad de las aristas que por defecto se considera que tiene una capacidad infinita.

```
G = nx.DiGraph()
G.add_edge(1, 2, capacity=5)
G.add_edge(2, 3, capacity=10)
G.add_edge(3, 4, capacity=20)
G.add_edge(1, 5, capacity=30)
G.add_edge(5, 6, capacity=15)
G.add_edge(6, 4, capacity=15)
G.add_edge(2, 5, capacity=20)
G.add_edge(5, 3, capacity=10)
G.add_edge(3, 6, capacity=5)

cut_value = nx.minimum_cut_value(G, 1, 6)
```

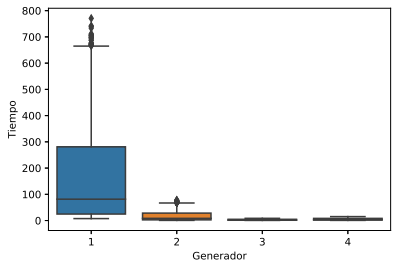
Resultados

Los resultados del análisis de varianza de los efectos simples se muestran en el cuadro 1, donde los factores que presentan una relación significativa con el tiempo de ejecución son la densidad, generador de grafo y orden, es decir, que tienen un p-valor menor a 0.05 (nivel de significancia) y para los que la hipótesis nula de que las medias son iguales se rechaza. Mientras que el factor algoritmo no es significativo con el tiempo de ejecución.

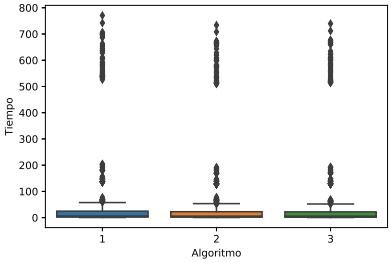
Cuadro 1: *Efectos simples*

Factor	P**-valor**
Algoritmo	0.316
Densidad	0.000
Generador	0.000
Orden	0.000

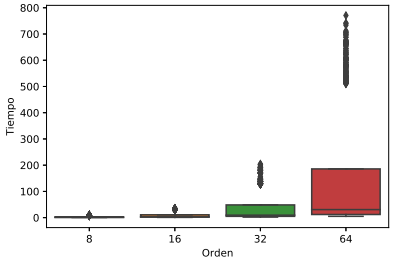
Por otro lado, en la figura 5 se muestran los diagramas de caja y bigotes para cada nivel de los factores donde se observa que el generador 1 es el que mayor tiempo consume a comparación del generador 2, los tres tipos de algoritmos parecen tener tiempos de ejecución similares, conforme va creciendo el orden de los grafos el tiempo de ejecución crece también y que a menor densidad el tiempo aumenta.



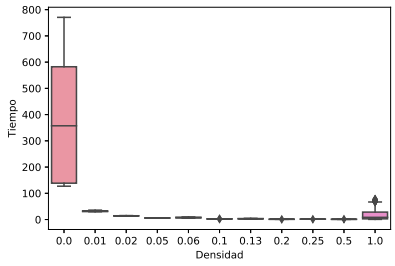
(a) Generador de grafo contra tiempo



(b) Algoritmo contra tiempo



(c) Orden del grafo contra tiempo



(d) Densidad del grafo contra tiempo

Figura 5: *Diagramas de caja y bigotes de los factores.*

Además, en el cuadro 2 se muestran los efectos de interacción para cada pareja de factor de los factores mencionados en el cuadro 1, donde se puede concluir que las interacciones generador-densidad, generador-orden y orden-densidad tienen relación significativa, es decir, que tienen un p-valor menor a 0.05 (nivel de significancia), por lo que la combinación que se tome de cada uno de los factores importa para el valor del tiempo de ejecución. Mientras que las interacciones

algoritmo-densidad, algoritmo-orden y generador-orden no tienen relación significativa con el tiempo dado que el p-valor es mayor a 0.05, por lo que no importa que combinación se tenga de los factores.

Cuadro 2: *Efectos de interacción*

Factor	**P-valor**
Algoritmo-Densidad	0.785
Algoritmo-Orden	0.417
Generador-Algoritmo	0.311
Generador-Densidad	0.000
Generador-Orden	0.000
Orden-Densidad	0.000

Referencias

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

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

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

[4] NumPy developers Versión 1.16.2. `https://networkx.github.io`.

[5] Augspurger T. and the pandas core team Versión 0.24.2. `https://pandas.pydata.org/`.

[6] NetworkX developers con última actualización el 19 de Septiembre 2018. `https://networkx.github.io/documentation/networkx-1.10/reference/drawing.html`.

Tarea 5

Para esta práctica se realizaron correcciones ortográficas y detalles en las gráficas que se utilizaron.

Caracterización estructural de instancias

Brenda Yaneth Sotelo Benítez
5705

30 de abril de 2019

Introducción

En este trabajo se realiza la selección de un algoritmo generador, un algoritmo de flujo máximo y de acomodo que sean adecuados para alguna aplicación específica para generar cinco instancias y establecer si características en los vértices como la distribución de grado, coeficiente de agrupamiento, centralidad de cercanía y de carga, excentricidad y PageRank afectan en el tiempo de ejecución o el valor óptimo en el algoritmo seleccionado, así como analizar que vértices resultan ser buenas fuentes o sumideros y cuáles sería mejor no usar como ninguno si se busca tener un flujo alto, pero no batallar con el tiempo de ejecución.

Los instancias se obtuvieron por medio de la librería Networkx [1] y Matplotlib [2] de `Python` [3] para la generación de grafos y guardar imágenes en el formato *eps* respectivamente. Además se hace uso de la librería Numpy [4] para la suma de tiempos, cálculo de tiempos promedios y desviaciones estándar y Pandas [5] para el tratamiento de datos. El código empleado se obtuvo consultando documentación oficial [6].

Se presenta una breve descripción de los algoritmos seleccionados y las características estructurales de los vértices, visualización de los grafos, fragmentos relevantes de código y una sección de resultados donde se realiza una comparación de distintas variables numéricas a través de grupos, por ejemplo el tiempo de ejecución y las características para vértices.

Las pruebas se han realizado en una PC con procesador Intel Core i3 2.00 GHz y 8.00 GB de RAM.

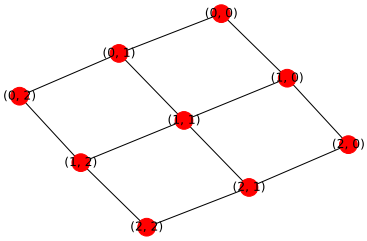
1. Instancias

Una red de distribución de agua tiene gran relevancia tanto en la industria como en la sociedad y puede ser representada mediante un grafo formado por aristas que representan tuberías y vértices, puntos de extracción y fuentes de abastecimiento como son los embalses y depósitos [7].

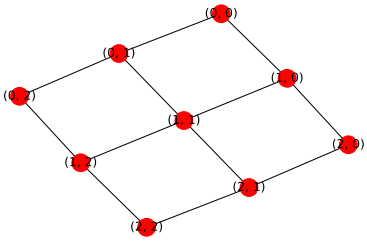
Por tal razón se ha seleccionado el generador `grid_graph` para hacer la representación de cinco instancias donde los vértices son puentes de extracción y fuentes de abastecimiento y las aristas tuberías, a las cuales se les han asignado pesos positivos normalmente distribuidos con media $\mu = 10$ y desviación estándar $\sigma = 3$. También se ha seleccionado el algoritmo `maximum_flow_value` y cada una de las instancias fueron visualizadas con el algoritmo de acomodo `kamada_kawai_layout` que produce un resultado entendible acorde a la aplicación. Las características de los algoritmos mencionados se pueden encontrar en la práctica dos y cuatro del repositorio de Sotelo [8].

En la figura 1 se muestran las instancias de orden 9 y 16 que se generaron con el algoritmo generador y de acomodo.

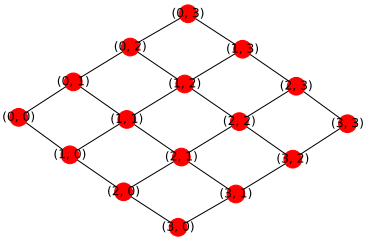
```
1     def __init__(self,x1,x2,x3):
2         self.dim_x=x1
3         self.dim_y=x2
4         self.num=x3
5
6     def crear(self):
7         self.G= nx.grid_graph(dim=[self.dim_x,self.dim_y])
8         self.pos=nx.kamada_kawai_layout(self.G)
9
10    def graficar(self):
11        nx.draw_networkx(self.G,self.pos)
```



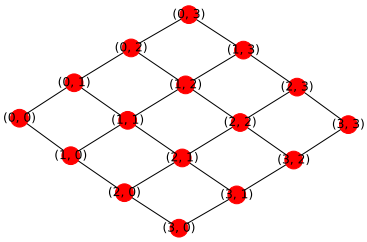
(a) Instancia 1



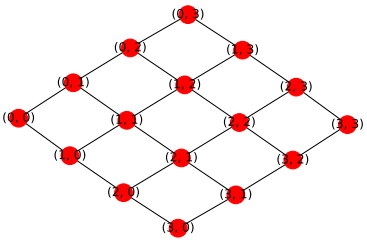
(b) Instancia 2



(c) Instancia 3



(d) Instancia 4



(e) Instancia 5

Figura 1: *Instancias generadas por el algoritmo grid_graph*

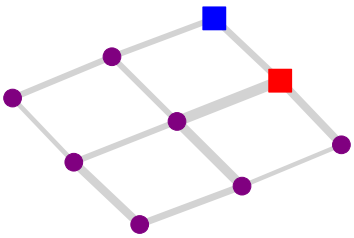
Posteriormente en la figura 2 se les asigna capacidad a las aristas, cuyo valor se representa en el grosor de las aristas. El vértice cuadrado azul representa el vértice fuente, el vértice cuadrado rojo representa el sumidero y el resto de los vértices tienen color morado.

