

Universidad Autónoma de Nuevo León
Facultad de Ingeniería Mecánica y Eléctrica

Posgrado en Ingeniería de Sistemas

Portafolio

Optimización de Flujo en Redes

Enero-Junio, 2019

Dania Danielles Suárez Suárez

1985272

Docente: Dr(a). Elisa Schaeffer

Correcciones realizadas:

En esta tarea se corrigieron los acentos, así como los espacios entre palabras y mejora en la redacción. Se le incluyeron las referencias a las figuras correspondientes; se modificó el estilo de letra en palabras señales. Se modificaron partes del código que tenía espacios subutilizados, así como se eliminaron acentos que generaban errores de lectura.

9.5

Tarea No 1 Representación de redes a través de la teoría de grafos

Dania Danielles Suarez Suarez

5272

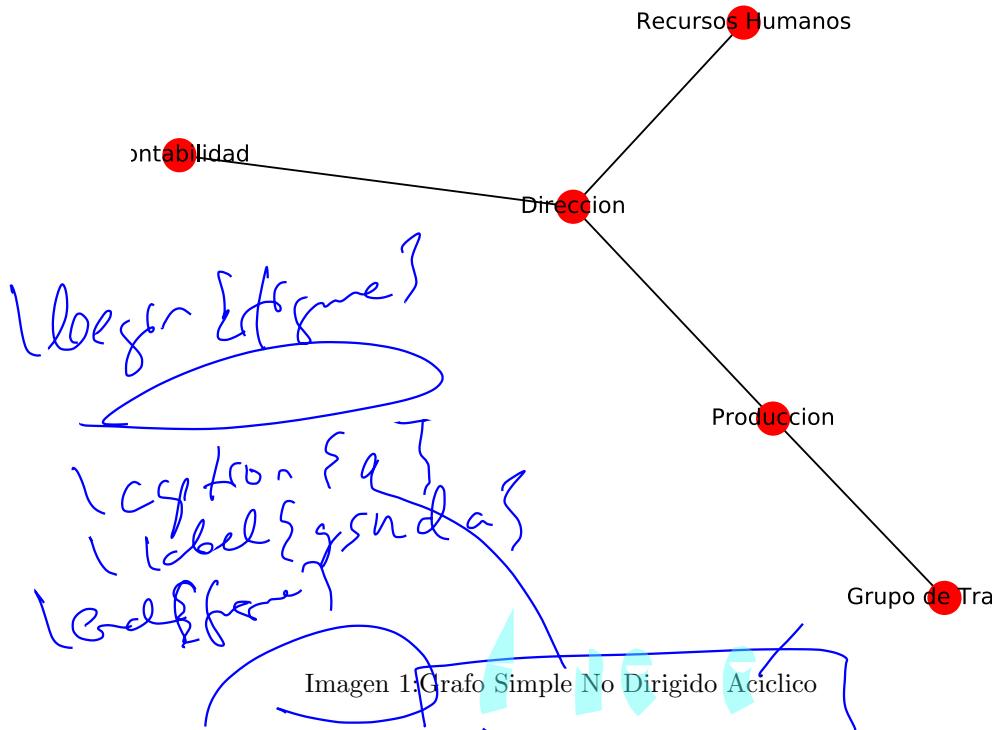
1. Grafo Simple No Dirigido Aciclico

En la Dulcería "Dulce Amor", el organigrama que representa la estructura jerárquica de la empresa está compuesto por un nodo raíz el cual sería la Dirección, de esta se derivan los departamentos de Producción, Contabilidad y Recursos Humanos; estos serían los nodos hijos y del departamento de Producción se desprende un nodo externo llamado grupo de trabajo [6].

```

1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.Graph() #Se crea un grafo vacio
5
6 G.add_node("Direccion") #Se crea el nodo raíz
7 G.add_nodes_from(["Produccion", "Contabilidad", "Recursos Humanos", "Grupo de Trabajo"])
8
9 G.add_edges_from([( "Direccion", "Produccion"), ("Direccion", "Contabilidad"), ("Direccion", "Recursos Humanos")])
10 G.add_edges_from([( "Produccion", "Grupo de Trabajo")])
11
12 nx.draw(G, with_labels=True) #Se dibuja el grafo
13 plt.savefig("Tarea1_01.eps")
14 plt.show() #Se dibuja en pantalla
    
```

Tarea1_01.py



La figura \ref{fig:ndg} muestra

2. Grafo Simple No Dirigido Ciclico

Se tiene un grupo de ordenadores los cuales se quieren conectar entre ellos en una oficina, aquí los nodos serían las PC mientras los cables que la conectan sería una arista [1].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.Graph() #Se crea un grafo vacio
5
6 G.add_node(1)
7 G.add_nodes_from([2,3,4,5,6])
8 G.add_edges_from([(1,2),(1,3),(1,4),(1,5),(1,6)])
9 G.add_edges_from([(2,3)])
10 G.add_edges_from([(3,4)])
11 G.add_edges_from([(4,5)])
12 G.add_edges_from([(5,6)])
13 G.add_edges_from([(6,2)])
14
15 nx.draw(G, with_labels=True) #Se dibuja el grafo
16 plt.savefig("Tarea1_02.eps")
17 plt.show() #Se dibuja en pantalla
```

Tarea1_02.py

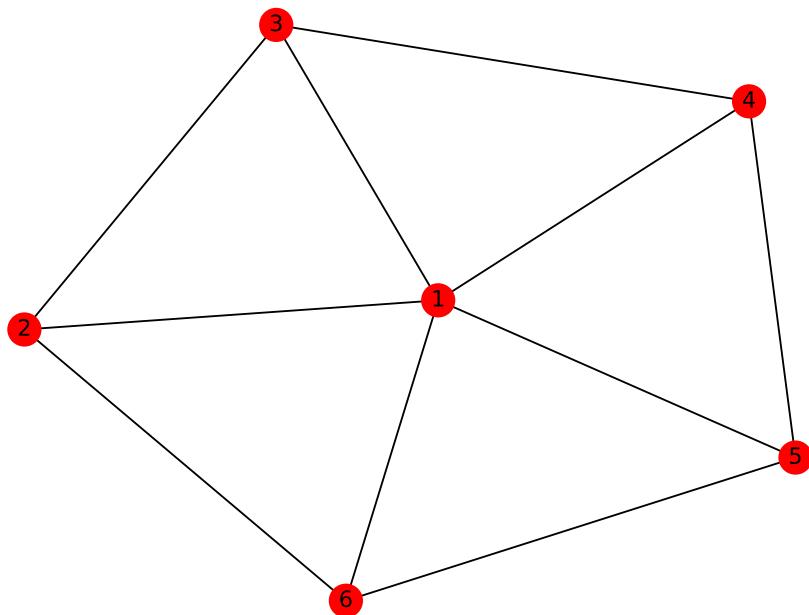


Imagen 2:Grafo Simple No Dirigido Ciclico

3. Grafo Simple No Dirigido Reflexivo

En un partido de football los jugadores serían los nodos, los cuales los pases que se realizan entre ellos conformarían las aristas, el jugador X realiza un autopase antes de darle continuidad a la pelota [4].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.Graph() #Se crea un grafo vacio
5
6 G.add_node(1) #Se crea el nodo raÃ§z
7 G.add_nodes_from([2,3,4])
8
9 G.add_edges_from([(1,2),(2,3),(3,4),(2,2),(4,1),(2,4)])
10
11 nx.draw(G, with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea1_03.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea1_03.py

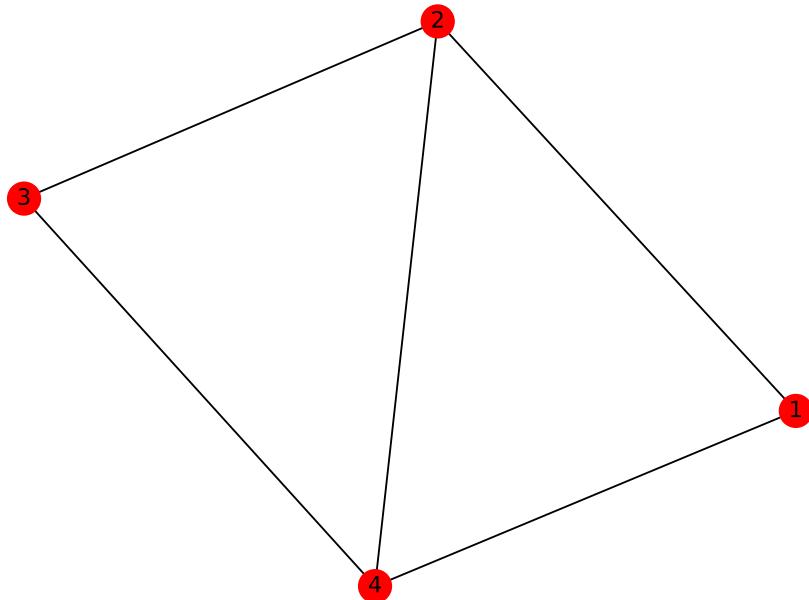


Imagen 3:Grafo Simple No Dirigido Reflexivo

4. Grafo Simple Dirigido Aciclico

Una colección de tareas que deben ordenarse con cierta secuencia, esta está sujeta a limitaciones dado que una tarea no puede realizarse antes que otras; tendría un vértice para cada tarea y un borde para cada restricción, la forma en que se puede ordenar es de manera topológica [12].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.DiGraph() #Se crea un grafo vacio
5
6 G.add_node(1)
7 G.add_nodes_from([3,2,4,5])
8 G.add_edges_from([(1,2),(1,3),(1,5),(1,5)])
9 G.add_edges_from([(2,4)])
10 G.add_edges_from([(3,2),(3,5),(4,5)])
11 G.add_edges_from([(4,5)])
12
13 nx.draw(G, with_labels=True) #Se dibuja el grafo
14 plt.savefig("Tarea1_04.eps")
15 plt.show() #Se dibuja en pantalla
```

Tarea1_04.py

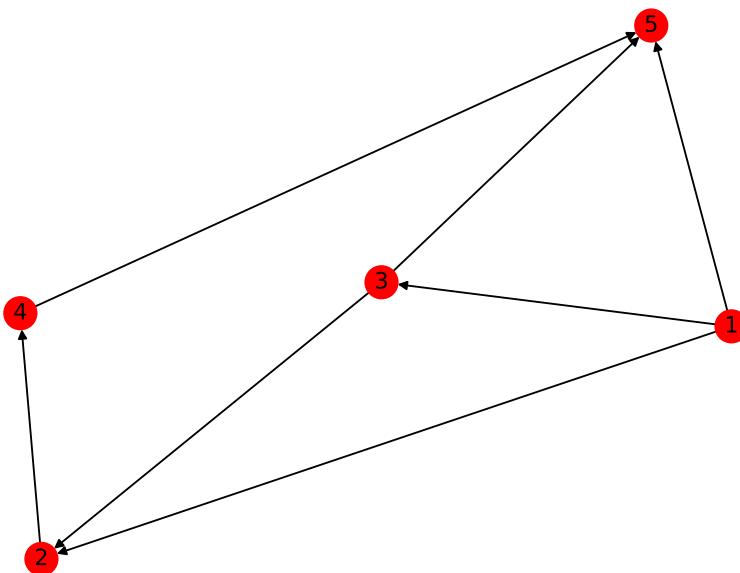


Imagen 4:Grafo Simple Dirigido Aciclico

5. Grafo Simple Dirigido Ciclico

Algo

Una Empresa "X" que se dedica a comercializar productos de limpieza necesita hacer la distribución de la mercancía, para esto se requiere encontrar un recorrido que garantice que se recorrerá todos los nodos de la red de modo que se minimice la distancia total recorrida y al final de la distribución el método de transporte vuelva al origen [8].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.DiGraph()
5
6 G.add_node("X")
7 G.add_nodes_from([1,2,3,4,5,6])
8
9 G.add_edges_from([( "X" ,2),(2,4),(4,6),(6,5),(5,3),(3,1),(1,"X")])
10
11 nx.draw(G, with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea1_05.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea1_05.py

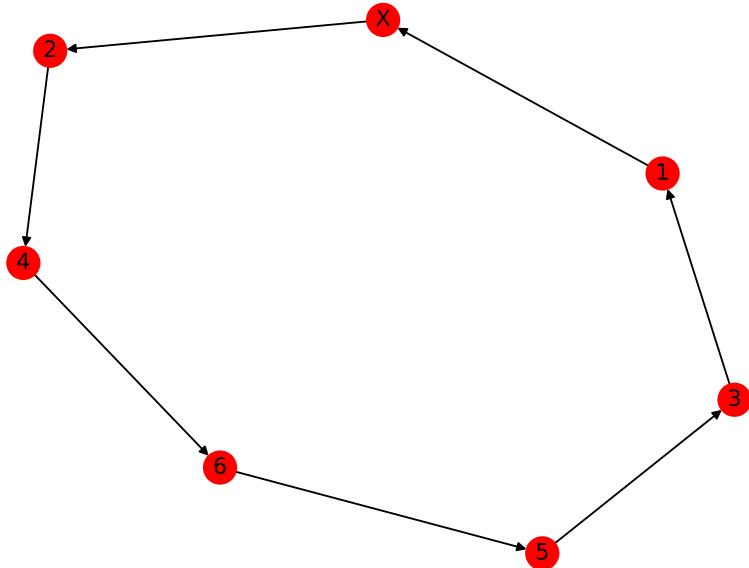


Imagen 5:Grafo Simple Dirigido Ciclico

$$A = \{x, y, z\}$$

6. Grafo Simple Dirigido Reflexivo

Sea el conjunto $A = \{x, y, z\}$. El grafo representa una relación binaria definida en A , puesto que los pares (x, z) , (y, x) (y, y) constituyen un subconjunto de $A \times A$ [11].

```

1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.DiGraph() #Se crea un grafo vacio
5
6 G.add_node("x")
7 G.add_nodes_from(["y", "z", "w", "s", "t"])
8 G.add_edges_from([( "t", "s"), ("s", "w"), ("s", "z"), ("z", "x"), ("x", "y"), ("y", "y")])
9
10 nx.draw(G, with_labels=True) #Se dibuja el grafo
11 plt.savefig("Tarea1_06.eps")
12 plt.show() #Se dibuja en pantalla

```

Tarea1_06.py

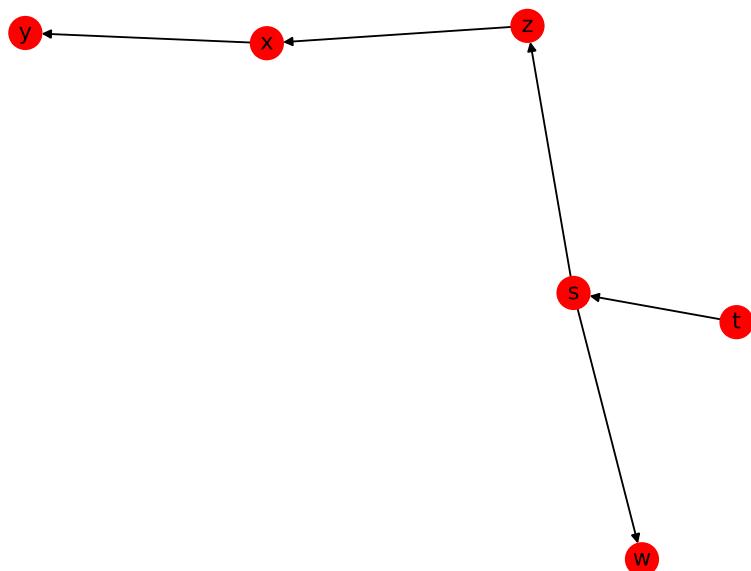


Imagen 6: Grafo Simple Dirigido Reflexivo

7. Multigrafo No Dirigido Aciclico

Un corredor sale desde su punto de partida pasando por varias paradas, al llegar a la primera parada hay dos alternativas de rutas para llegar al próximo destino, en este ejemplo las paradas constituyen los nodos mientras el recorrido entre los nodos son las aristas [2].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiGraph() #Se crea un grafo vacio
5
6 G.add_node("A") #Se crea el nodo raíz
7 G.add_nodes_from(["B", "C", "D", "E"])
8
9 G.add_edges_from([( "A" , "B" ),( "B" , "C" ),( "B" , "C" ),( "C" , "D" ),( "D" , "E" )])
10
11 nx.draw(G, with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea1_07.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea1_07.py