En el cuadro 1 se indica el vértice fuente y sumidero de cada una de las instancias ya que en las siguientes visualizaciones no se muestra el número de nodo.

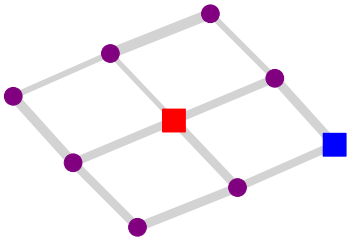
Cuadro 1: *Vértices fuente y sumidero*

Instancia	Fuente	Sumidero
1	(0,0)	(1,0)
2	(2,0)	(1,1)
3	(3,2)	(1,1)
4	(2,1)	(3,0)
5	(1,3)	(2,0)

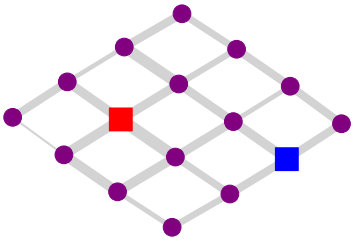
```
1     def Asignarpesos(self, media, desviacion):
2         self.media=media
3         self.desviacion=desviacion
4         K=[i for i in self.G.edges()]
5         Pesos=[round (max(0.1,random.gauss(self.media,self.
      desviacion)),2) for i in range(len(K))]
6         e=[]
7         for i in range(len(self.G.edges())):
8             a=(K[i][0],K[i][1],Pesos[i])
9             e.append(a)
10        self.G.add_weighted_edges_from(e)
11
12    def solucion(self):
13        #Seleccionar nodo fuente y nodo destino
14        buscar=True
15        while buscar:
16
17            origen1=random.randint(0, self.dim_x-1)
18            origen2=random.randint(0, self.dim_y-1)
19            destino1=random.randint(0, self.dim_x-1)
20            destino2=random.randint(0, self.dim_y-1)
```



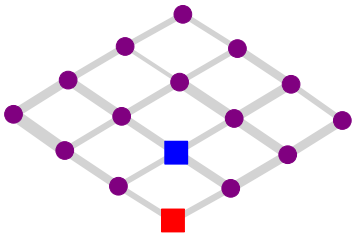
(a) Instancia 1



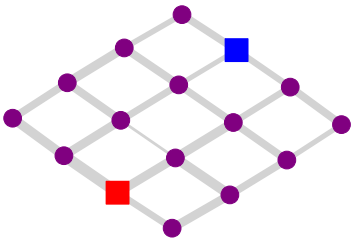
(b) Instancia 2



(c) Instancia 3



(d) Instancia 4

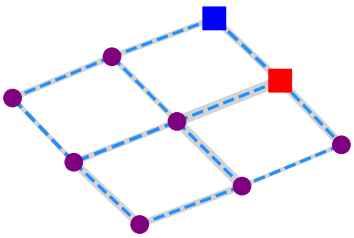


(e) Instancia 5

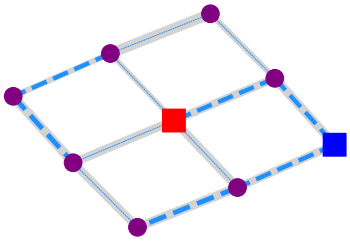
Figura 2: *Instancias con capacidad*

Por último, se representa en la figura 3 el flujo correspondiente a cada arista de color celeste, manteniendo la idea de que el grosor representa la cantidad de flujo que pasa por cada una de ellas.

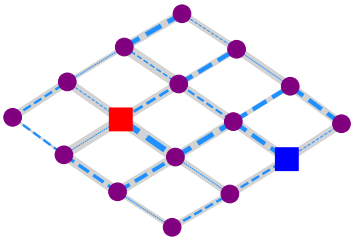
```
        flow_value, flow_dict = nx.maximum_flow(self.G, self.inicio, self.final, capacity='weight')
        self.objetivo=flow_value
        print(self.objetivo)
        #Aqui indica el flujo que pasa por cada arista
        flujo=[]
        pesos=nx.get_edge_attributes(self.G,'weight')
        pesos1=[pesos[i] for i in pesos]

        for i in flow_dict.keys():
            for j in flow_dict[i].keys():
                if flow_dict[i][j]>0:
                    flujo.append(flow_dict[i][j])

        self.grosor_capacidad=[i*0.75 for i in pesos1]
        self.grosor_flujo=[i*0.4 for i in flujo]

        nx.draw_networkx_nodes(self.G,self.pos,node_color='purple')
        nx.draw_networkx_nodes(self.G,self.pos,nodelist=[(origen1, origen2)],node_color='b', node_shape='s',node_size=500)
        nx.draw_networkx_nodes(self.G,self.pos,nodelist=[(destino1, destino2)],node_color='r', node_shape='s',node_size=500)
        nx.draw_networkx_edges(self.G,self.pos,edge_color='lightgray',width=self.grosor_capacidad)
        plt.axis('off')
        plot_margin = 0.05
        x0, x1, y0, y1 = plt.axis()
        plt.axis((x0 - plot_margin,x1 + plot_margin,y0 - plot_margin,y1 + plot_margin))
        plt.tight_layout()
        imagen2="I2_"+str(self.num)+".eps"
        plt.savefig(imagen2)
        plt.show()

        nx.draw_networkx_nodes(self.G,self.pos,node_color='purple')
        nx.draw_networkx_nodes(self.G,self.pos,nodelist=[(origen1, origen2)],node_color='b', node_shape='s',node_size=500)
        nx.draw_networkx_nodes(self.G,self.pos,nodelist=[(destino1, destino2)],node_color='r', node_shape='s',node_size=500)
        nx.draw_networkx_edges(self.G,self.pos,edge_color='lightgray',width=self.grosor_capacidad)
        nx.draw_networkx_edges(self.G,self.pos,edge_color='dodgerblue',width=self.grosor_flujo, style='--')
```



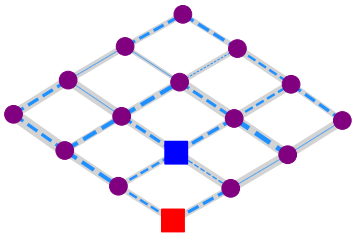
(a) Instancia 1



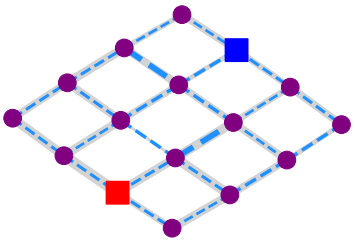
(b) Instancia 2



(c) Instancia 3



(d) Instancia 4



(e) Instancia 5

Figura 3: *Instancias con solución*

2. Características estructurales de los vértices

Distribución de grado

Dado un grafo de tamaño n, el grado de un vértice i se define como el número total de aristas que inciden en él, mientras que la distribución de grado se define como la probabilidad de que un vértice i tenga exactamente k conexiones con otros vértices.

`Degree_centrality`. Calcula la centralidad de grado para los vértices y recibe como único parámetro un grafo.

```
1        A1 = nx.degree_centrality(self.G)
```

Coeficiente de agrupamiento

El coeficiente de agrupamiento mide el nivel de agrupamiento que existe entorno a un vértice, que se calcula como el número total de aristas que conectan a los vecinos más cercanos entre el número máximo de aristas posibles entre todos los vecinos más cercanos. Se puede ver como la probabilidad de que dos vértices vecinos a un vértice i sean vecinos entre sí.

`Clustering`. Este algoritmo calcula el coeficiente de agrupamiento de los vértices y recibe como parámetros un grafo, lista de nodos para los que se desea calcular el coeficiente y peso de las aristas. Estos últimos dos parámetros son opcionales por lo que de no ser incluidos se toman por defecto todos los vértices del grafo y cada arista recibe un peso de uno.

```
1        A3 = nx.clustering(self.G)
```

Centralidad de cercanía

La centralidad de cercanía indica que tan cerca está un vértice i del resto de los vértices del grafo. Se calcula como el promedio de las distancias más cortas desde el vértice i a los demás vértices. En algunos casos, el recíproco de la distancia promedio se utiliza como la medida de centralidad de cercanía para evitar problemas con las distancias en los grafos no conexos y en estos casos los valores más altos indican una centralidad más alta.