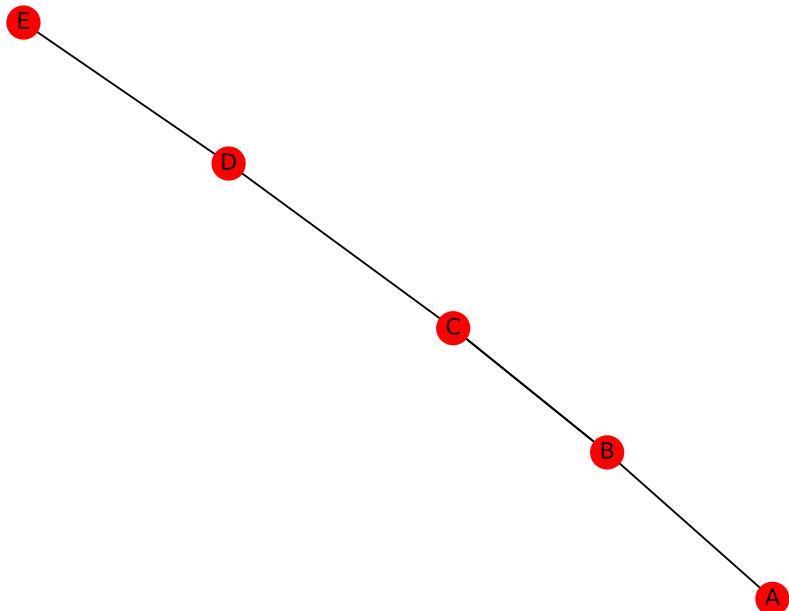


Imagen 7:Multigrafo No Dirigido Aciclico

8. Multigrafo No Dirigido Ciclico

Las llamadas telefónicas es una aplicación en la vida práctica de los grafos dado que estas se pueden representan mediante una red; cada número telefónico sería un nodo y cada llamada sería una arista, esta sale desde el número de teléfono que hace la llamada hasta el número que la recibe [7].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G=nx.MultiDiGraph()
5
6 G.add_node(1)
7 G.add_nodes_from([2,3,4,5])
8 H=nx.Graph()
9 G.add_edges_from([(1,2),(2,1),(1,3),(4,5),(4,1)])
10
11 nx.draw(G, with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea1_08.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea1_08.py

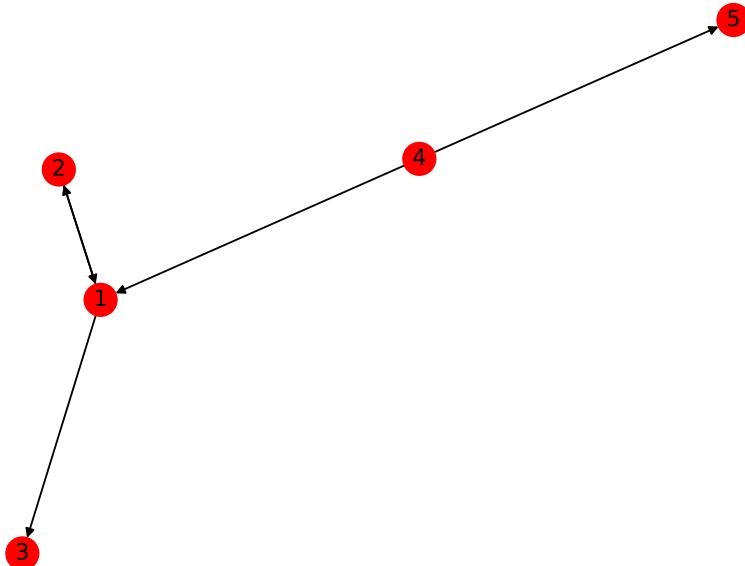


Imagen 8: Multigrafo No Dirigido Ciclico

9. Multigrafo No Dirigido Reflexivo

En el juego del Policias y Ladrones el laberinto que conforma este juego está representado con un grafo, donde las calles son las aristas donde se mueven los policías y los ladrones de vértice en vértice los cuales están ubicados en las intersecciones de las calles. Este juego se juega en grafos reflexivos, que son aquellos cuyos vértices tienen bucles, de esta forma, los jugadores pueden permanecer en el mismo vértice varias rondas seguidas y también permite que contenga aristas múltiples [9].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiDiGraph() #Se crea un grafo vacio
5
6 G.add_node(1) #Se crea el nodo raz
7 G.add_nodes_from([2,3,4,5,6,7,8,9])
8
9 G.add_edges_from([(1,2),(2,1),(2,3),(3,2),(3,4),(4,3),(3,5),(5,3),(5,6),(6,5),(6,7),(7,6),
10 ,(7,8),(8,7),(7,9),(9,7)])
11 nx.draw(G, with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea1_09.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea1_09.py

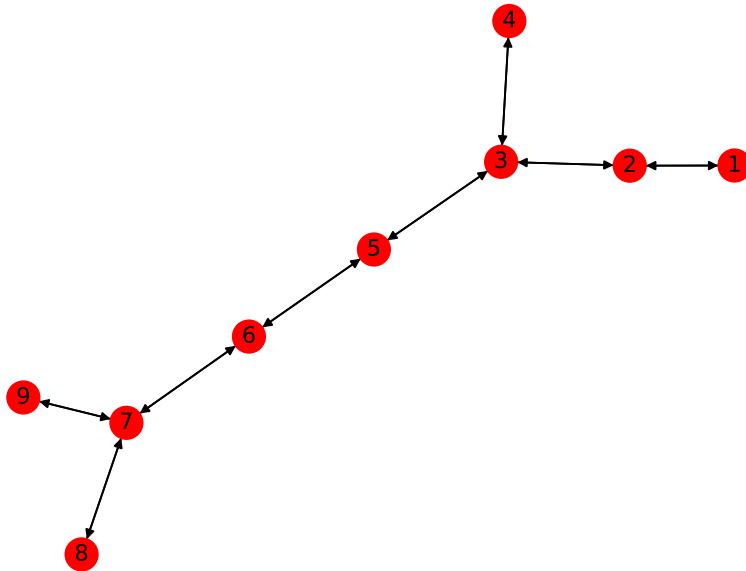


Imagen 9: Multigrafo No Dirigido Reflexivo

10. Multigrafo Dirigido Acíclico

Se puede considerar una representación grafica de un mapa de carreteras en el cual una arista (con flecha de sentido) entre dos ciudades corresponde a un carril en una autopista entre las dos ciudades. Como a menudo hay autopistas de varios carriles entre pares de ciudades, esta representación origina un multígrafo [3].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiDiGraph() #Se crea un grafo vacio
5
6 G.add_node("A") #Se crea el nodo raíz
7 G.add_nodes_from(["C", "B", "R", "G"])
8
9 G.add_edges_from([( "A" , "C" ),( "C" , "A" ),( "C" , "B" ),( "B" , "C" ),( "B" , "R" ),( "R" , "B" ),( "R" , "G" )])
10
11 nx.draw(G, with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea1_10.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea1_10.py

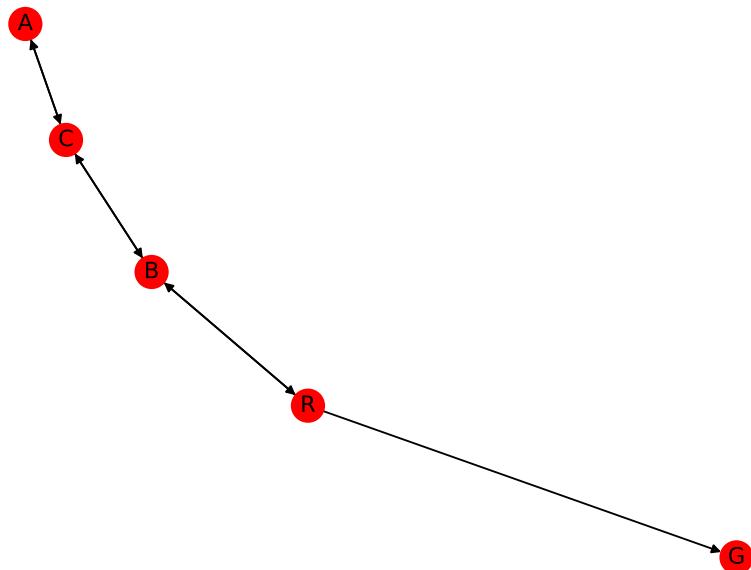


Imagen 10:Multigrafo Dirigido Aciclico

11. Multigrafo Dirigido Ciclico

Para modelar las posibles conexiones de vuelo ofrecidas por una aerolínea se tendría un multígrafo dirigido, donde cada nodo es una localidad y donde pares de aristas paralelas conectan estas localidades, según un vuelo es hacia o desde una localidad a la otra [10].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiDiGraph() #Se crea un grafo vacio
5
6 G.add_node(1) #Se crea el nodo tañaz
7 G.add_nodes_from([2,3,4,5])
8
9 G.add_edges_from([(1,2),(2,1),(2,3),(3,4),(4,5),(5,2)])
10
11 nx.draw(G, with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea1_11.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea1_11.py

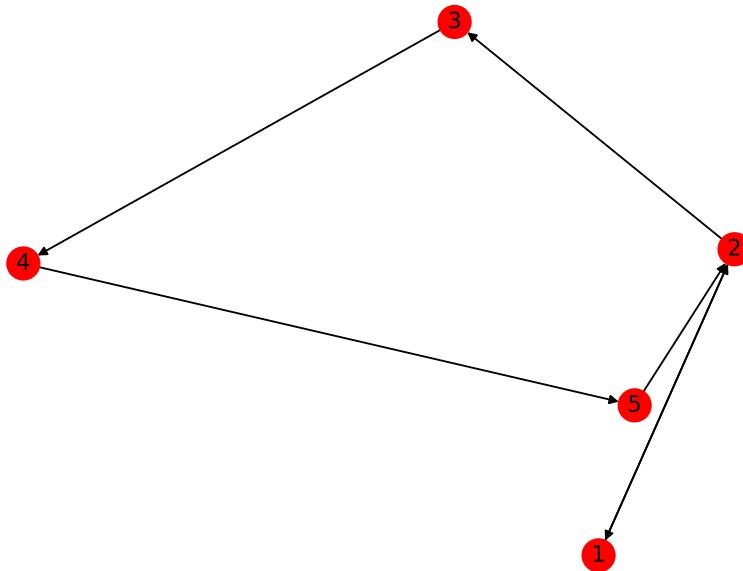


Imagen 11:Multigrafo Dirigido Ciclico

12. Multigrafo Dirigido Reflexivo

Una ruta de camión empieza su recorrido en la mañana donde tiene dos alternativas de rutas para comenzar, cuando llega al final del recorrido descarga al personal y comienza nuevamente su recorrido en ese mismo lugar con personal que se encuentre en esa parada. Los nodos ~~así~~ serían las paradas y las aristas las calles que unen cada parada [5].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiDiGraph() #Se crea un grafo vacio
5
6 G.add_node(1) #Se crea el nodo ranA z
7 G.add_nodes_from([2,3,4,5])
8
9 G.add_edges_from([(1,2),(1,2),(2,3),(3,4),(3,5),(4,5),(5,1)])
10
11 nx.draw(G, with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea1_12.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea1_12.py

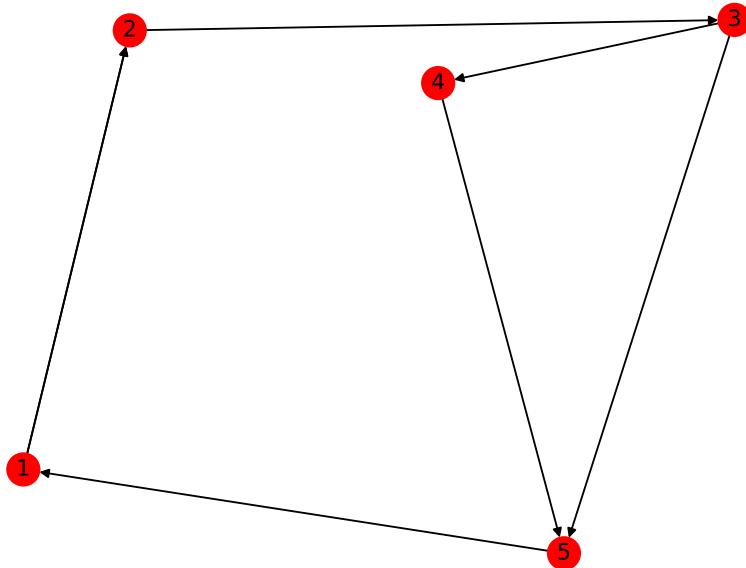


Imagen 12: Multigrafo Dirigido Reflexivo

Análisis

Referencias

- [1] Armando de Jesus Ruiz Calderon, Sofia Barron Perez, Abel Gonzalez Canas, Daniel Roberto Del Toro Arias, and Argel Frias Escudero. Modelado de grafos en la red. ?
- [2] ESTER MARTOS CARRION. *Analisis sobre las nuevas formas de comunicacion a traves de las comunidades virtuales o redes sociales*. PhD thesis, 2011.
- [3] Maria del Carmen Somoza Lopez. *Estructuras metricas en grafos*. PhD thesis, Universidade de Vigo, 2015.
- [4] Natalia Martinez, Gheisa Ferreira, and Zoila Zenaida Garcia. Sistema de enseñanza/aprendizaje inteligente para grafos. 2007.
- [5] Johani Olivos Iparraguirre. Algoritmos para caminos minimos. 2009.
- [6] Jorge J Frias Perles. Grafos:las redes que mueven el mundo. ?, *SAC*
- [7] CAROLINA RAMIREZ. Aplicaciones de grafos. ?
- [8] Antonio Rifon Sanchez. Los ordenes semanticos. 2009.
- [9] Blanca Maria Pozuelo Rollon. Juego de policias y ladrones en grafos. ?
- [10] Jorge E SAGULA and Rene J TESEYRA. Algoritmo de busqueda de rutas con preferencias. 2011.
- [11] Andres Ramos Pedro Linares Pedro Sanchez Angel Sarabia, Begona Vitoriano. Teoria de grafos o redes. ?
- [12] Fabiola Werlinger and Dante D Caceres. Aplicacion de grafos aciclicos dirigidos en la evaluacion de un set minimo de ajuste de confusores: un complemento al modelamiento estadistico en estudios epidemiologicos observacionales. *Revista medica de Chile*, 146(7):907–913, 2018.

Tarea 1

5272

1. Grafo simple no dirigido acíclico

En la Dulcería "Dulce Amor", el organigrama que representa la estructura jerárquica de la empresa está compuesto por un nodo raíz que es la Dirección de la empresa; de esta se derivan los departamentos de Producción, Contabilidad y Recursos Humanos; estos serían los nodos hijos y del departamento de Producción se desprende un nodo externo llamado grupo de trabajo [6] .

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.Graph() #Se crea un grafo vacio
5
6 G.add_node("Direccion") #Se crea el nodo raiz
7 G.add_nodes_from(["Produccion","Contabilidad","Recursos Humanos","Grupo de Trabajo"])
8
9 G.add_edges_from([( "Direccion", "Produccion"),( "Direccion", "Contabilidad"),( "Direccion", "Recursos Humanos")])
10 G.add_edges_from([( "Produccion", "Grupo de Trabajo")])
11
12 nx.draw(G, with_labels=True) #Se dibuja el grafo
13 plt.savefig("Tarea1_01.eps")
14 plt.show() #Se dibuja en pantalla
```

Tarea1_01.py

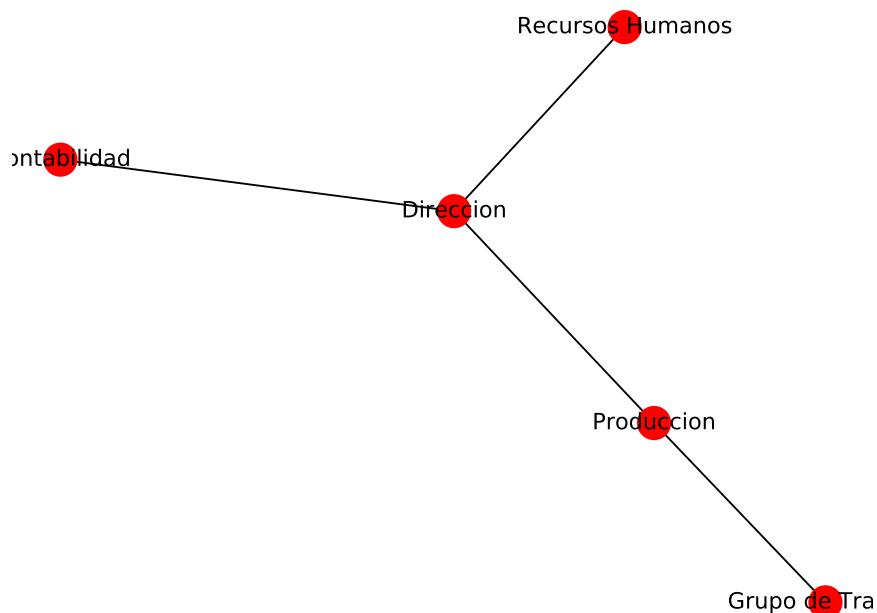


Figura 1: Grafo simple no dirigido acíclico

2. Grafo simple no dirigido cíclico

Se tiene un grupo de ordenadores los cuales se quieren conectar entre ellos en una oficina, aquí los nodos serían las PC mientras los cables que la conectan serían las aristas [1].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.Graph() #Se crea un grafo vacío
5
6 G.add_node(1)
7 G.add_nodes_from([2,3,4,5,6])
8 G.add_edges_from([(1,2),(1,3),(1,4),(1,5),(1,6)])
9 G.add_edges_from([(2,3)])
10 G.add_edges_from([(3,4)])
11 G.add_edges_from([(4,5)])
12 G.add_edges_from([(5,6)])
13 G.add_edges_from([(6,2)])
14
15 nx.draw(G, with_labels=True) #Se dibuja el grafo
16 plt.savefig("Tarea1_02.eps")
17 plt.show() #Se dibuja en pantalla
```

Tarea1_02.py

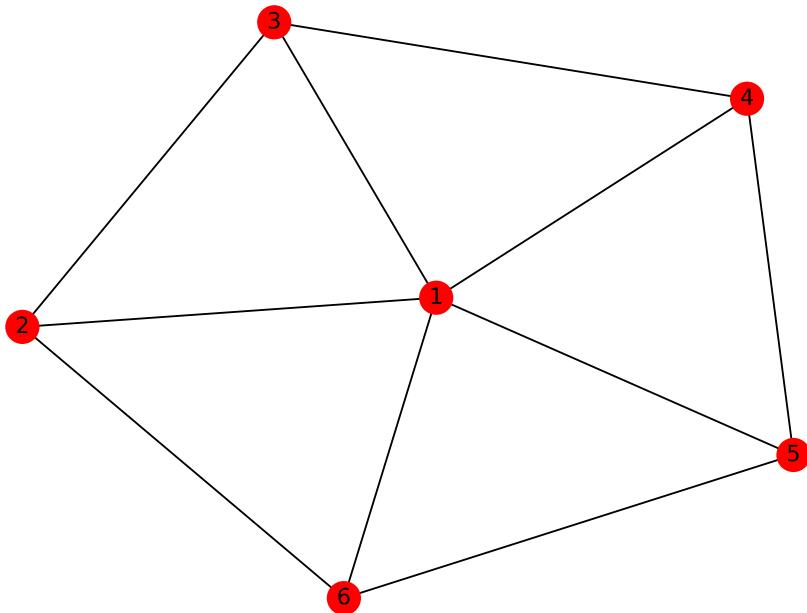


Figura 2: Grafo simple no dirigido cíclico

3. Grafo simple no dirigido reflexivo

En un partido de football los jugadores serían los nodos, los cuales los pases que se realizan entre ellos conformarían las aristas, el jugador X realiza un autopase antes de darle continuidad a la pelota [4].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.Graph() #Se crea un grafo vacio
5
6 G.add_node(1) #Se crea el nodo raÃ±z
7 G.add_nodes_from([2,3,4])
8
9 G.add_edges_from([(1,2),(2,3),(3,4),(2,2),(4,1),(2,4)])
10
11 nx.draw(G, with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea1_03.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea1_03.py

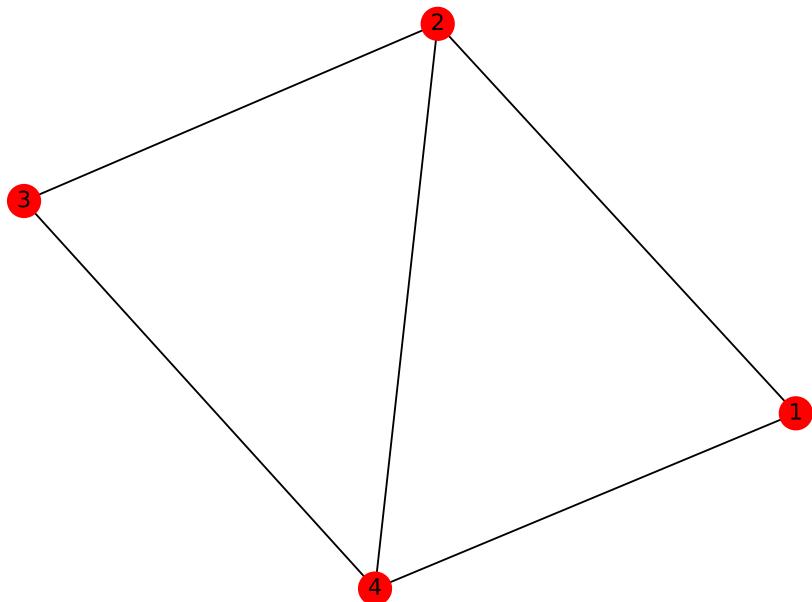


Figura 3: Grafo simple no dirigido reflexivo

4. Grafo simple dirigido acíclico

Una colección de tareas que deben ordenarse con cierta secuencia, esta está sujeta a limitaciones dado que una tarea no puede realizarse antes que otras; tendría un vértice para cada tarea y un borde para cada restricción, la forma en que se puede ordenar es de manera topológica [12].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.DiGraph() #Se crea un grafo vacio
5
6 G.add_node(1)
7 G.add_nodes_from([3,2,4,5])
8 G.add_edges_from([(1,2),(1,3),(1,5),(1,5)])
9 G.add_edges_from([(2,4)])
10 G.add_edges_from([(3,2),(3,5),(4,5)])
11 G.add_edges_from([(4,5)])
12
13 nx.draw(G, with_labels=True) #Se dibuja el grafo
14 plt.savefig("Tarea1_04.eps")
15 plt.show() #Se dibuja en pantalla
```

Tarea1_04.py

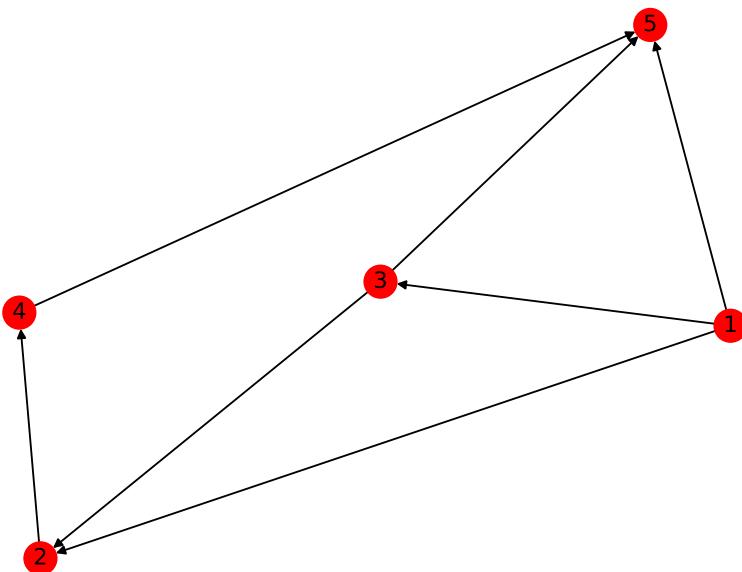


Figura 4: Grafo simple dirigido acíclico

5. Grafo simple dirigido cíclico

Una empresa X que se dedica a comercializar productos de limpieza necesita hacer la distribución de la mercancía, para esto se requiere encontrar un recorrido que garantice que se recorra todos los nodos de la red de modo que se minimice la distancia total recorrida y al final de la distribución el método de transporte vuelva al origen [8].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.DiGraph()
5
6 G.add_node("X")
7 G.add_nodes_from([1,2,3,4,5,6])
8
9 G.add_edges_from([( "X",2),(2,4),(4,6),(6,5),(5,3),(3,1),(1,"X")])
10
11 nx.draw(G, with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea1_05.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea1_05.py

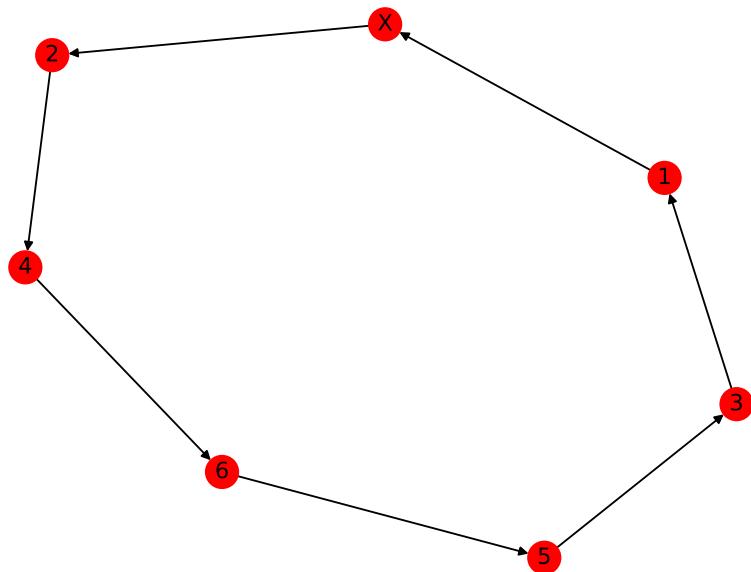


Figura 5: Grafo simple dirigido cíclico

6. Grafo simple dirigido reflexivo

Sea el conjunto $A = x, y, z$. El grafo representa una relación binaria definida en A, puesto que los pares (x, z) , (y, x) (y, y) constituyen un subconjunto de AxA [11].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.DiGraph() #Se crea un grafo vacio
5
6 G.add_node("x")
7 G.add_nodes_from([ "y" , "z" , "w" , "s" , "t" ])
8 G.add_edges_from([( "t" , "s" ),( "s" , "w" ),( "s" , "z" ),( "z" , "x" ),( "x" , "y" ),( "y" , "y" )])
9
10 nx.draw(G, with_labels=True) #Se dibuja el grafo
11 plt.savefig("Tarea1_06.eps")
12 plt.show() #Se dibuja en pantalla
```

Tarea1_06.py

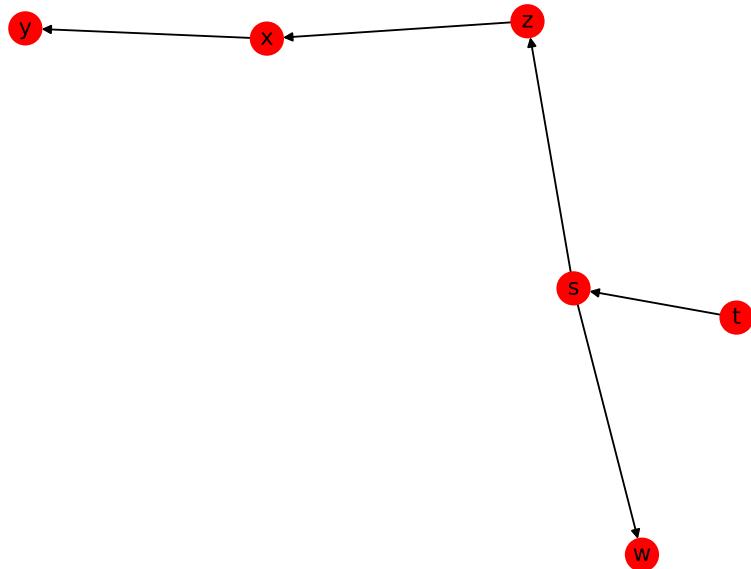


Figura 6: Grafo simple dirigido reflexivo

7. Multigrafo no dirigido acíclico

Un corredor sale desde su punto de partida pasando por varias paradas, al llegar a la primera parada hay dos alternativas de rutas para llegar al próximo destino, en este ejemplo las paradas constituyen los nodos mientras el recorrido entre los nodos son las aristas [2].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiGraph() #Se crea un grafo vacio
5
6 G.add_node("A") #Se crea el nodo raiz
7 G.add_nodes_from(["B","C","D","E"])
8
9 G.add_edges_from([( "A" , "B" ),( "B" , "C" ),( "B" , "C" ),( "C" , "D" ),( "D" , "E" )])
10
11 nx.draw(G, with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea1_07.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea1_07.py

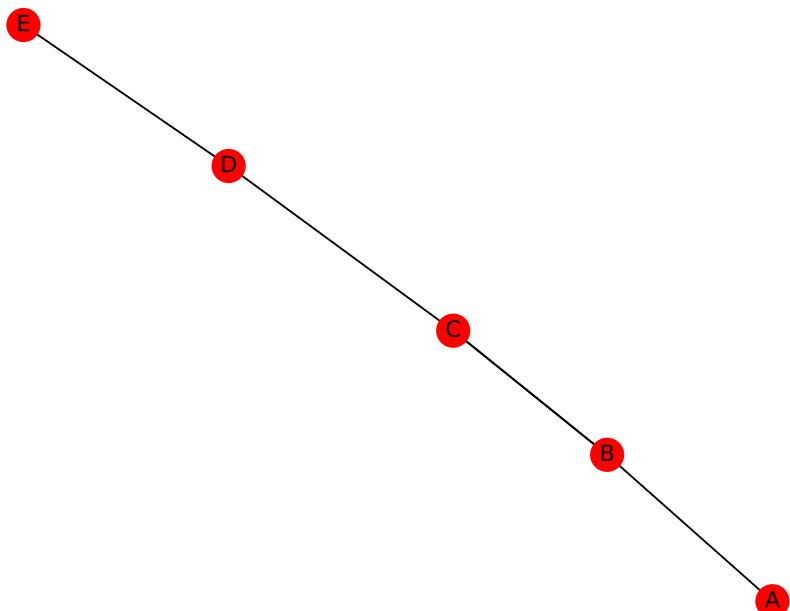


Figura 7: Multigrafo no dirigido acíclico

8. Multigrafo no dirigido cíclico

Las llamadas telefónicas es una aplicación en la vida práctica de los grafos dado que estas se pueden representan mediante una red; cada número telefónico sería un nodo y cada llamada seria una arista, esta sale desde el número de teléfono que hace la llamada hasta el número que la recibe [7].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiDiGraph()
5
6 G.add_node(1)
7 G.add_nodes_from([2, 3, 4, 5])
8 H = nx.Graph()
9 H.add_edges_from([(1, 2), (2, 1), (1, 3), (4, 5), (4, 1)])
10
11 nx.draw(G, with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea1_08.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea1_08.py