`Closeness_centrality`. Este algoritmo calcula la centralidad de cercanía de los vértices, tomando el recíproco del promedio de las distancias más cortas y

normalizando. Recibe como parámetros un grafo, el vértice para el que se quiere encontrar el valor si se desea para uno en específico, la distancia y la opción de normalizar los coeficientes.

```
1        A2 = nx.closeness_centrality(self.G)
```

Centralidad de carga

La centralidad de carga de un vértice i se define como la fracción de todos los caminos más cortos que pasan a través de ese vértice.

`Load_centrality`. Recibe como parámetro principal un grafo y parámetros opcionales la normalización de los valores, el peso y el corte que si se especifica solo considera rutas de longitud menor o igual al corte.

```
1        A4 = nx.load_centrality(self.G)
```

Excentricidad

Se define como la distancia máxima desde un vértice i a a todos los demás vértices del grafo.

`Eccentricity`. Retorna la excentricidad de los nodos de un grafo y recibe como parámetros un grafo y un vértice en particular para el que se desea calcular la excentricidad.

```
1        A5 = nx.eccentricity(self.G)
```

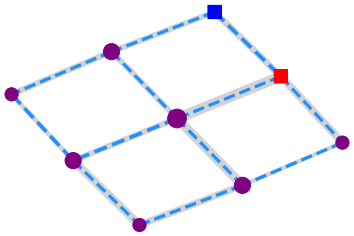
PageRank

Explora e indica la categorización de vértices en redes con base en su importancia.

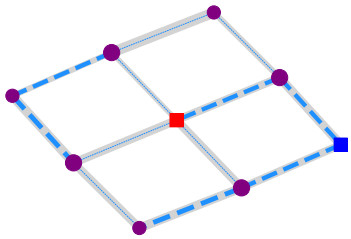
`Pagerank`. Retorna el PageRank de los vértices, donde se calcula una clasificación de los vértices de un grafo en función de la estructura de las conexiones entrantes. Recibe como parámetros un grafo, parametro de atenuación, un número máximo de iteraciones para resolver valores propios, toleancia de error, valor de inicio de la iteración para cada vértice y el peso de cada arista.

```
1        A6 = nx.pagerank(self.G)
```

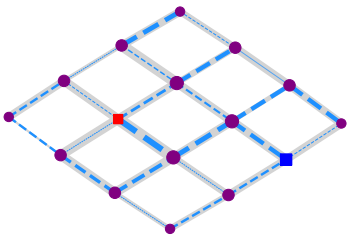
En la figura 4 se incluyen las visualizaciones de las instancias añadiendo la variación de tamaño de los vértices con las características antes mencionadas, para determinar cuales resultan ser relevantes. Se puede observar que para instancia los vértices de las esquinas tienen menor tamaño a comparación de los demás, debido posiblemente a la estructura que se tiene con el generador, las capacidadades y el flujo.



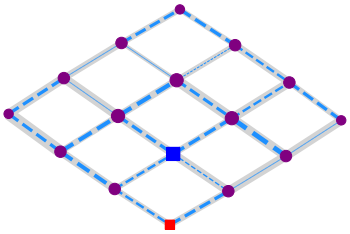
(a) Instancia 1



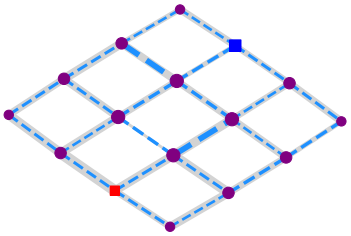
(b) Instancia 2



(c) Instancia 3



(d) Instancia 4



(e) Instancia 5

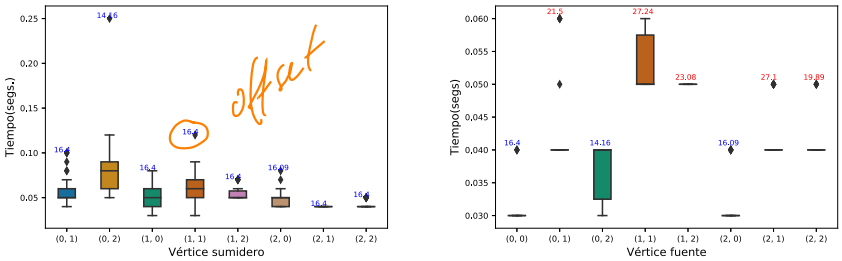
Figura 4: *Instancias con caracterización*

3. Resultados

A continuación se presenta la comparación del tiempo de ejecución variando el vértice fuente y sumidero mediante diagramas de caja y bigote, identificando con rojo los casos en los que el valor de la función objetivo mejora y con azul los que no tienen mejora o empeoran.

El valor de la función objetivo de la instancia 1 es 16.4 y en la figura 5 se observa que al variar el vértice sumidero no se encuentran mejoras, por lo que si busca es tener un tiempo menor, se pueden elegir los valores con el mismo valor objetivo que tardan menos, como lo son el (2,1) y el (2,2).

Por otra parte, al variar el vértice fuente, los vértices (0,1), (1,1), (1,2), (2,1), (2,2) presentan una mejora, siendo el (1,1) el que tiene mayor valor de la función objetivo pero que también consume mucho tiempo a comparación de los otros, mientras el vértice (2,1) presenta un valor objetivo similar con un tiempo de ejecución menor.

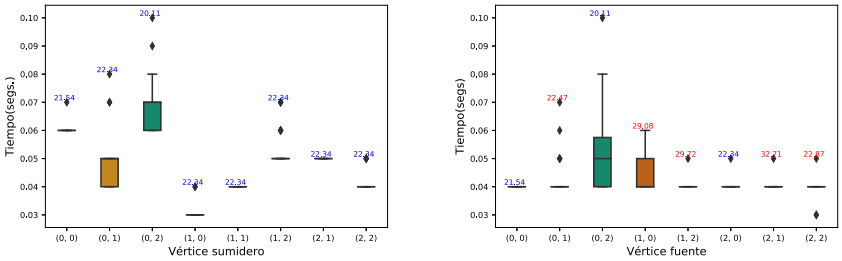


(a) Tiempo de ejecución variando sumidero (b) Tiempo de ejecución variando fuente

Figura 5: *Instancia 1*

El valor objetivo de la instancia 2 es 22.34 y en la figura 6 se observa que al variar el vértice sumidero no se encuentran mejoras, pero si lo que se busca es conservar el valor de la función objetivo con menor tiempo, se consideraría el vértice (1,0).

Por otra parte, al variar el vértice fuente, los vértices que presentan mejora son el (0,1), (1,0), (1,2), (2,1) y (2,2), de los cuales el de mayor valor objetivo y menor tiempo de ejecución es el nodo (2,1).

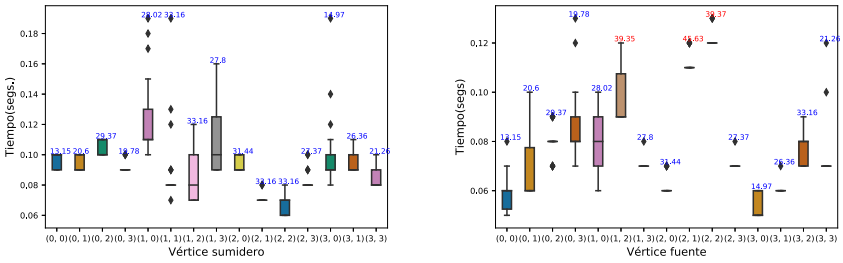


(a) Tiempo de ejecución variando sumidero (b) Tiempo de ejecución variando fuente

Figura 6: *Instancia 2*

El valor objetivo de la instancia 3 es 33.16 y en la figura 7 se observa que al variar el vértice sumidero no se encuentran mejoras, pero si lo que se busca es conservar el valor de la función objetivo con menor tiempo se considera elegir el vértice (2,2).

Por otra parte, los vértices que presentan mejora son el (1,2), (2,1) y (2,2), de los cuales el vértice con mayor valor objetivo es el (2,1) pero con la desventaja de que consume mucho tiempo, mientras que el de menor tiempo es el (1,2).



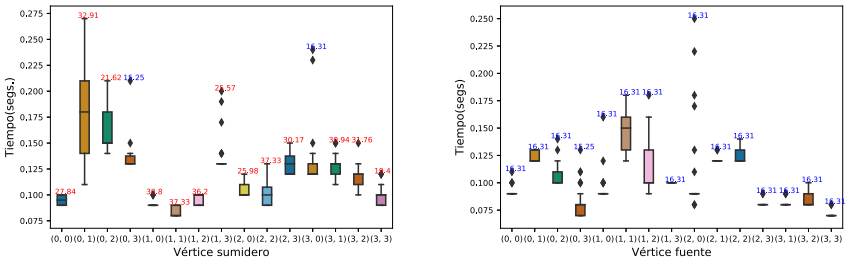
(a) Tiempo de ejecución variando sumidero (b) Tiempo de ejecución variando fuente

Figura 7: *Instancia 3*

El valor objetivo de la instancia 4 es 16.31 y en la figura 8 se muestra que hay mejoras en el valor objetivo de la instancia variando el vértice sumidero original, siendo el (1,1) el de mayor valor objetivo y menor tiempo de ejecución.

Por otra parte, se observa que al variar el vértice fuente no se encuentran mejo-

ras, pero si lo que se busca es conservar el valor objetivo con menor tiempo se considera elegir el vértice (3,3).