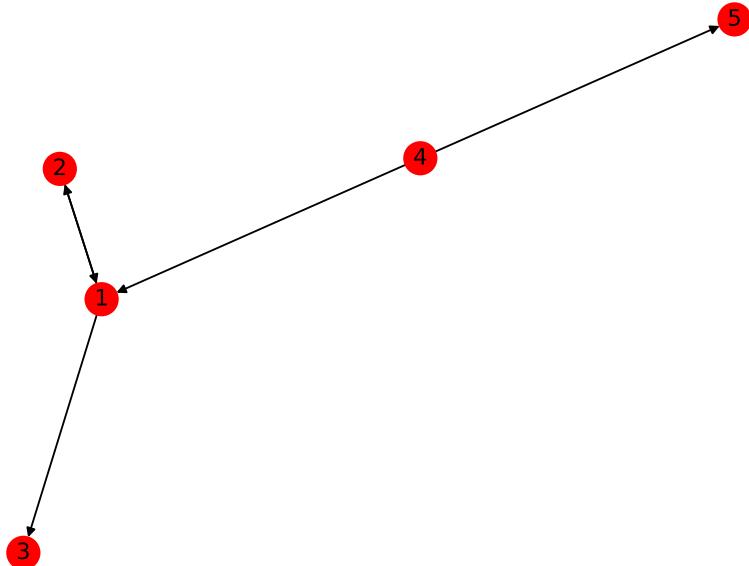


Figura 8: Multigrafo no dirigido cíclico

9. Multigrafo no dirigido reflexivo

En el juego del Policías y Ladrones el laberinto que conforma este juego está representado con un grafo, donde las calles son las aristas donde se mueven los policías y los ladrones de vértice en vértice los cuales están ubicados en las intersecciones de las calles. Este juego se juega en grafos reflexivos, que son aquellos cuyos vértices tienen bucles, de esta forma, los jugadores pueden permanecer en el mismo vértice varias rondas seguidas y también permite que contenga aristas múltiples [9].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiDiGraph() #Se crea un grafo vacio
5
6 G.add_node(1) #Se crea el nodo raiz
7 G.add_nodes_from([2,3,4,5,6,7,8,9])
8
9 G.add_edges_from([(1,2),(2,1),(2,3),(3,2),(3,4),(4,3),(3,5),(5,3),(5,6),(6,5),(6,7),(7,6),
10 ,(7,8),(8,7),(7,9),(9,7)])
11 nx.draw(G, with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea1_09.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea1_09.py

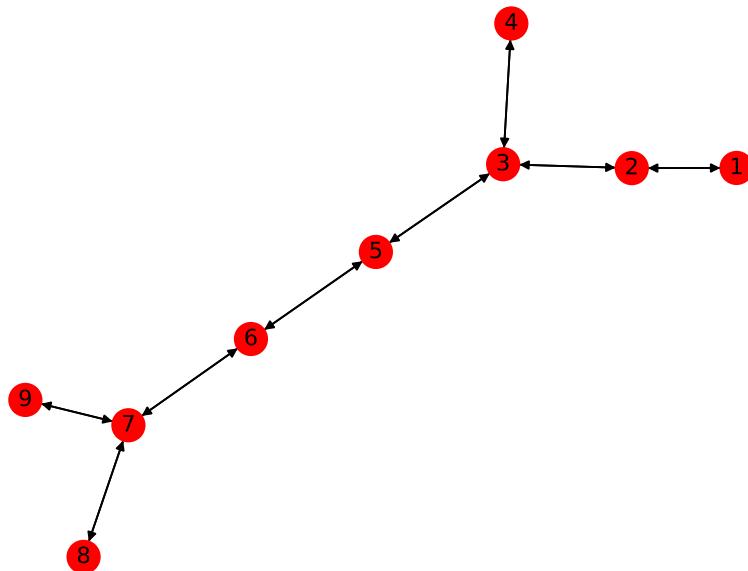


Figura 9: Multigrafo dirigido reflexivo

10. Multigrafo dirigido acíclico

Se puede considerar una representación gráfica de un mapa de carreteras en el cual una arista (con flecha de sentido) entre dos ciudades corresponde a un carril en una autopista entre las dos ciudades. Como a menudo hay autopistas de varios carriles entre pares de ciudades, esta representación origina un multígrafo [3].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiDiGraph() #Se crea un grafo vacio
5
6 G.add_node("A") #Se crea el nodo raiz
7 G.add_nodes_from(["C","B","R","G"])
8
9 G.add_edges_from([( "A" , "C" ),( "C" , "A" ),( "C" , "B" ),( "B" , "C" ),( "B" , "R" ),( "R" , "B" ),( "R" , "G" )])
10
11 nx.draw(G, with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea1_10.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea1_10.py

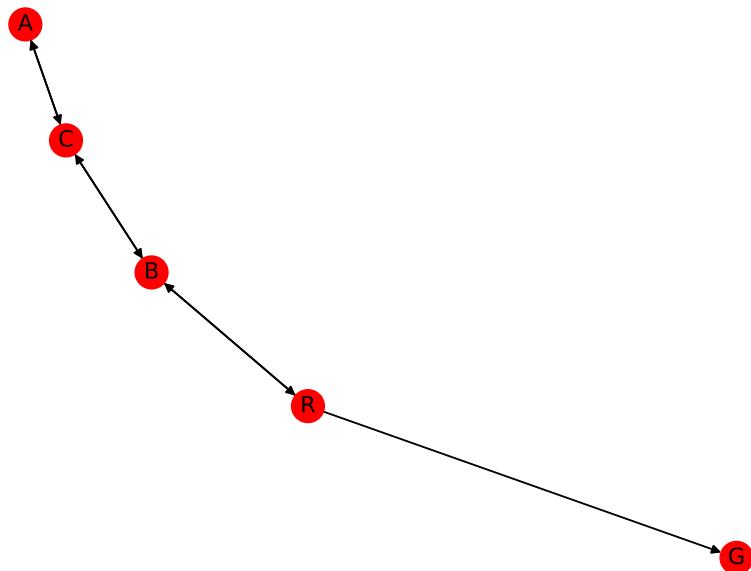


Figura 10: Multigrafo dirigido acíclico

11. Multigrafo dirigido cíclico

Para modelar las posibles conexiones de vuelo ofrecidas por una aerolínea se tendría un multígrafo dirigido, donde cada nodo es una localidad y donde pares de aristas paralelas conectan estas localidades, según un vuelo es hacia o desde una localidad a la otra [10].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiDiGraph() #Se crea un grafo vacio
5
6 G.add_node(1) #Se crea el nodo raz
7 G.add_nodes_from([2,3,4,5])
8
9 G.add_edges_from([(1,2),(2,1),(2,3),(3,4),(4,5),(5,2)])
10
11 nx.draw(G, with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea1_11.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea1_11.py

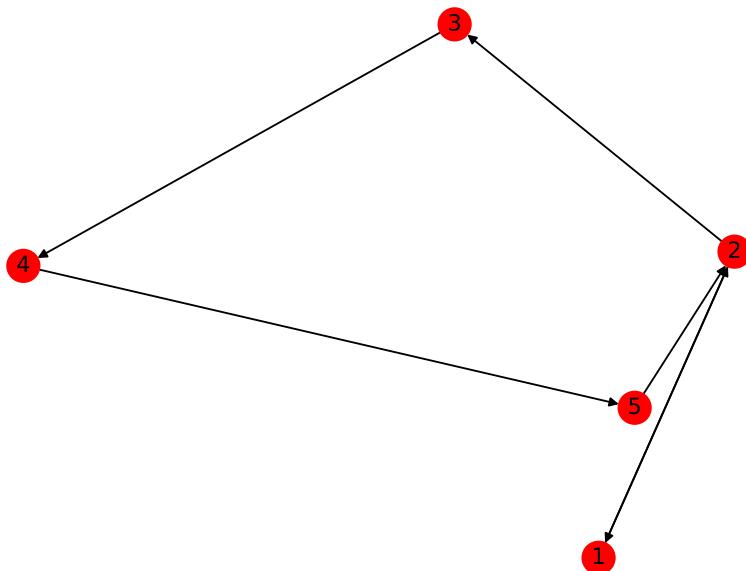


Figura 11: Multigrafo dirigido cíclico

12. Multigrafo dirigido reflexivo

Una ruta de camión empieza su recorrido en la mañana donde tiene dos alternativas de rutas para comenzar, cuando llega al final del recorrido descarga al personal y comienza nuevamente su recorrido en ese mismo lugar con personal que se encuentre en esa parada. Los nodos serian las paradas y las aristas las calles que unen cada parada [5].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiDiGraph() #Se crea un grafo vacio
5
6 G.add_node(1) #Se crea el nodo raiz
7 G.add_nodes_from([2,3,4,5])
8
9 G.add_edges_from([(1,2),(1,2),(2,3),(3,4),(3,5),(4,5),(5,1)])
10
11 nx.draw(G, with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tareal_12.eps")
13 plt.show() #Se dibuja en pantalla
```

Tareal_12.py

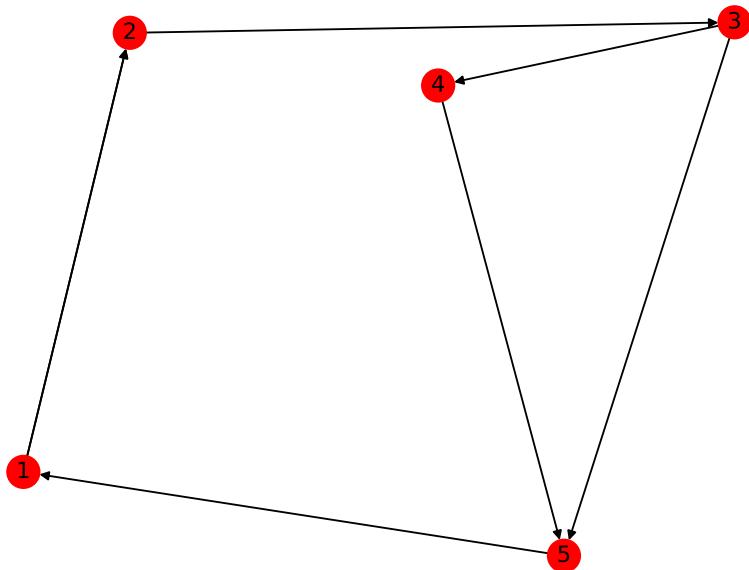


Figura 12: Multigrafo dirigido reflexivo

Referencias

- [1] Armando de Jesus Ruiz Calderon, Sofia Barron Perez, Abel Gonzalez Canas, Daniel Roberto Del Toro Arias, and Argel Frias Escudero. Modelado de grafos en la red.
- [2] ESTER MARTOS CARRION. *Analisis sobre las nuevas formas de comunicacion a traves de las comunidades virtuales o redes sociales*. PhD thesis, 2011.
- [3] Maria del Carmen Somoza Lopez. *Estructuras metricas en grafos*. PhD thesis, Universidade de Vigo, 2015.
- [4] Natalia Martinez, Gheisa Ferreira, and Zoila Zenaida Garcia. Sistema de enseñanza/aprendizaje inteligente para grafos. 2007.
- [5] Johani Olivos Iparraguirre. Algoritmos para caminos minimos. 2009.
- [6] Jorge J Frias Perles. Grafos: las redes que mueven el mundo.
- [7] CAROLINA RAMIREZ. Aplicaciones de grafos.
- [8] Antonio Rifon Sanchez. Los ordenes semanticos. 2009.
- [9] Blanca Maria Pozuelo Rollon. Juego de policias y ladrones en grafos.
- [10] Jorge E SAGULA and Rene J TESEYRA. Algoritmo de busqueda de rutas con preferencias. 2011.
- [11] Andres Ramos Pedro Linares Pedro Sanchez Angel Sarabia Begoña Vitoriano. *Teoria de grafos o redes*.
- [12] Fabiola Werlinger and Dante D Caceres. Aplicacion de grafos aciclicos dirigidos en la evaluacion de un set minimo de ajuste de confusores: un complemento al modelamiento estadistico en estudios epidemiologicos observacionales. *Revista medica de Chile*, 146(7):907–913, 2018.

Correcciones realizadas:

En esta tarea se corrigieron los acentos, así como los espacios entre palabras y mejora en la redacción. Se le incluyeron las referencias a las figuras correspondientes; se modificó el estilo de letra en palabras señales. Se modificaron partes del código que tenía espacios subutilizados, así como se eliminaron acentos que generaban errores de lectura.

Tarea No. 2

5272



1. Circular Layout

Este algoritmo de diseño como su nombre lo indica puede ser útil cuando se desea estructurar la información con un patrón circular y se trata de disminuir los cruces entre los vértices [2]; es aplicado en redes sociales y administración de redes, con este se puede realizar estructuras de grupos y árbol dentro de una red [6].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4
5 G = nx.DiGraph()
6
7
8 G.add_node("X")
9 G.add_nodes_from([1,2,3,4,5,6])
10
11 G.add_edges_from([( "X" ,2),(2,4),(4,6),(6,5),(5,3),(3,1),(1,"X")])
12
13 nx.draw(G, pos=nx.circular_layout(G), with_labels= True) # Dibuja el grafo G en forma
14 #plt.savefig("Tarea2_01.eps")
15 plt.show() #Se dibuja en pantalla
```