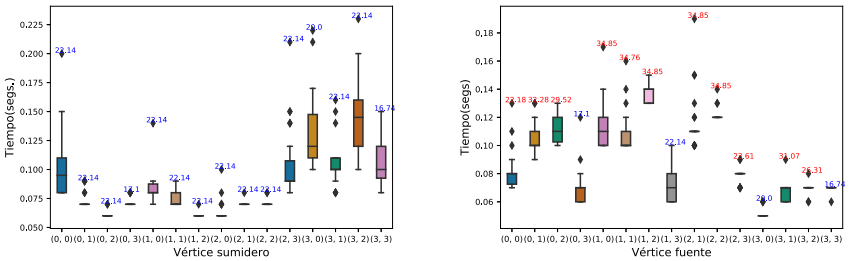


(a) Tiempo de ejecución variando sumidero (b) Tiempo de ejecución variando fuente

Figura 8: *Instancia 4*

El valor objetivo de la instancia 5 es 22.14 y en la figura 9 se observa que al variar el vértice sumidero no se encuentran mejoras, pero si lo que se busca es conservar el valor objetivo con menor tiempo se considera elegir el vértice (0,2) y (1,2).

Por otra parte los vértices que no presentan mejora son el (0,3), (1,3), (3,0) y (3,3), siendo el (1,0) y (1,1) los que presentan mayor valor objetivo y menor tiempo de ejecución .

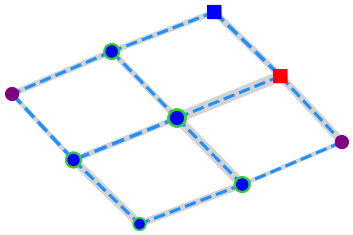


(a) Tiempo de ejecución variando sumidero (b) Tiempo de ejecución variando fuente

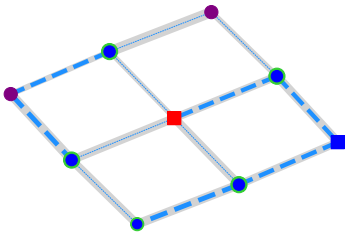
Figura 9: *Instancia 5*

En la figura 10 se visualiza cada una las caracterizaciones en la instancia y donde si se tiene un buen vértice fuente pero no sumidero se representa con

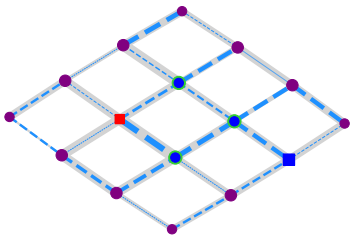
azul y verde limón, aquellos que son mejor vértice sumidero pero no fuente se representa con color rojo y amarillo mientras que los que son mejor fuente y sumidero se representan con el color verde oscuro. Para las instancias 1, 2 y 5 se observa que tienen vértices que son buena opción para considerar como vértices fuente si se quiere mejorar el valor de la función objetivo, mientras que la instancia 2 se tiene que vértices son buena opción como vértices sumidero.



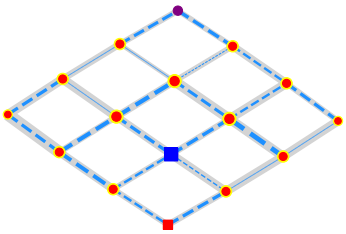
(a) Instancia 1



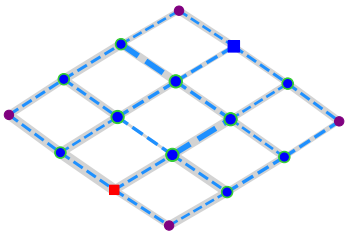
(b) Instancia 2



(c) Instancia 3



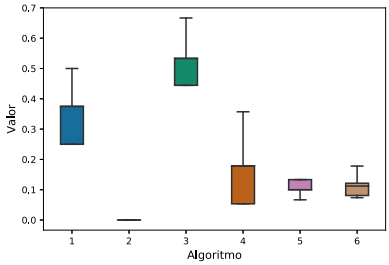
(d) Instancia 4



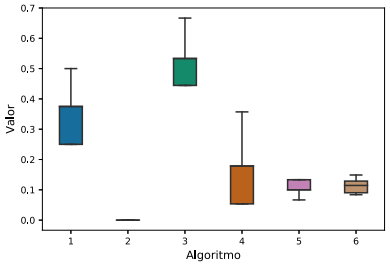
(e) Instancia 5

Figura 10: *Instancias con caracterización*

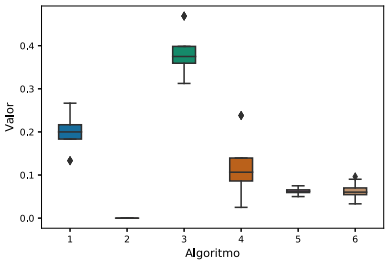
Por último, en la figura 11 se comparan los algoritmos `degree_centrality`, `clustering`, `closeness_centrality`, `load_centrality`, `eccentricity` y `pagerank` para cada instancia, esperando determinar que características pudieran ser relevantes para la solución.



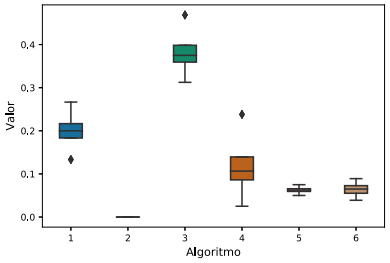
(a) Instancia 1



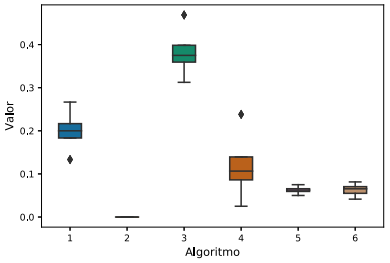
(b) Instancia 2



(c) Instancia 3



(d) Instancia 4



(e) Instancia 5

Figura 11: *Valores obtenidos con los algoritmos*

De acuerdo a los resultados obtenidos de los diagramas de caja y bigote se

obtienen las siguientes conclusiones generales:

1. Se observa que el `closeness_centrality` es el que tiene valores más altos, seguido por `degree_centrality`, `load_centrality`, `pagerank`, `eccentricity` y `clustering` teniendo valor cero en todas las instancias.
2. El algoritmo generador se comporta muy similar sin importar la cantidad de vértices y siempre se obtienen valores similares de las características, por lo que podría decirse que el valor de la función objetivo no depende de las características sino de una buena elección del vértice fuente y sumidero.
3. Tener un vértice fuente en las orillas y un vértice sumidero en el centro asi como ambos en el centro, hace que se obtenga un mejor valor objetivo, ya que los vértices del centro parecen tener mejores valores de las características que los vértices de las orillas y esquinas tal y como se muestra en la figura 10.

Referencias

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

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

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

[4] NumPy developers Versión 1.16.2. `https://networkx.github.io`.

[5] Augspurger T. and the pandas core team Versión 0.24.2. `https://pandas.pydata.org/`.

[6] NetworkX developers con última actualización el 19 de Septiembre 2018. `https://networkx.github.io/documentation/networkx-1.10/reference/drawing.html`.

[7] Oscar Tomas Vegas Niño and Velitchko Martínez Alzamora, Fernando y Tzatchkov. Aplicación de la teoría de grafos a la identificación de subsistemas hidráulicos en redes de distribución de agua. 09 2016.

[8] Sotelo B. Repositorio optimización flujo en redes. `https://github.com/BrendaSotelo/Flujo_Redes_BSotelo`.

Caracterización estructural de instancias

Brenda Yaneth Sotelo Benítez
5705

3 de junio de 2019

Introducción

En este trabajo se realiza la selección de un algoritmo generador, un algoritmo de flujo máximo y de acomodo que sean adecuados para alguna aplicación específica para generar cinco instancias y establecer si características en los vértices como la distribución de grado, coeficiente de agrupamiento, centralidad de cercanía y de carga, excentricidad y PageRank afectan en el tiempo de ejecución o el valor óptimo en el algoritmo seleccionado, así como analizar que vértices resultan ser buenas fuentes o sumideros y cuáles sería mejor no usar como ninguno si se busca tener un flujo alto, pero no batallar con el tiempo de ejecución.

Los instancias se obtuvieron por medio de la librería Networkx [1] y Matplotlib [2] de `Python` [3] para la generación de grafos y guardar imágenes en el formato *eps* respectivamente. Además se hace uso de la librería Numpy [4] para la suma de tiempos, cálculo de tiempos promedios y desviaciones estándar y Pandas [5] para el tratamiento de datos. El código empleado se obtuvo consultando documentación oficial [6].

Se presenta una breve descripción de los algoritmos seleccionados y las características estructurales de los vértices, visualización de los grafos, fragmentos relevantes de código y una sección de resultados donde se realiza una comparación de distintas variables numéricas a través de grupos, por ejemplo el tiempo de ejecución y las características para vértices.

Las pruebas se han realizado en una PC con procesador Intel Core i3 2.00 GHz y 8.00 GB de RAM.

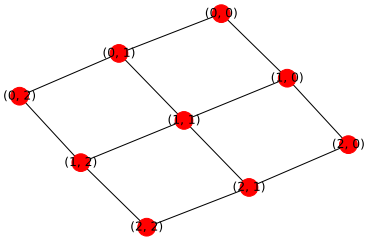
1. Instancias

Una red de distribución de agua tiene gran relevancia tanto en la industria como en la sociedad y puede ser representada mediante un grafo formado por aristas que representan tuberías y vértices, puntos de extracción y fuentes de abastecimiento como son los embalses y depósitos [7].

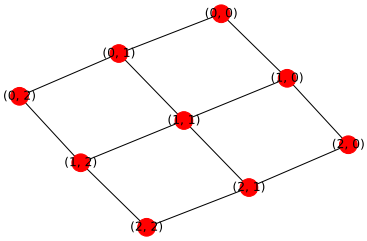
Por tal razón se ha seleccionado el generador `grid_graph` para hacer la representación de cinco instancias donde los vértices son puentes de extracción y fuentes de abastecimiento y las aristas tuberías, a las cuales se les han asignado pesos positivos normalmente distribuidos con media $\mu = 10$ y desviación estándar $\sigma = 3$. También se ha seleccionado el algoritmo `maximum_flow_value` y cada una de las instancias fueron visualizadas con el algoritmo de acomodo `kamada_kawai_layout` que produce un resultado entendible acorde a la aplicación. Las características de los algoritmos mencionados se pueden encontrar en la práctica dos y cuatro del repositorio de Sotelo [8].

En la figura 1 se muestran las instancias de orden 9 y 16 que se generaron con el algoritmo generador y de acomodo.

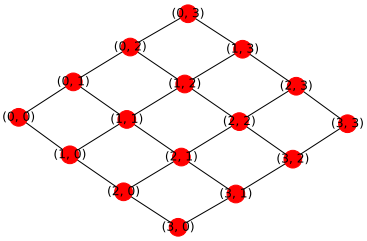
```
1     def __init__(self,x1,x2,x3):
2         self.dim_x=x1
3         self.dim_y=x2
4         self.num=x3
5
6     def crear(self):
7         self.G= nx.grid_graph(dim=[self.dim_x,self.dim_y])
8         self.pos=nx.kamada_kawai_layout(self.G)
9
10    def graficar(self):
11        nx.draw_networkx(self.G,self.pos)
```



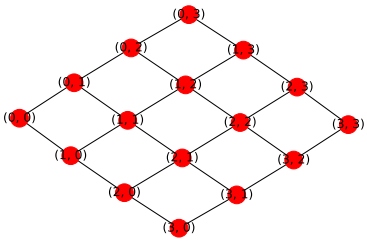
(a) Instancia 1



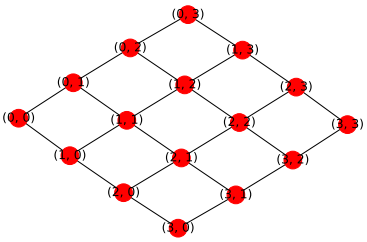
(b) Instancia 2



(c) Instancia 3



(d) Instancia 4



(e) Instancia 5

Figura 1: *Instancias generadas por el algoritmo grid_graph*

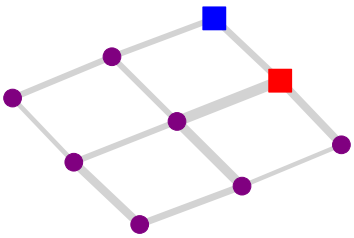
Posteriormente en la figura 2 se les asigna capacidad a las aristas, cuyo valor se representa en el grosor de las aristas. El vértice cuadrado azul representa el vértice fuente, el vértice cuadrado rojo representa el sumidero y el resto de los vértices tienen color morado.

En el cuadro 1 se indica el vértice fuente y sumidero de cada una de las instancias ya que en las siguientes visualizaciones no se muestra el número de nodo.

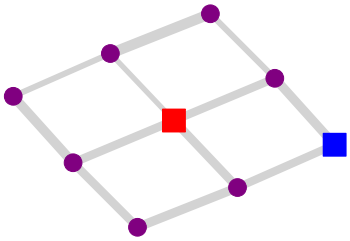
Cuadro 1: *Vértices fuente y sumidero*

Instancia	Fuente	Sumidero
1	(0,0)	(1,0)
2	(2,0)	(1,1)
3	(3,2)	(1,1)
4	(2,1)	(3,0)
5	(1,3)	(2,0)

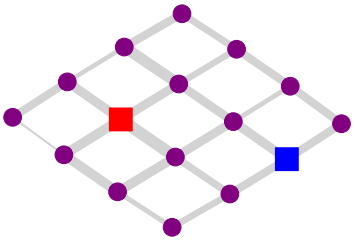
```
1     def Asignarpesos(self, media, desviacion):
2         self.media=media
3         self.desviacion=desviacion
4         K=[i for i in self.G.edges()]
5         Pesos=[round (max(0.1,random.gauss(self.media,self.
      desviacion)),2) for i in range(len(K))]
6         e=[]
7         for i in range(len(self.G.edges())):
8             a=(K[i][0],K[i][1],Pesos[i])
9             e.append(a)
10        self.G.add_weighted_edges_from(e)
11
12    def solucion(self):
13        #Seleccionar nodo fuente y nodo destino
14        buscar=True
15        while buscar:
16
17            origen1=random.randint(0, self.dim_x-1)
18            origen2=random.randint(0, self.dim_y-1)
19            destino1=random.randint(0, self.dim_x-1)
20            destino2=random.randint(0, self.dim_y-1)
```



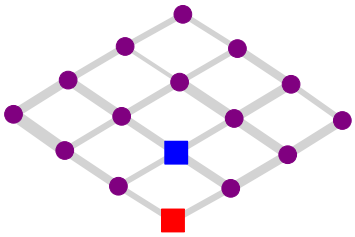
(a) Instancia 1



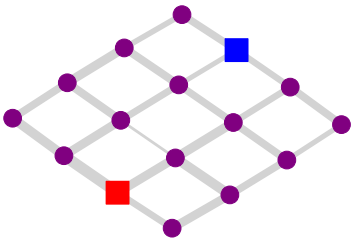
(b) Instancia 2



(c) Instancia 3



(d) Instancia 4

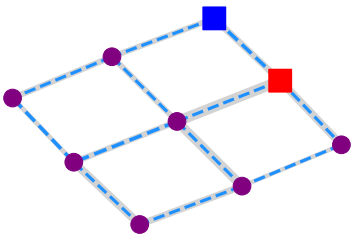


(e) Instancia 5

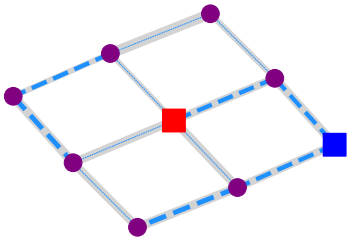
Figura 2: *Instancias con capacidad*

Por último, se representa en la figura 3 el flujo correspondiente a cada arista de color celeste, manteniendo la idea de que el grosor representa la cantidad de flujo que pasa por cada una de ellas.

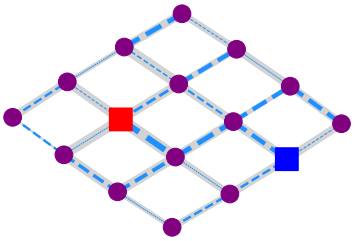
```
        flow_value, flow_dict = nx.maximum_flow(self.G, self.inicio, self.final, capacity='weight')
        self.objetivo=flow_value
        print(self.objetivo)
        #Aqui indica el flujo que pasa por cada arista
        flujo=[]
        pesos=nx.get_edge_attributes(self.G,'weight')
        pesos1=[pesos[i] for i in pesos]

        for i in flow_dict.keys():
            for j in flow_dict[i].keys():
                if flow_dict[i][j]>0:
                    flujo.append(flow_dict[i][j])

        self.grosor_capacidad=[i*0.75 for i in pesos1]
        self.grosor_flujo=[i*0.4 for i in flujo]

        nx.draw_networkx_nodes(self.G,self.pos,node_color='purple')
        nx.draw_networkx_nodes(self.G,self.pos,nodelist=[(origen1,origen2)],node_color='b', node_shape='s',node_size=500)
        nx.draw_networkx_nodes(self.G,self.pos,nodelist=[(destino1,destino2)],node_color='r', node_shape='s',node_size=500)
        nx.draw_networkx_edges(self.G,self.pos,edge_color='lightgray',width=self.grosor_capacidad)
        plt.axis('off')
        plot_margin = 0.05
        x0, x1, y0, y1 = plt.axis()
        plt.axis((x0 - plot_margin,x1 + plot_margin,y0 - plot_margin,y1 + plot_margin))
        plt.tight_layout()
        imagen2="I2_"+str(self.num)+".eps"
        plt.savefig(imagen2)
        plt.show()

        nx.draw_networkx_nodes(self.G,self.pos,node_color='purple')
        nx.draw_networkx_nodes(self.G,self.pos,nodelist=[(origen1,origen2)],node_color='b', node_shape='s',node_size=500)
        nx.draw_networkx_nodes(self.G,self.pos,nodelist=[(destino1,destino2)],node_color='r', node_shape='s',node_size=500)
        nx.draw_networkx_edges(self.G,self.pos,edge_color='lightgray',width=self.grosor_capacidad)
        nx.draw_networkx_edges(self.G,self.pos,edge_color='dodgerblue',width=self.grosor_flujo, style='--')
```



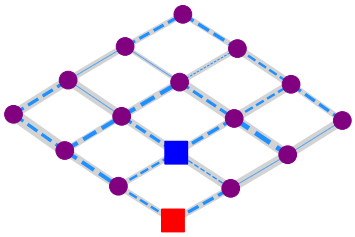
(a) Instancia 1



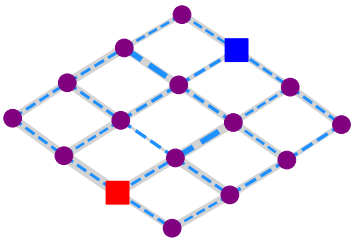
(b) Instancia 2



(c) Instancia 3



(d) Instancia 4



(e) Instancia 5

Figura 3: *Instancias con solución*

2. Características estructurales de los vértices

En esta sección se da una explicación de las características estructurales y fragmentos relevantes de código.