Tarea2_01.py

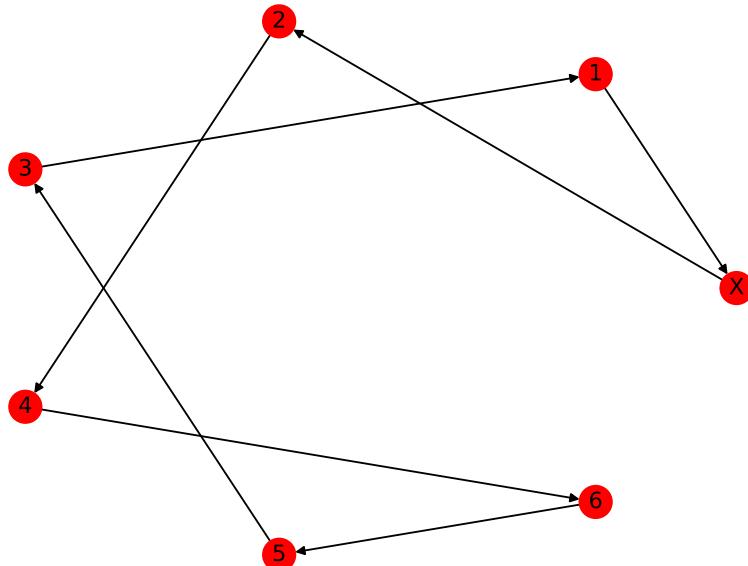


Figura 1: Circular Layout

2. Kamada-Kawai Layout

Este algoritmo es para grafos conectados y no dirigidos; se relaciona con un sistema de resorte dinámico donde la fuerza entre dos vértices es inversamente proporcional al cuadrado de la distancia entre estos ,es decir, los vértices que tienen los resortes más fuertes se encuentran más cerca [1].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.Graph() #Se crea un grafo vacio
5
6 G.add_node(1) #Se crea el nodo raÃ±z
7 G.add_nodes_from([2,3,4])
8
9 G.add_edges_from([(1,2),(2,3),(3,4),(2,2),(4,1),(2,4)])
10
11 color_map = []
12 for node in G:
13     if (node==2):
14         color_map.append('blue')
15     else:
16         color_map.append('green')
17
18 nx.draw(G, pos=nx.kamada_kawai_layout(G), node_color=color_map, with_labels=True) #Se dibuja el
19 #grafo
20 plt.savefig("Tarea2_02.eps")
21 plt.show() #Se dibuja en pantalla
```

Tarea2_02.py

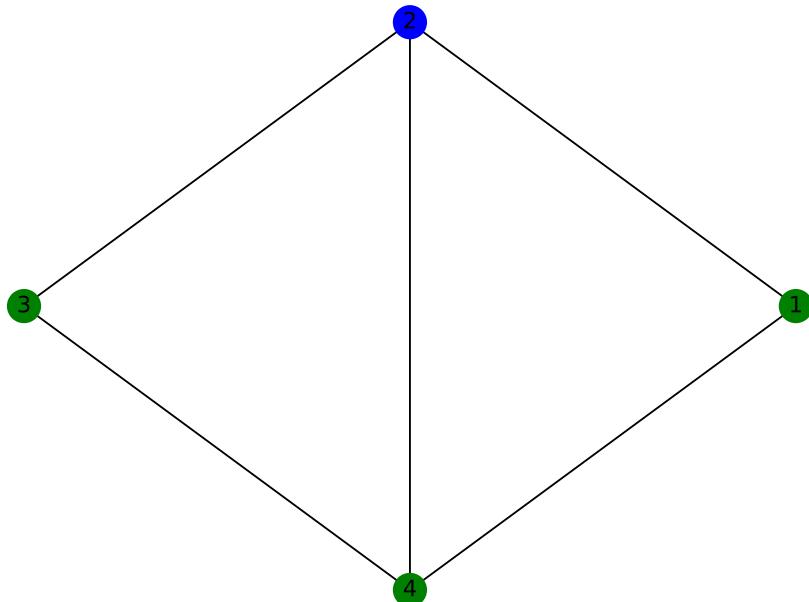


Figura 2: Kamada-Kawai Layout

3. Random Layout

Este diseño se utiliza para cualquier tipo de grafo conectado y no conectado, planos y no planos. Se caracteriza por darle un orden aleatorio a los nodos de un grafo y tiene como limitación que para asegurarse de que los nodos no se superpongan con los margenes de la región del diseño, este algoritmo calcula las coordenadas al azar dentro de una región con las dimensiones más pequeñas que la región del diseño [11].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiGraph() #Se crea un grafo vacío
5
6 G.add_node("A") #Se crea el nodo A
7 G.add_nodes_from([ "B" , "C" , "D" , "E" ])
8
9 G.add_edge("B" , "C" , color='red' , weight=6)
10 G.add_edges_from([( "A" , "B" ),( "B" , "C" ),( "C" , "D" ),( "D" , "E" )] , color='blue' , weight=2)
11
12 edges = G.edges()
13
14 colors = []
15 weight = []
16
17 for (u,v,attrib_dict) in list(G.edges.data()):
18     colors.append(attrib_dict['color'])
19     weight.append(attrib_dict['weight'])
20
21 pos=nx.random_layout(G)
22
23 nx.draw(G, pos , edges=edges , edge_color=colors , width=weight , with_labels=True , font_size=8) #Se
24     dibuja el grafo
25 plt.savefig("Tarea2_02.eps")
26 plt.show() #Se dibuja en pantalla
```

Tarea2_03.py

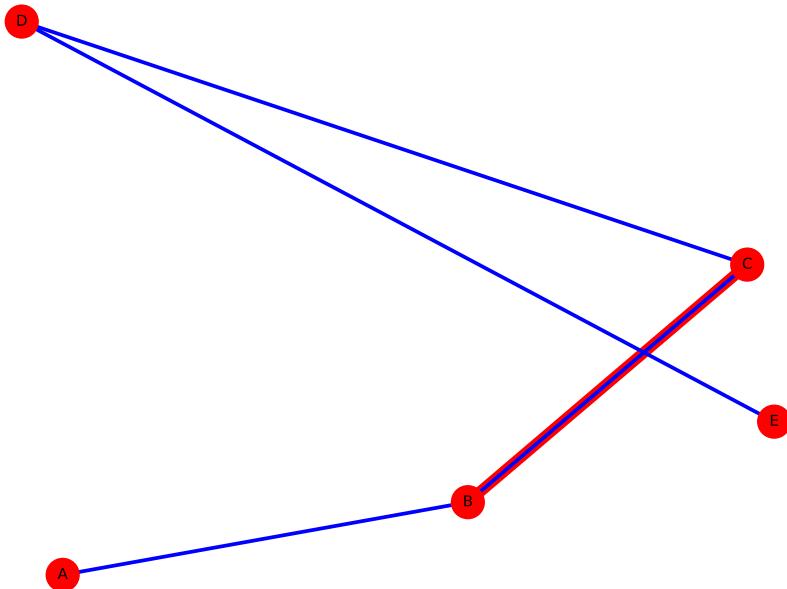


Figura 3: Random Layout

```

1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiDiGraph() #Se crea un grafo vacio
5
6 G.add_node("A") #Se crea el nodo raíz
7 G.add_nodes_from(["C", "B", "R", "G"])
8
9 G.add_edges_from([( "A", "C") ,("C", "A") ,("C", "B") ,("B", "C") ,("B", "R") ,("R", "B") ,("R", "G")])
10
11 nx.draw(G, pos=nx.random_layout(G), with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea2_10.eps")
13 plt.show() #Se dibuja en pantalla

```

Tarea2_10.py

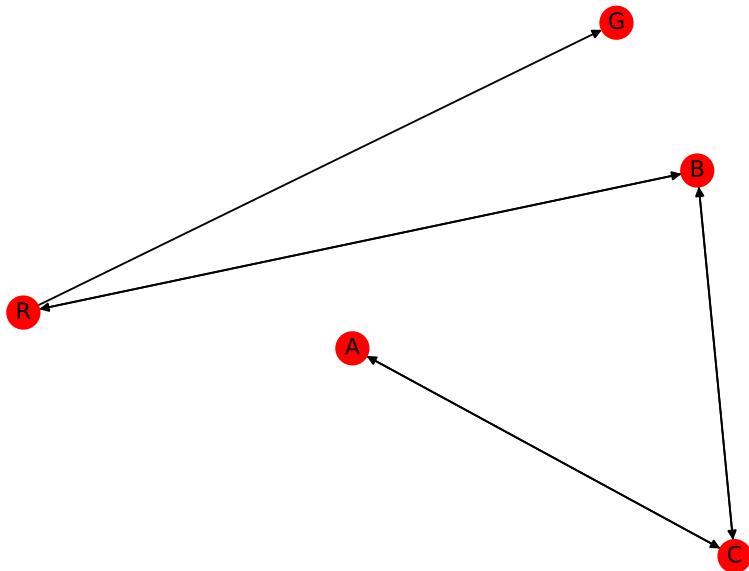


Figura 4: Random Layout

4. ForceAtlas2 Layout

En este diseño los nodos se rechazan entre sí,mientras que los bordes lo atraen como resortes,la posición de cada nodo depende de los otros nodos [9].

```
1 import networkx as nx
2 from fa2 import ForceAtlas2
3 import matplotlib.pyplot as plt
4
5 G = nx.Graph()
6
7 G.add_node(1)
8 G.add_nodes_from([2, 3, 4, 5, 6, 7, 8, 9, 10])
9
10 G.add_edges_from([(1, 10), (2, 10), (10, 3), (10, 4), (4, 5), (4, 6), (4, 7), (7, 4), (7, 8),
11 (7, 9)])
12 #Este fragmento de código fue tomado de https://github.com/bhargavchippada/forceatlas2/blob/master/examples/forceatlas2-layout.ipynb
13
14 forceatlas2 = ForceAtlas2(
15
16     outboundAttractionDistribution=True,
17     linLogMode=False,
18     adjustSizes=False,
19     edgeWeightInfluence=1.0,
20
21     jitterTolerance=1.0,
22     barnesHutOptimize=True,
23     barnesHutTheta=1.2,
24     multiThreaded=False,
25
26     scalingRatio=2.0,
27     strongGravityMode=False,
28     gravity=1.0,
29
30     verbose=True)
31
32 positions = forceatlas2.forceatlas2_networkx_layout(G, pos=None, iterations=2000)
33 nx.draw_networkx_nodes(G, positions, node_size=30, with_labels=False, node_color="green",
34 alpha=0.7)
35 nx.draw_networkx_edges(G, positions, edge_color="blue", alpha=0.06)
36 plt.axis('off')
37 plt.savefig("Tarea2_10.eps")
38 plt.show()
```

Tarea2_4.py

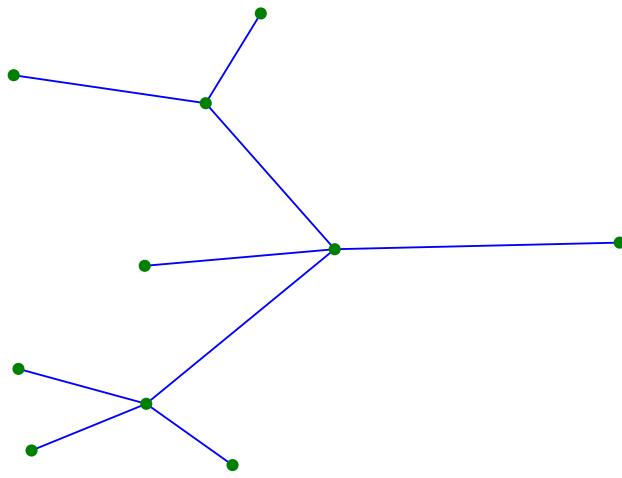
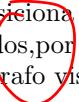


Figura 5: ForceAtlas2 Layout

5. Shell Layout

Este diseño posciona los nodos en círculos concéntricos [10], basandose en la distancia entre el nodo central al resto de los nodos, por lo que los nodos se distribuyen de forma circular; este tiene como ventaja que su diseño hace que sea un grafo visualmente de fácil entender [4] 

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.Graph() #Se crea un grafo vacio
5
6 G.add_node(1)
7 G.add_nodes_from([2,3,4,5,6])
8 G.add_edges_from([(1,2),(1,3),(1,4),(1,5),(1,6)])
9 G.add_edges_from([(2,3)])
10 G.add_edges_from([(3,4)])
11 G.add_edges_from([(4,5)])
12 G.add_edges_from([(5,6)])
13 G.add_edges_from([(6,2)])
14
15 shells=[[1],[2,3,4,5,6]]
16
17 nx.draw(G, pos=nx.shell_layout(G, shells), with_labels=True) #Se dibuja el grafo
18 plt.savefig("Tarea2_05.eps")
19 plt.show() #Se dibuja en pantalla
```

Tarea2_05.py

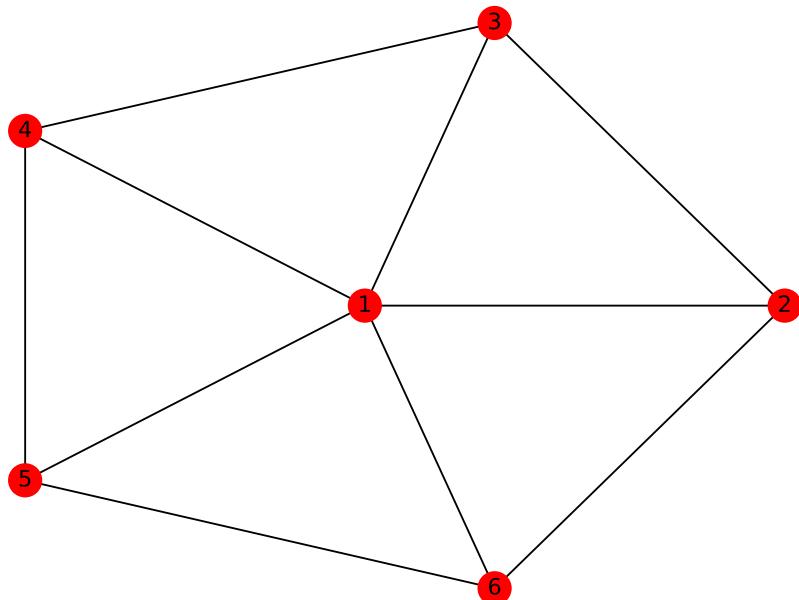


Figura 6: Shell Layout

6. Kamada Kawai Layout

Este algoritmo es dirigido por la fuerza entre dos nodos cualesquiera, donde los mismo se representan por anillos de acero y las aristas son los resortes entre ellos. Las fuerzas de atracción y de repulsión son análogas a la fuerza de resorte, donde la suma de las fuerzas determinan en que dirección se debe mover un nodo [5]; es utilizado en grafos no dirigidos muy grandes y garantiza que los nodos cercanos sean ubicados en la misma vecindad y los lejanos se ubican uno lejos de otros.

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiGraph() #Se crea un grafo vacio
5
6 G.add_node(1) #Se crea el nodo raÃ±z
7
8 G.add_nodes_from([2,3,4,5,6,7,8,9])
9 G.add_edge(1,2,color='blue',weight=4)
10 G.add_edge(3,2,color='blue',weight=4)
11 G.add_edge(3,4,color='blue',weight=4)
12 G.add_edge(3,5,color='blue',weight=4)
13 G.add_edge(6,5,color='blue',weight=4)
14 G.add_edge(6,7,color='blue',weight=4)
15 G.add_edge(8,7,color='blue',weight=4)
16 G.add_edge(8,7,color='blue',weight=4)
17 G.add_edge(9,7,color='blue',weight=4)
18
19 G.add_edges_from([(2,1),(2,3),(4,3),(5,3),(5,6),(7,6),(7,8),(9,7)],color='red',weight=1)
20
21 edges=G.edges()
22
23 colors=[]
24 weight=[]
25
26 for (u,v,attrib_dict) in list(G.edges.data()):
27     colors.append(attrib_dict['color'])
28     weight.append(attrib_dict['weight'])
29
30 pos=nx.fruchterman_reingold_layout(G)
31
32
33 nx.draw(G,pos,edges=edges,edge_color=colors,width=weight,with_labels=True,font_size=8) #Se
34     dibuja el grafo
35 plt.savefig("Tarea2_06.eps")
36 plt.show() #Se dibuja en pantalla
```

Tarea2_06.py

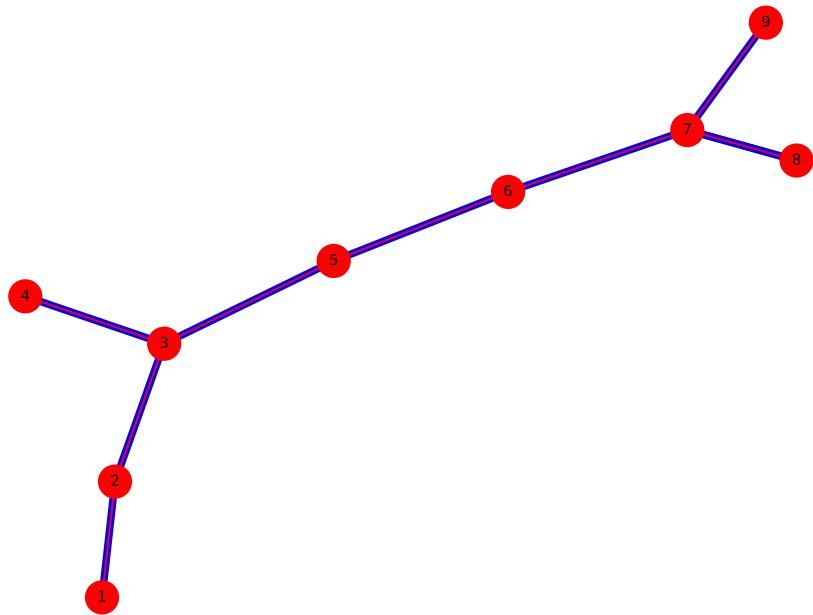


Figura 7: Kamada ~~Kawai~~ Layout

7. Spring Layout

En este diseño los nodos serían los cuerpos y los bordes los resortes que conectan a estos, los mismo son los que proporcionan fuerzas entre los nodos; estos se mueven de acuerdo con las fuerzas que se ejercen hasta alcanzar la mínima energía local [7].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.Graph() #Se crea un grafo vacio
5
6 G.add_node(1)
7 G.add_nodes_from([3,2,4,5])
8 G.add_edges_from([(1,2),(1,3),(1,5),(1,5)])
9 G.add_edges_from([(2,4)])
10 G.add_edges_from([(3,2),(3,5),(4,5)])
11 G.add_edges_from([(4,5)])
12
13 nx.draw(G, pos=nx.spring_layout(G), with_labels=True) #Se dibuja el grafo
14 plt.savefig("Tarea2_07.eps")
15 plt.show() #Se dibuja en pantalla
```

Tarea2_07.py

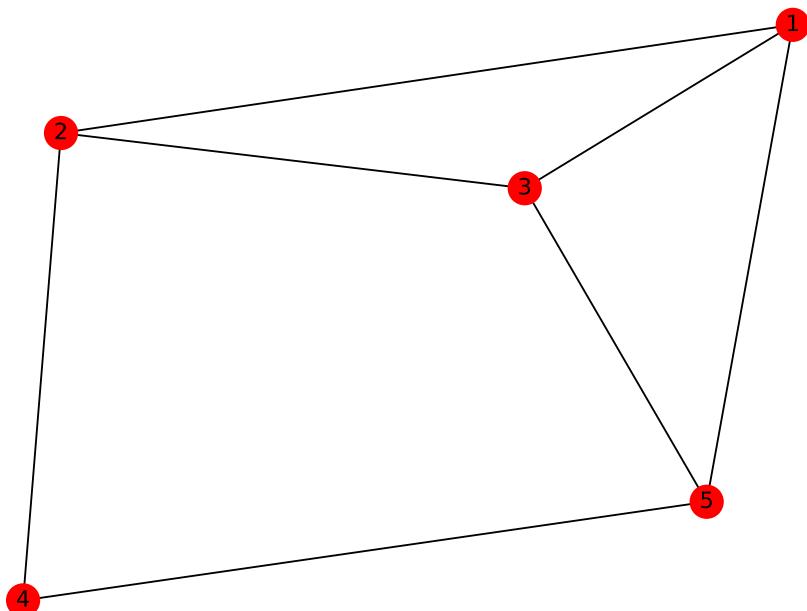


Figura 8: Spring Layout

```

1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G=nx.MultiDiGraph()
5
6 G.add_node(1)
7 G.add_nodes_from([2,3,4,5])
8 H=nx.Graph()
9 G.add_edges_from([(1,2),(2,1),(1,3),(4,5),(4,1)])
10
11 nx.draw(G, pos=nx.spring_layout(G), with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea2_11.eps")
13 plt.show() #Se dibuja en pantalla

```

Tarea2_11.py

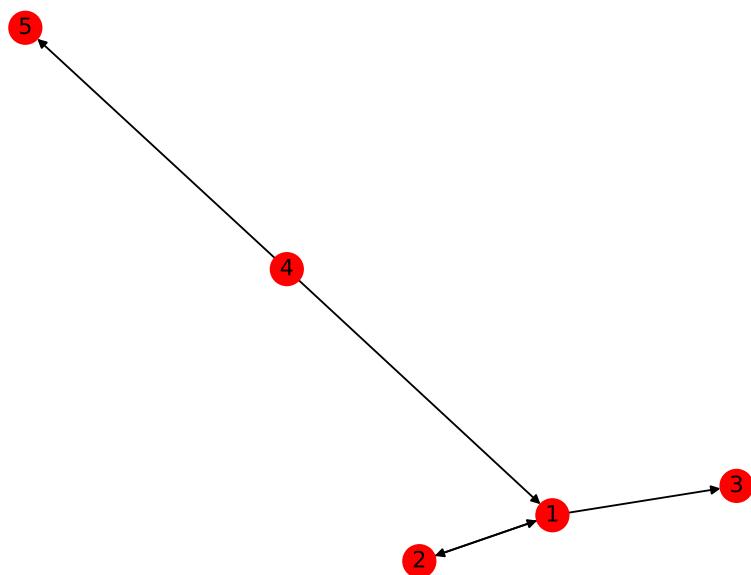


Figura 9: Spring Layout

8. Spectral Layout

Este diseño utiliza los vectores de matrices relacionadas con gráficas como las de adyacencia, tiene como ventaja la capacidad de calcular diseños óptimos y presentar un tiempo de cálculo muy rápido [3].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiDiGraph() #Se crea un grafo vacío
5
6 G.add_node(1) #Se crea el nodo raíz
7 G.add_nodes_from([2,3,4,5])
8 G.add_edge(2,1,color='red',weight=1)
9 G.add_edges_from([(1,2),(2,3),(3,4),(4,5),(5,2)],color='blue',weight=3)
10
11 edges=G.edges()
12
13 colors=[]
14 weight=[]
15
16 for (u,v,attrib_dict) in list(G.edges.data()):
17     colors.append(attrib_dict['color'])
18     weight.append(attrib_dict['weight'])
19
20 nx.draw(G, pos=nx.spectral_layout(G), edges=edges, edge_color=colors, width=weight, with_labels=True, font_size=8) #Se dibuja el grafo
21 plt.savefig("Tarea2_07.eps")
22 plt.show() #Se dibuja en pantalla
```

Tarea2_08.py

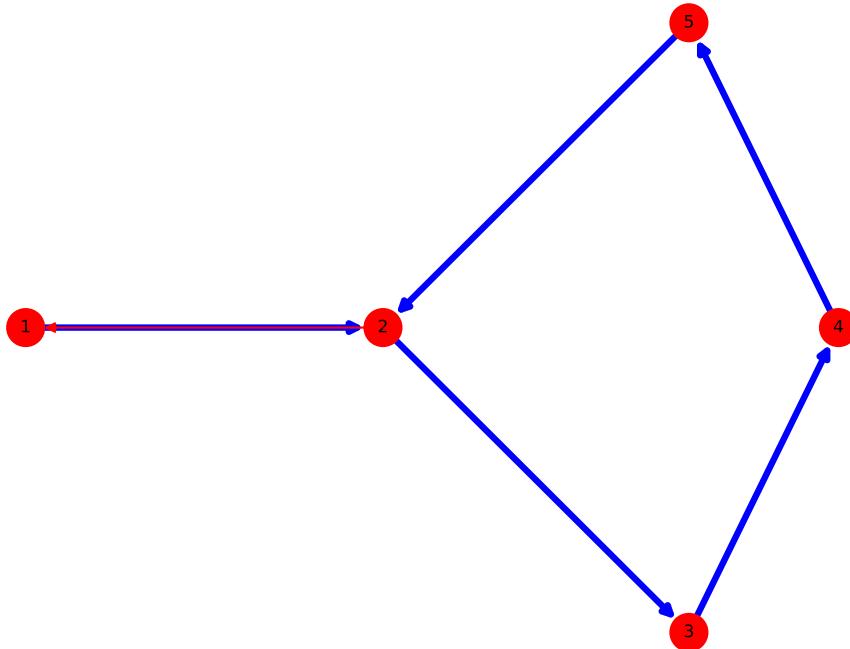


Figura 10: Spectral Layout

9. Bipartite Layout

Estos grafos representan las relaciones entre dos conjuntos y permiten la exploración paso a paso a gran escala. Los nodos de un mismo conjunto no se encuentran conectados entre sí [8].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.DiGraph() #Se crea un grafo vacio
5
6 G.add_nodes_from([('s','x'), bipartite=0)
7 G.add_nodes_from([('w','z','y','t'), bipartite=1)
8 G.add_edges_from([( ('s','w'), ('s','z'), ('s','t'), ('x','y'), ('z','x'))])
9
10
11 nx.draw(G, pos=nx.bipartite_layout(G,['s','x']), with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea2_06.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea2_09.py

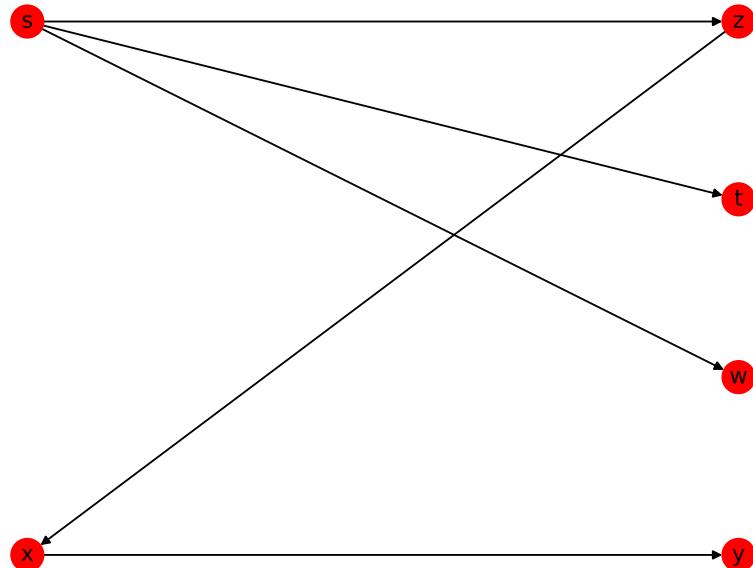


Figura 11: Bipartite Layout

```

1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.Graph() #Se crea un grafo vacio
5
6 G.add_nodes_from([1 , 2] , bipartite=0)
7 G.add_nodes_from([3 ,4 ,5] , bipartite=1)
8 G.add_edges_from([(1 ,3) ,(1 ,4) ,(2 ,4) ,(2 ,5)])
9
10 nx.draw(G, pos=nx.bipartite_layout(G,[1 ,2]) ,with_labels=True) #Se dibuja el grafo
11 plt.savefig("Tarea2_12.eps")
12 plt.show() #Se dibuja en pantalla

```

Tarea2_12.py

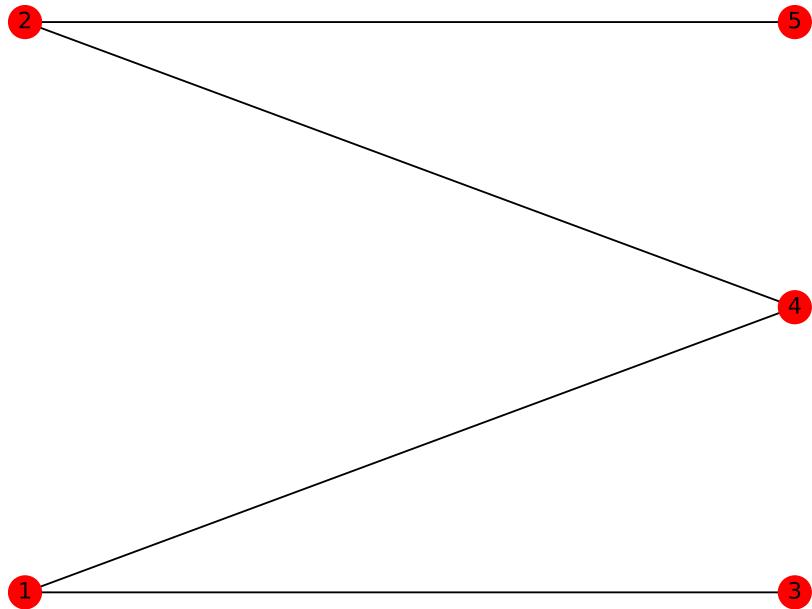


Figura 12: Bipartite Layout

Referencias

- [1] Un algoritmo para dibujar gráficos generales no dirigidos. *Cartas de procesamiento de información.*
- [2] Mehmet Esat Belviranlı. A circular layout algorithm for clustered graphs. Master's thesis, *bilkent university,* 2009.
- [3] Ulrik Brandes, Daniel Fleischer, and Thomas Puppe. Dynamic spectral layout with an application to small worlds. *Journal of Graph Algorithms and Applications*, 11(2):325–343, 2007.
- [4] Ken Cherven. *Mastering Gephi network visualization.* Packt Publishing Ltd, 2015. *ad duc*
- [5] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.
- [6] Emden R Gansner and Yehuda Koren. Improved circular layouts. In *International Symposium on Graph Drawing*, pages 386–398. Springer, 2006. *B Sc.*
- [7] Gerardo Huck. Posicionamiento automático de etiquetas en grafos. B.S. thesis, Facultad de Ciencias Exactas, Ingeniería y Agrimensura. Universidad Nacional, 2014. *@ phd-hcs*
- [8] Takao Ito, Kazuo Misue, and Jiro Tanaka. Drawing clustered bipartite graphs in multi-circular style. In *2010 14th International Conference Information Visualisation*, pages 23–28. IEEE, 2010.
- [9] Mathieu Jacomy, Tommaso Venturini, Sébastien Heymann, and Mathieu Bastian. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PLOS ONE*, 9(6):1–12, 06 2014. URL <https://doi.org/10.1371/journal.pone.0098679>.
- [10] NetworkX. Source code for networkx.drawing.layout, 2018. https://networkx.github.io/documentation/latest/_modules/networkx/drawing/layout.html, Last accessed on 2019-02-13.
- [11] Rogue Wave Software. Random layout, 2016. <https://docs.roguewave.com/visualization/views/6.1/views.html#page/Options%2Flayouts.51.19.html%23>, Last accessed on 2019-02-23.