Distribución de grado

Dado un grafo de tamaño n, el grado de un vértice i se define como el número total de aristas que inciden en él, mientras que la distribución de grado se define como la probabilidad de que un vértice i tenga exactamente k conexiones con otros vértices.

`Degree_centrality`. Calcula la centralidad de grado para los vértices y recibe como único parámetro un grafo.

```
1        A1 = nx.degree_centrality(self.G)
```

Coeficiente de agrupamiento

El coeficiente de agrupamiento mide el nivel de agrupamiento que existe entorno a un vértice, que se calcula como el número total de aristas que conectan a los vecinos más cercanos entre el número máximo de aristas posibles entre todos los vecinos más cercanos. Se puede ver como la probabilidad de que dos vértices vecinos a un vértice i sean vecinos entre sí.

`Clustering`. Este algoritmo calcula el coeficiente de agrupamiento de los vértices y recibe como parámetros un grafo, lista de nodos para los que se desea calcular el coeficiente y peso de las aristas. Estos últimos dos parámetros son opcionales por lo que de no ser incluidos se toman por defecto todos los vértices del grafo y cada arista recibe un peso de uno.

```
1        A3 = nx.clustering(self.G)
```

Centralidad de cercanía

La centralidad de cercanía indica que tan cerca está un vértice i del resto de los vértices del grafo. Se calcula como el promedio de las distancias más cortas desde el vértice i a los demás vértices. En algunos casos, el recíproco de la distancia promedio se utiliza como la medida de centralidad de cercanía para evitar problemas con las distancias en los grafos no conexos y en estos casos los valores más altos indican una centralidad más alta.

`Closeness_centrality`. Este algoritmo calcula la centralidad de cercanía de los vértices, tomando el recíproco del promedio de las distancias más cortas y normalizando. Recibe como parámetros un grafo, el vértice para el que se quiere encontrar el valor si se desea para uno en específico, la distancia y la opción de normalizar los coeficientes.

```
A2 = nx.closeness_centrality(self.G)
```

Centralidad de carga

La centralidad de carga de un vértice i se define como la fracción de todos los caminos más cortos que pasan a través de ese vértice.

`Load_centrality`. Recibe como parámetro principal un grafo y parámetros opcionales la normalización de los valores, el peso y el corte que si se especifica solo considera rutas de longitud menor o igual al corte.

```
A4 = nx.load_centrality(self.G)
```

Excentricidad

Se define como la distancia máxima desde un vértice i a a todos los demás vértices del grafo.

`Eccentricity`. Retorna la excentricidad de los nodos de un grafo y recibe como parámetros un grafo y un vértice en particular para el que se desea calcular la excentricidad.

```
A5 = nx.eccentricity(self.G)
```

PageRank

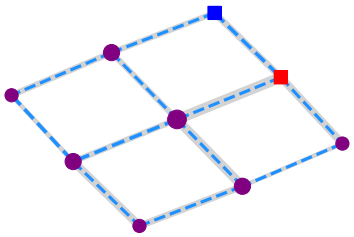
Explora e indica la categorización de vértices en redes con base en su importancia.

`Pagerank`. Retorna el PageRank de los vértices, donde se calcula una clasificación de los vértices de un grafo en función de la estructura de las conexiones entrantes. Recibe como parámetros un grafo, parametro de atenuación, un número máximo de iteraciones para resolver valores propios, toleancia de error, valor de inicio de la iteración para cada vértice y el peso de cada arista.

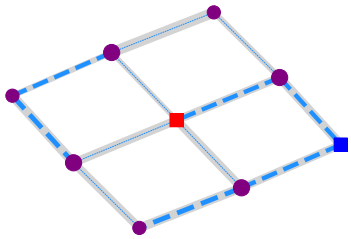
```
A6 = nx.pagerank(self.G)
```

En la figura 4 se incluyen las visualizaciones de las instancias añadiendo la variación de tamaño de los vértices con las características antes mencionadas,

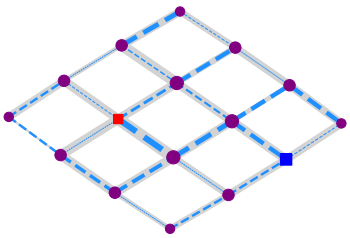
para determinar cuales resultan ser relevantes. Se puede observar que para instancia los vértices de las esquinas tienen menor tamaño a comparación de los demás, debido posiblemente a la estructura que se tiene con el generador, las capacidadades y el flujo.



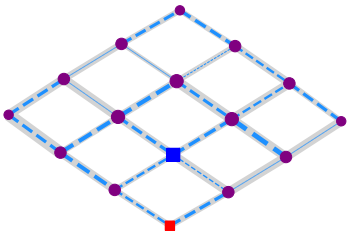
(a) Instancia 1



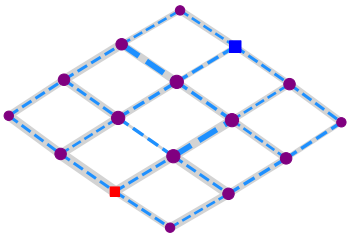
(b) Instancia 2



(c) Instancia 3



(d) Instancia 4



(e) Instancia 5

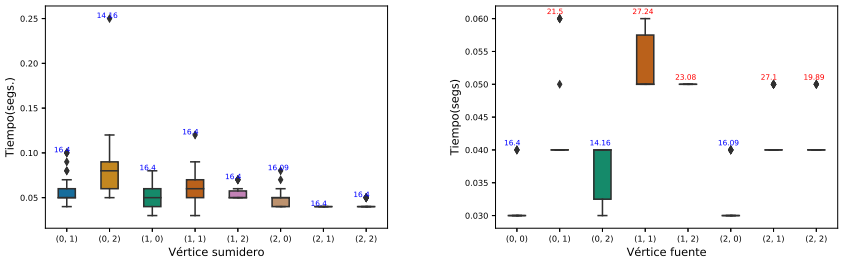
Figura 4: *Instancias con caracterización*

3. Resultados

A continuación se presenta la comparación del tiempo de ejecución variando el vértice fuente y sumidero mediante diagramas de caja y bigote, identificando con rojo los casos en los que el valor de la función objetivo mejora y con azul los que no tienen mejora o empeoran.

El valor de la función objetivo de la instancia 1 es 16.4 y en la figura 5 se observa que al variar el vértice sumidero no se encuentran mejoras, por lo que si busca es tener un tiempo menor, se pueden elegir los valores con el mismo valor objetivo que tardan menos, como lo son el (2,1) y el (2,2).

Por otra parte, al variar el vértice fuente, los vértices (0,1), (1,1), (1,2), (2,1), (2,2) presentan una mejora, siendo el (1,1) el que tiene mayor valor de la función objetivo pero que también consume mucho tiempo a comparación de los otros, mientras el vértice (2,1) presenta un valor objetivo similar con un tiempo de ejecución menor.



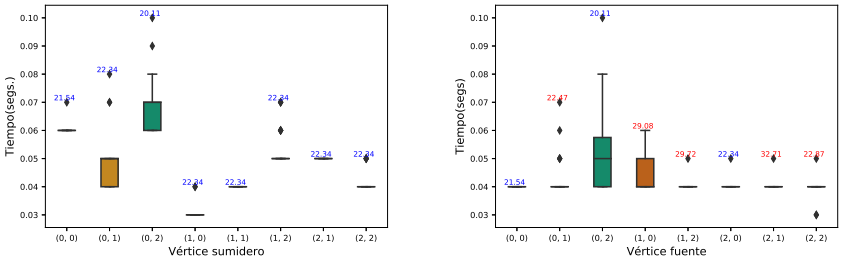
(a) Variando sumidero

(b) Variando fuente

Figura 5: *Tiempo de ejecución de la instancia 1*

El valor objetivo de la instancia 2 es 22.34 y en la figura 6 se observa que al variar el vértice sumidero no se encuentran mejoras, pero si lo que se busca es conservar el valor de la función objetivo con menor tiempo, se consideraría el vértice (1,0).

Por otra parte, al variar el vértice fuente, los vértices que presentan mejora son el (0,1), (1,0), (1,2), (2,1) y (2,2), de los cuales el de mayor valor objetivo y menor tiempo de ejecución es el nodo (2,1).