Tarea 2

5272

1. Circular layout

Este algoritmo de diseño como su nombre lo indica puede ser útil cuando se desea estructurar la información con un patrón circular y se trata de disminuir los cruces entre los nodos [2]; es aplicado en redes sociales y administración de redes, con este se puede realizar estructuras de grupos y árbol dentro de una red [6].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.DiGraph()
5
6 G.add_node("X")
7 G.add_nodes_from([1, 2, 3, 4, 5, 6])
8
9 G.add_edges_from([( "X", 2), (2, 4), (4, 6), (6, 5), (5, 3), (3, 1), (1, "X")])
10
11 nx.draw(G, pos=nx.circular_layout(G), with_labels=True) # Dibuja el grafo G en forma circular
12 #plt.savefig("Tarea2_01.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea2_01.py

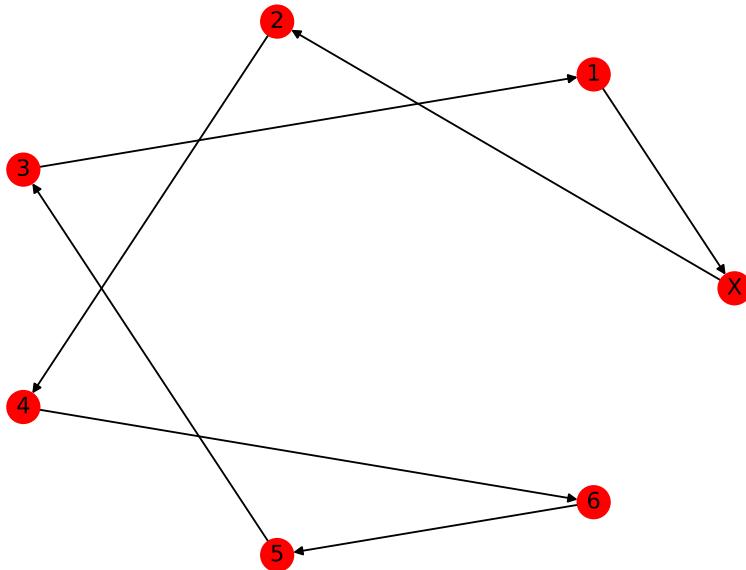


Figura 1: Circular layout

2. Kamada-Kawai layout

Este algoritmo es para grafos conectados y no dirigidos; se relaciona con un sistema de resorte dinámico donde la fuerza entre dos vértices es inversamente proporcional al cuadrado de la distancia entre estos, es decir, los vértices que tienen los resortes más fuertes se encuentran más cerca [1].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.Graph() #Se crea un grafo vacio
5
6 G.add_node(1) #Se crea el nodo raiz
7 G.add_nodes_from([2,3,4])
8
9 G.add_edges_from([(1,2),(2,3),(3,4),(2,2),(4,1),(2,4)])
10
11 color_map=[]
12 for node in G:
13     if (node==2):
14         color_map.append('blue')
15     else:
16         color_map.append('green')
17
18 nx.draw(G, pos=nx.kamada_kawai_layout(G), node_color=color_map, with_labels=True)
19 plt.savefig("Tarea2_02.eps")
20 plt.show() #Se dibuja en pantalla
```

Tarea2_02.py

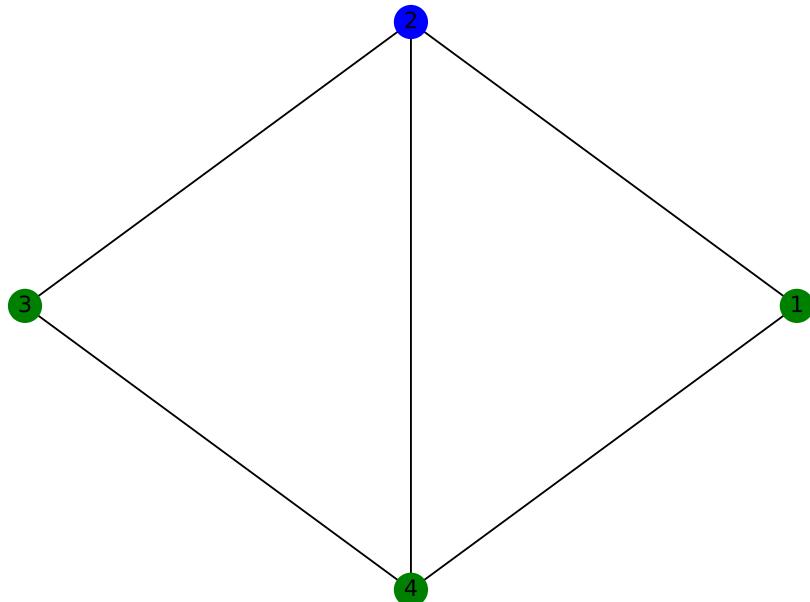


Figura 2: Kamada Kawai Layout

3. Random layout

Este diseño se utiliza para cualquier tipo de grafo conectado y no conectado, planos y no planos. Se caracteriza por darle un orden aleatorio a los nodos de un grafo y tiene como limitación que para asegurarse de que los nodos no se superpongan con los margenes de la región del diseño, este algoritmo calcula las coordenadas al azar dentro de una región con las dimensiones más pequeñas qu la región del diseño [11].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiGraph() #Se crea un grafo vacio
5
6 G.add_node("A") #Se crea el nodo raiz
7 G.add_nodes_from([ "B" , "C" , "D" , "E" ])
8
9 G.add_edge("B" , "C" , color='red' , weight=6)
10 G.add_edges_from([( "A" , "B" ),( "B" , "C" ),( "C" , "D" ),( "D" , "E" )] , color='blue' , weight=2)
11
12 edges = G.edges()
13
14 colors = []
15 weight = []
16
17 for (u,v,attrib_dict) in list(G.edges.data()):
18     colors.append(attrib_dict['color'])
19     weight.append(attrib_dict['weight'])
20
21 pos=nx.random_layout(G)
22
23 nx.draw(G,pos ,edges=edges ,edge_color=colors ,width=weight ,with_labels=True ,font_size=8)
24 plt.savefig("Tarea2_02.eps")
25 plt.show() #Se dibuja en pantalla
```

Tarea2_03.py

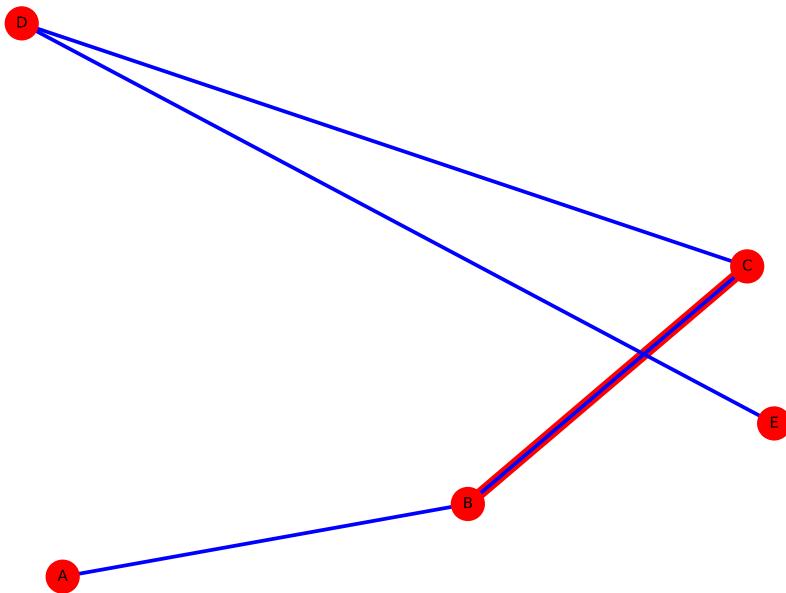


Figura 3: Random layout

```

1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiDiGraph() #Se crea un grafo vacio
5
6 G.add_node("A") #Se crea el nodo raiz
7 G.add_nodes_from(["C","B","R","G"])
8
9 G.add_edges_from([( "A" , "C" ),( "C" , "A" ),( "C" , "B" ),( "B" , "C" ),( "B" , "R" ),( "R" , "B" ),( "R" , "G" )])
10
11 nx.draw(G, pos=nx.random_layout(G), with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea2_10.eps")
13 plt.show() #Se dibuja en pantalla

```

Tarea2_10.py

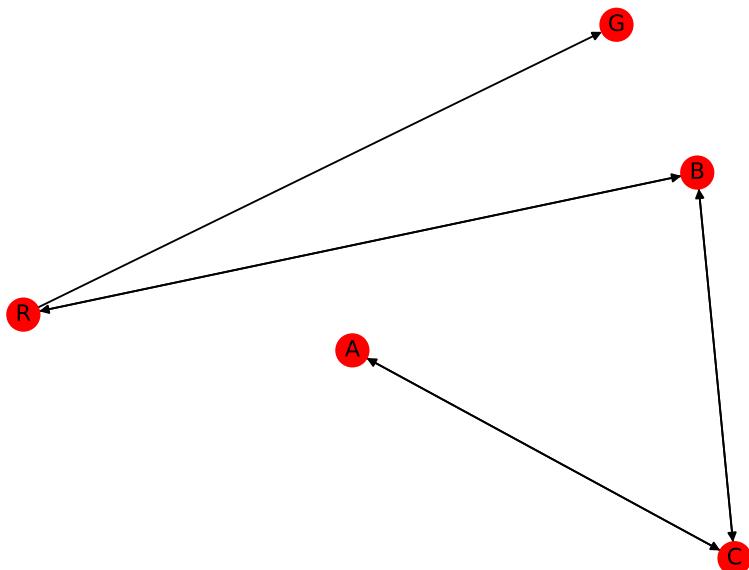


Figura 4: Random layout

4. ForceAtlas2 layout

En este diseño los nodos se rechazan entre sí, mientras que los bordes lo atraen como resortes, la posición de cada nodo depende de los otros nodos [9].

```
1 import networkx as nx
2 from fa2 import ForceAtlas2
3 import matplotlib.pyplot as plt
4
5 G = nx.Graph()
6
7 G.add_node(1)
8 G.add_nodes_from([2, 3, 4, 5, 6, 7, 8, 9, 10])
9
10 G.add_edges_from([(1, 10), (2, 10), (10, 3), (10, 4), (4, 5), (4, 6), (4, 7), (7, 4), (7, 8),
11 (7, 9)])
12 #Este fragmento de código fue tomado de https://github.com/bhargavchippada/forceatlas2/blob/
13 #master/examples/forceatlas2-layout.ipynb
14
15 forceatlas2 = ForceAtlas2(
16     outboundAttractionDistribution=True,
17     linLogMode=False,
18     adjustSizes=False,
19     edgeWeightInfluence=1.0,
20
21     jitterTolerance=1.0,
22     barnesHutOptimize=True,
23     barnesHutTheta=1.2,
24     multiThreaded=False,
25
26     scalingRatio=2.0,
27     strongGravityMode=False,
28     gravity=1.0,
29
30     verbose=True)
31
32 positions = forceatlas2.forceatlas2_networkx_layout(G, pos=None, iterations=2000)
33 nx.draw_networkx_nodes(G, positions, node_size=30, with_labels=False, node_color="green",
34 alpha=0.7)
35 nx.draw_networkx_edges(G, positions, edge_color="blue", alpha=0.06)
36 plt.axis('off')
37 plt.savefig("Tarea2_10.eps")
38 plt.show()
```

Tarea2_4.py

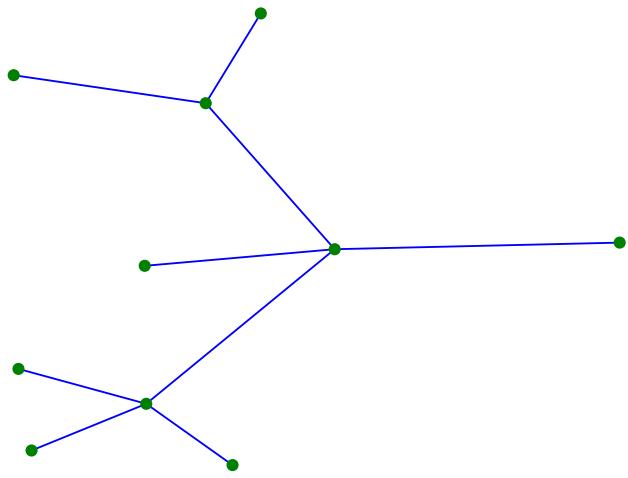


Figura 5: ForceAtlas2 layout

5. Shell layout

Este diseño posiciona los nodos en círculos concéntricos [10], basándose en la distancia entre el nodo central al resto de los nodos, por lo que los nodos se distribuyen de forma circular; este tiene como ventaja que su diseño hace que sea un grafo visualmente de fácil entender [4].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.Graph() #Se crea un grafo vacio
5
6 G.add_node(1)
7 G.add_nodes_from([2,3,4,5,6])
8 G.add_edges_from([(1,2),(1,3),(1,4),(1,5),(1,6)])
9 G.add_edges_from([(2,3)])
10 G.add_edges_from([(3,4)])
11 G.add_edges_from([(4,5)])
12 G.add_edges_from([(5,6)])
13 G.add_edges_from([(6,2)])
14
15 shells=[[1],[2,3,4,5,6]]
16
17 nx.draw(G, pos=nx.shell_layout(G, shells), with_labels=True) #Se dibuja el grafo
18 plt.savefig("Tarea2_05.eps")
19 plt.show() #Se dibuja en pantalla
```

Tarea2_05.py

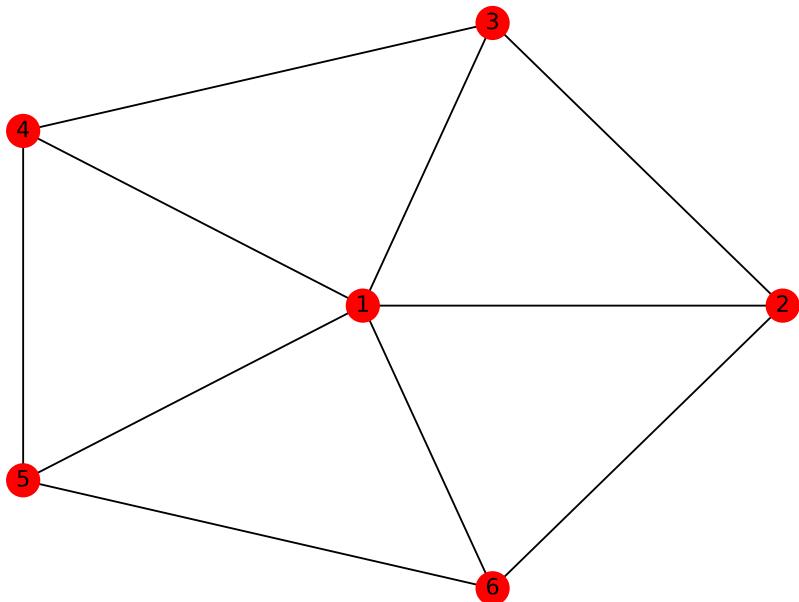


Figura 6: Shell layout

6. Kamada Kawai layout

Este algoritmo es dirigido por la fuerza entre dos nodos cualesquiera, donde los mismo se representan por anillos de acero y las aristas son los resortes entre ellos. Las fuerzas de atracción y de repulsión son análogas a la fuerza de resorte, donde la suma de las fuerzas determinan en que dirección se debe mover un vértice [5]; es utilizado en grafos no dirigidos muy grandes y garantiza que los nodos cercanos sean ubicados en la misma vecindad y los lejanos se ubican uno lejos de otros.

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiGraph() #Se crea un grafo vacio
5
6 G.add_node(1) #Se crea el nodo raiz
7
8 G.add_nodes_from([2,3,4,5,6,7,8,9])
9 G.add_edge(1,2,color='blue',weight=4)
10 G.add_edge(3,2,color='blue',weight=4)
11 G.add_edge(3,4,color='blue',weight=4)
12 G.add_edge(3,5,color='blue',weight=4)
13 G.add_edge(6,5,color='blue',weight=4)
14 G.add_edge(6,7,color='blue',weight=4)
15 G.add_edge(8,7,color='blue',weight=4)
16 G.add_edge(8,7,color='blue',weight=4)
17 G.add_edge(9,7,color='blue',weight=4)
18
19 G.add_edges_from([(2,1),(2,3),(4,3),(5,3),(5,6),(7,6),(7,8),(9,7)],color='red',weight=1)
20
21 edges=G.edges()
22
23 colors=[]
24 weight=[]
25
26 for (u,v,attrib_dict) in list(G.edges.data()):
27     colors.append(attrib_dict['color'])
28     weight.append(attrib_dict['weight'])
29
30 pos=nx.fruchterman_reingold_layout(G)
31
32
33 nx.draw(G,pos,edges=edges,edge_color=colors,width=weight,with_labels=True,font_size=8) #Se
34 dibuja el grafo
35 plt.savefig("Tarea2_06.eps")
plt.show() #Se dibuja en pantalla
```