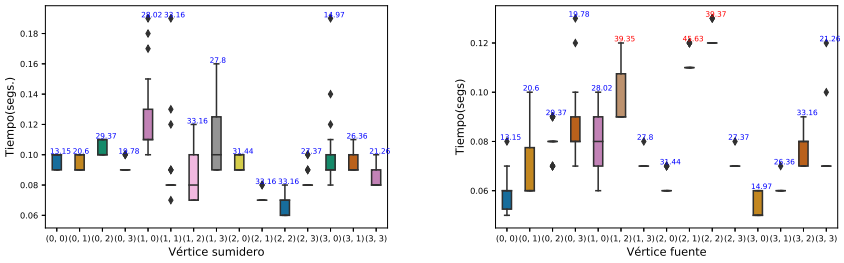
(a) Variando sumidero

(b) Variando fuente

Figura 6: *Tiempo de ejecución de la instancia 2*

El valor objetivo de la instancia 3 es 33.16 y en la figura 7 se observa que al variar el vértice sumidero no se encuentran mejoras, pero si lo que se busca es conservar el valor de la función objetivo con menor tiempo se considera elegir el vértice (2,2).

Por otra parte, los vértices que presentan mejora son el (1,2), (2,1) y (2,2), de los cuales el vértice con mayor valor objetivo es el (2,1) pero con la desventaja de que consume mucho tiempo, mientras que el de menor tiempo es el (1,2).



(a) Variando sumidero

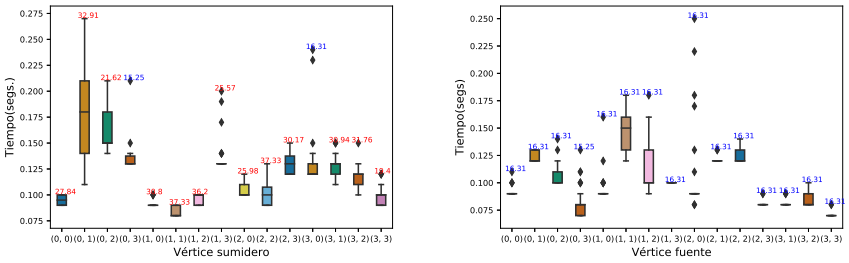
(b) Variando fuente

Figura 7: *Tiempo de ejecución de la instancia 3*

El valor objetivo de la instancia 4 es 16.31 y en la figura 8 se muestra que hay mejoras en el valor objetivo de la instancia variando el vértice sumidero original, siendo el (1,1) el de mayor valor objetivo y menor tiempo de ejecución.

Por otra parte, se observa que al variar el vértice fuente no se encuentran mejo-

ras, pero si lo que se busca es conservar el valor objetivo con menor tiempo se considera elegir el vértice (3,3).

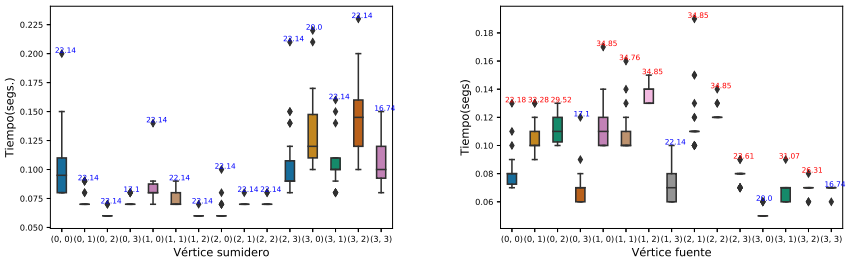


(a) Variando sumidero (b) Variando fuente

Figura 8: *Tiempo de ejecución de la instancia 4*

El valor objetivo de la instancia 5 es 22.14 y en la figura 9 se observa que al variar el vértice sumidero no se encuentran mejoras, pero si lo que se busca es conservar el valor objetivo con menor tiempo se considera elegir el vértice (0,2) y (1,2).

Por otra parte los vértices que no presentan mejora son el (0,3), (1,3), (3,0) y (3,3), siendo el (1,0) y (1,1) los que presentan mayor valor objetivo y menor tiempo de ejecución.

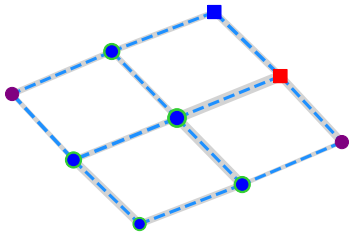


(a) Variando sumidero (b) Variando fuente

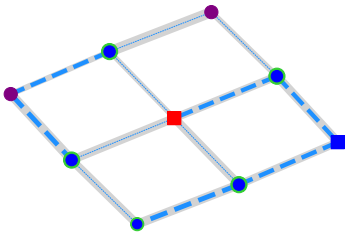
Figura 9: *Tiempo de ejecución de la instancia 5*

En la figura 10 se visualiza cada una las caracterizaciones en la instancia y donde si se tiene un buen vértice fuente pero no sumidero se representa con

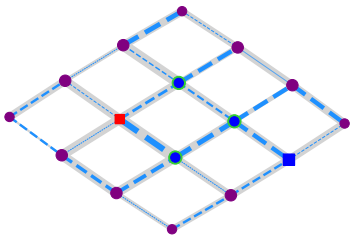
azul y verde limón, aquellos que son mejor vértice sumidero pero no fuente se representa con color rojo y amarillo mientras que los que son mejor fuente y sumidero se representan con el color verde oscuro. Para las instancias 1, 2 y 5 se observa que tienen vértices que son buena opción para considerar como vértices fuente si se quiere mejorar el valor de la función objetivo, mientras que la instancia 2 se tiene que vértices son buena opción como vértices sumidero.



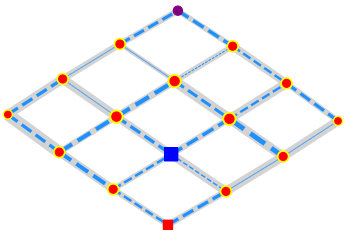
(a) Instancia 1



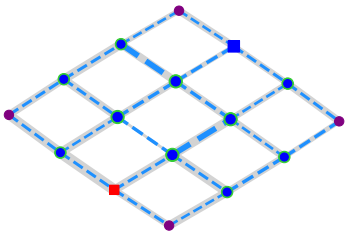
(b) Instancia 2



(c) Instancia 3



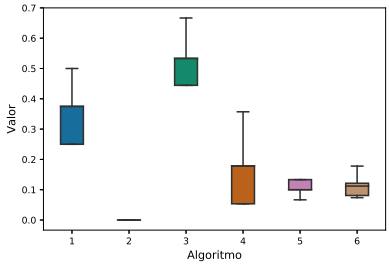
(d) Instancia 4



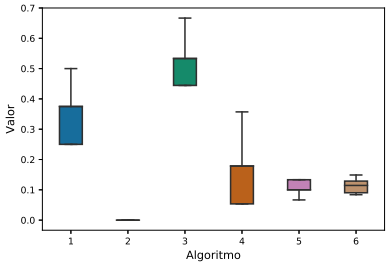
(e) Instancia 5

Figura 10: *Instancias con caracterización*

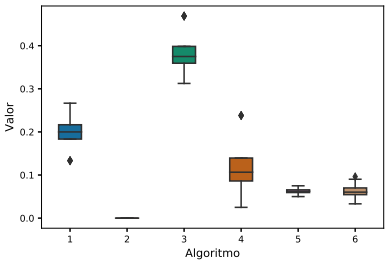
Por último, en la figura 11 se comparan los algoritmos `degree_centrality`, `clustering`, `closeness_centrality`, `load_centrality`, `eccentricity` y `pagerank` para cada instancia, esperando determinar que características pudieran ser relevantes para la solución.



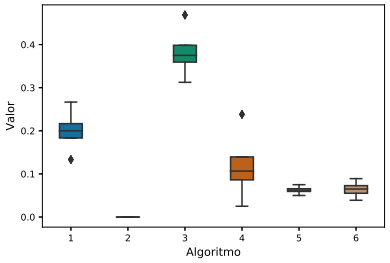
(a) Instancia 1



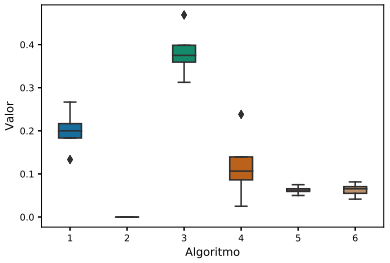
(b) Instancia 2



(c) Instancia 3



(d) Instancia 4



(e) Instancia 5

Figura 11: *Valores obtenidos con los algoritmos*

De acuerdo a los resultados obtenidos de los diagramas de caja y bigote se

obtienen las siguientes conclusiones generales:

1. Se observa que el `closeness_centrality` es el que tiene valores más altos, seguido por `degree_centrality`, `load_centrality`, `pagerank`, `eccentricity` y `clustering` teniendo valor cero en todas las instancias.
2. El algoritmo generador se comporta muy similar sin importar la cantidad de vértices y siempre se obtienen valores similares de las características, por lo que podría decirse que el valor de la función objetivo no depende de las características sino de una buena elección del vértice fuente y sumidero.
3. Tener un vértice fuente en las orillas y un vértice sumidero en el centro asi como ambos en el centro, hace que se obtenga un mejor valor objetivo, ya que los vértices del centro parecen tener mejores valores de las características que los vértices de las orillas y esquinas tal y como se muestra en la figura 10.

Referencias

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

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

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

[4] NumPy developers Versión 1.16.2. `https://networkx.github.io`.

[5] Augspurger T. and the pandas core team Versión 0.24.2. `https://pandas.pydata.org/`.

[6] NetworkX developers con última actualización el 19 de Septiembre 2018. `https://networkx.github.io/documentation/networkx-1.10/reference/drawing.html`.

[7] Vegas Niño. Aplicación de la teoría de grafos a la identificación de subsistemas hidráulicos en redes de distribución de agua. 09 2016.

[8] Sotelo B. Repositorio optimización flujo en redes. `https://github.com/BrendaSotelo/Flujo_Redes_BSotelo`.

Tarea 6

Finalmente se incluye la tarea 6 que no consta de un archivo con correcciones.

Problema de evacuación de una ciudad

Brenda Yaneth Sotelo Benítez
5705

3 de junio de 2019

Introducción

Problemas de optimización pueden ser resueltos utilizando el algoritmo de flujo máximo, tales como el problema de corte mínimo o el problema de acoplamiento máximo. En este trabajo se plantea el problema de evacuación de personas o evacuación de una ciudad para el que se muestra la red representativa del problema y la red modificada de tal forma que se resuelva como un problema de flujo máximo.

Los instancias se obtuvieron por medio de la librería Networkx [1] y Matplotlib [2] de `Python` [3] para la generación de grafos y guardar imágenes en el formato *eps* respectivamente. Además cada una de las instancias fueron visualizadas con el algoritmo de acomodo `bipartite_layout` que produce un resultado entendible acorde a la aplicación. Las caracteristicas de los algoritmos mencionados se pueden encontrar en la práctica dos y cuatro del repositorio de Sotelo [4]. El código empleado se obtuvo consultando documentación oficial [5].

Se presenta una breve descripción del problema seleccionado, visualización de las redes, fragmentos relevantes de código y una sección de resultados.

Las pruebas se han realizado en una PC con procesador Intel Core i3 2.00 GHz y 8.00 GB de RAM.

Problema de evacuación de personas

Un plan de evacuación se define como la planificación y organización para utilizar de forma óptima los medios que se disponen con la finalidad de reducir

las posibles consecuencias que pudieran derivarse de una situación de riesgo como desbordamientos de ríos, deslaves de montañas, sismos, terremotos, maremotos, tornados, huracanes, incendios forestales, tsunamis, ciclones, erupción de un volcán, inundaciones, guerras, epidemias, ataques terroristas, incendios, explosiones, etc.

En la figura 1 se muestra la red representativa del problema donde se tienen diez personas, tres vehículos y tres calles. Se desea saber cuál es la mejor asignación de persona–vehículo y vehículo–calle. Cada persona y cada vehículo puede ir a cualquier vehículo y calle respectivamente, solamente una vez.

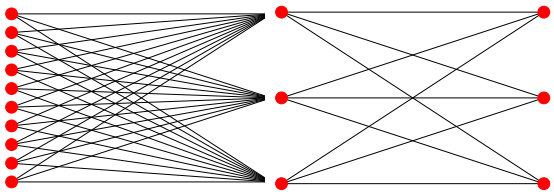


Figura 1: *Red representativa del problema*

Datos de entrada

Los parámetros utilizados para la resolución del problema son los siguientes:

- **Número de personas.** Como su nombre lo indica, este parámetro representa la cantidad de habitantes que hay en la zona afectada que serán evacuados en una cierta cantidad de vehículos que se tiene disponible para cuado ocurre alguna situación de emergencia.
- **Número de vehículos.** Ante una situación de este tipo se cuenta con una cantidad finita de vehículos.
- **Número de calles.** Durante el traslado de las personas existen diferentes rutas que se pueden tomar, de tal forma que se permita tener alternativas ya sea más rápidas o ante situaciones de congestionamiento.
- **Capacidad del vehículo.** Representa la cantidad de personas que pueden trasladarse por vehículo.
- **Capacidad de las calles.** Las calles, carreteras o avenidas pueden ir en un sentido o dos, lo que se busca es que el traslado sea lo más rápido posible por lo que no se quisiera que todos los vehículos siguieran una misma ruta uno tras otro, es por eso que se establece una cantidad de vehículos

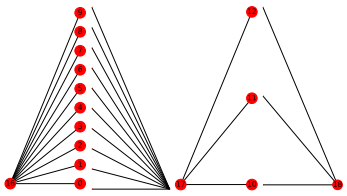
por calle para que se tomen rutas alternas y evitar el agrupamiento de vehículos por calle.

En código 1 se muestra el fragmento de código de los parámetros de la instancia.

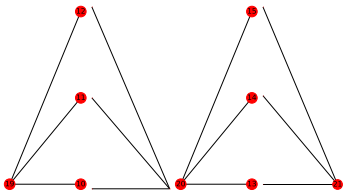
Código 1: *Parámetros de la instancia*

```
#Cardinalidades
numpersonas = 1000
numvehiculos =80
numcalles = 20

#Capacidades
personasxvehiculo=3
vehiculosxcalle=4
```

En la figura 2 se muestra la red utilizada para resolver el algoritmo de flujo máximo, la cual ha sido dividida en dos secciones. La primer sección consta de asignar cada persona a algún vehículo y la segunda consiste en realizar la asignación de cada vehículo a alguna calle, además se consideran las capacidades de personas por vehículo así como vehículo por calle para cumplir con las restricciones deseadas.



(a) Asignación persona vehículo



(b) Asignación vehículo calle

Figura 2: *Red solución*

En el cuadro 1 se tiene una instancia del problema en la que son 1000 personas que necesitan ser evacuadas, aunque es claro que en las ciudades puede haber más habitantes. Se tienen también 250 vehículos y 20 calles, cada una con capacidad 3 y 4 respectivamente.

Cuadro 1: *Datos de entrada de la instancia*

Parámetros	**Unidades**	**Capacidad**
Personas	1000	–
Vehículos	80	3
Calles	20	4

Datos de salida

Como solución se retornan dos valores, el primero de ellos representa la cantidad de personas que son asignadas y evacuadas dependiendo de la instancia y el segundo es la cantidad de vehículos que serán utilizados.

Código 2: *Crear conjuntos de nodos y solución*

```
A1 = [ (u,v,{'cap':1})  if u>v else (v,u,{'cap':1}) for (u,v) in G1
    .edges] #Con menor porque el S es el siguiente
A2 = [ (u,v,{'cap':1})  if u<v else (v,u,{'cap':1}) for (u,v) in G2
    .edges] #Con mayor porque
A3 = [ (u,v,{'cap':personasxvehiculo})  if u<v else (v,u,{'cap':
    personasxvehiculo}) for (u,v) in G3.edges]
A4 = [ (u,v,{'cap':personasxvehiculo})  if u<v else (v,u,{'cap':
    personasxvehiculo}) for (u,v) in G4.edges] #Con mayor porque

A5 = [ (u,v,{'cap':1})  if u>v else (v,u,{'cap':1}) for (u,v) in G5
    .edges] #Con menor porque el S es el siguiente
A6 = [ (u,v,{'cap':1})  if u<v else (v,u,{'cap':1}) for (u,v) in G6
    .edges] #Con mayor porque
A7 = [ (u,v,{'cap':vehiculosxcalle})  if u<v else (v,u,{'cap':
    vehiculosxcalle}) for (u,v) in G7.edges]
A8 = [ (u,v,{'cap':vehiculosxcalle})  if u<v else (v,u,{'cap':
    vehiculosxcalle}) for (u,v) in G8.edges] #Con mayor porque

Gfinal1 = nx.Graph()
Gfinal1.add_edges_from(A1+A2+A3+A4)

Gfinal2 = nx.Graph()
Gfinal2.add_edges_from(A5+A6+A7+A8)

valor1,flujo_arcos1 = nx.maximum_flow(Gfinal1,aux1,aux3,'cap')
valor2,flujo_arcos2 = nx.maximum_flow(Gfinal2,aux4,aux6,'cap')
```

En el cuadro 2 se muestra la solución de la instancia mencionada en el cuadro

1 en la que se tienen 20 calles disponibles y cada una tiene una capacidad de 4 vehículos y donde a lo más transitarían 80 vehículos. Luego si se tienen 80 vehículos y cada uno tiene una capacidad de 3 personas se lograrían transportar como máximo 240 personas.

Cuadro 2: *Datos de salida de la instancia*

Parámetros	**Unidades**	**Capacidad**	**Solución**
Personas	1000	–	240
Vehículos	80	3	80
Calles	20	4	20

Referencias

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

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

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

[4] Sotelo B. Repositorio optimización flujo en redes. `https://github.com/BrendaSotelo/Flujo_Redes_BSotelo`.

[5] NetworkX developers con última actualización el 19 de Septiembre 2018. `https://networkx.github.io/documentation/networkx-1.10/reference/drawing.html`.