Tarea2_06.py

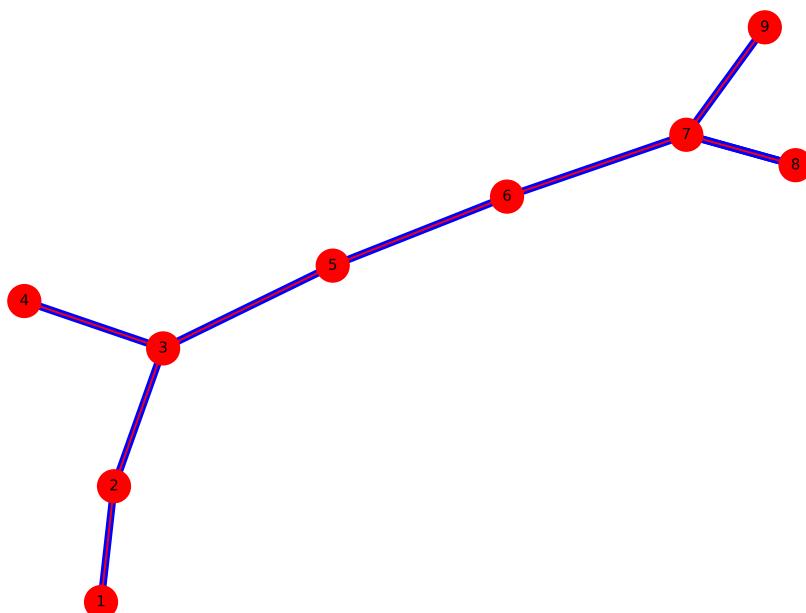


Figura 7: Kamada Kawai layout

7. Spring layout

En este diseño los nodos serían los cuerpos y los bordes los resortes que conectan a estos, los mismo son los que proporcionan fuerzas entre los nodos; estos se mueven de acuerdo con las fuerzas que se ejercen hasta alcanzar la mínima energía local [7].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.Graph() #Se crea un grafo vacio
5
6 G.add_node(1)
7 G.add_nodes_from([3,2,4,5])
8 G.add_edges_from([(1,2),(1,3),(1,5),(1,5)])
9 G.add_edges_from([(2,4)])
10 G.add_edges_from([(3,2),(3,5),(4,5)])
11 G.add_edges_from([(4,5)])
12
13 nx.draw(G, pos=nx.spring_layout(G), with_labels=True) #Se dibuja el grafo
14 plt.savefig("Tarea2_07.eps")
15 plt.show() #Se dibuja en pantalla
```

Tarea2_07.py

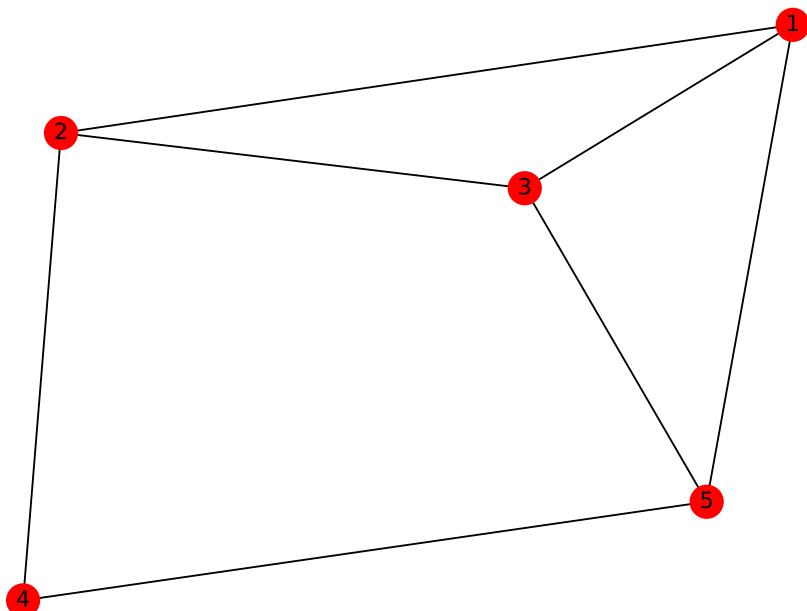


Figura 8: Spring layout

```

1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G=nx.MultiDiGraph()
5
6 G.add_node(1)
7 G.add_nodes_from([2,3,4,5])
8 H=nx.Graph()
9 G.add_edges_from([(1,2),(2,1),(1,3),(4,5),(4,1)])
10
11 nx.draw(G, pos=nx.spring_layout(G), with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea2_11.eps")
13 plt.show() #Se dibuja en pantalla

```

Tarea2_11.py

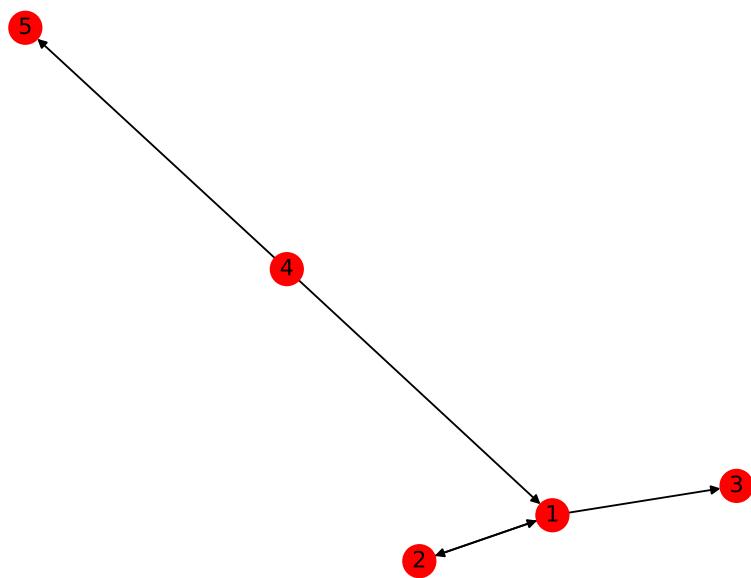


Figura 9: Spring Layout

8. Spectral layout

Este diseño utiliza los vectores de matrices relacionadas con gráficas como las de adyacencia, tiene como ventaja la capacidad de calcular diseños óptimos y presentar un tiempo de calculo muy rápido [3].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.MultiDiGraph() #Se crea un grafo vacio
5
6 G.add_node(1) #Se crea el nodo raiz
7 G.add_nodes_from([2,3,4,5])
8 G.add_edge(2,1,color='red',weight=1)
9 G.add_edges_from([(1,2),(2,3),(3,4),(4,5),(5,2)],color='blue',weight=3)
10
11 edges=G.edges()
12
13 colors=[]
14 weight=[]
15
16 for (u,v,attrib_dict) in list(G.edges.data()):
17     colors.append(attrib_dict['color'])
18     weight.append(attrib_dict['weight'])
19
20 nx.draw(G, pos=nx.spectral_layout(G), edges=edges, edge_color=colors, width=weight, with_labels=True, font_size=8) #Se dibuja el grafo
21 plt.savefig("Tarea2_07.eps")
22 plt.show() #Se dibuja en pantalla
```

Tarea2_08.py

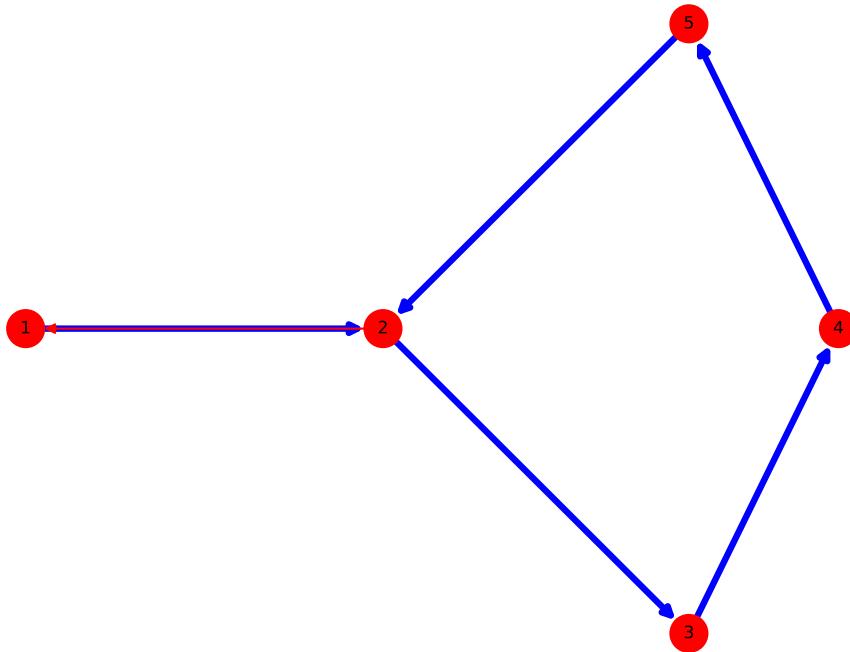


Figura 10: Spectral layout

9. Bipartite layout

Estos grafos representan las relaciones entre dos conjuntos y permiten la exploración paso a paso a gran escala. Los nodos de un mismo conjunto no se encuentran conectados entre sí [8].

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.DiGraph() #Se crea un grafo vacio
5
6 G.add_nodes_from([('s','x'),], bipartite=0)
7 G.add_nodes_from([('w','z','y','t'),], bipartite=1)
8 G.add_edges_from([( ('s','w'), ('s','z') ),( ('s','z'), ('s','t') ),( ('x','y'), ('z','x') )])
9
10
11 nx.draw(G, pos=nx.bipartite_layout(G,['s','x']), with_labels=True) #Se dibuja el grafo
12 plt.savefig("Tarea2_06.eps")
13 plt.show() #Se dibuja en pantalla
```

Tarea2_09.py

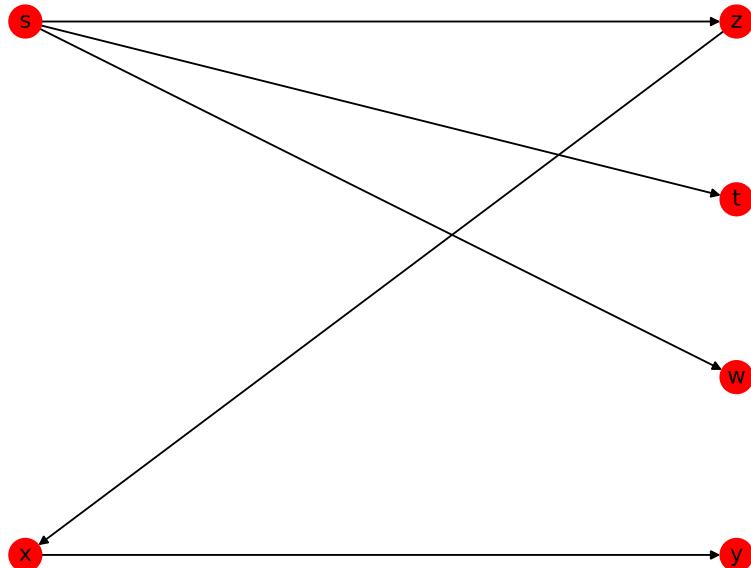


Figura 11: Bipartite layout

```

1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 G = nx.Graph() #Se crea un grafo vacio
5
6 G.add_nodes_from([1 , 2] , bipartite=0)
7 G.add_nodes_from([3 ,4 ,5] , bipartite=1)
8 G.add_edges_from([(1 ,3) ,(1 ,4) ,(2 ,4) ,(2 ,5)])
9
10 nx.draw(G, pos=nx.bipartite_layout(G,[1 ,2]) ,with_labels=True) #Se dibuja el grafo
11 plt.savefig("Tarea2_12.eps")
12 plt.show() #Se dibuja en pantalla

```

Tarea2_12.py

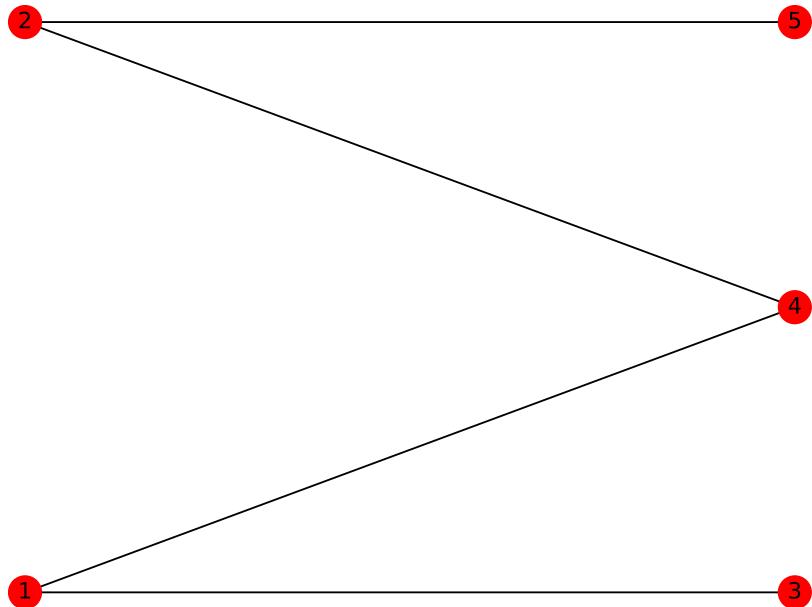


Figura 12: Bipartite layout

Referencias

- [1] Un algoritmo para dibujar gráficos generales no dirigidos. *Cartas de procesamiento de información*.
- [2] Mehmet Esat Belviranlı. A circular layout algorithm for clustered graphs. Master's thesis, bilkent university, 2009.
- [3] Ulrik Brandes, Daniel Fleischer, and Thomas Puppe. Dynamic spectral layout with an application to small worlds. *Journal of Graph Algorithms and Applications*, 11(2):325–343, 2007.
- [4] Ken Cherven. *Mastering Gephi network visualization*. Packt Publishing Ltd, 2015.
- [5] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.
- [6] Emden R Gansner and Yehuda Koren. Improved circular layouts. In *International Symposium on Graph Drawing*, pages 386–398. Springer, 2006.
- [7] Gerardo Huck. Posicionamiento automático de etiquetas en grafos. B.S. thesis, Facultad de Ciencias Exactas, Ingeniería y Agrimensura. Universidad Nacional, 2014.
- [8] Takao Ito, Kazuo Misue, and Jiro Tanaka. Drawing clustered bipartite graphs in multi-circular style. In *2010 14th International Conference Information Visualisation*, pages 23–28. IEEE, 2010.
- [9] Mathieu Jacomy, Tommaso Venturini, Sébastien Heymann, and Mathieu Bastian. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PLOS ONE*, 9(6):1–12, 06 2014. URL <https://doi.org/10.1371/journal.pone.0098679>.
- [10] NetworkX. Source code for networkx.drawing.layout, 2018. https://networkx.github.io/documentation/latest/_modules/networkx/drawing/layout.html, Last accessed on 2019-02-13.
- [11] Rogue Wave Software. Random layout, 2016. <https://docs.roguewave.com/visualization/views/6.1/views.html#page/Options%2Flayouts.51.19.html%23>, Last accessed on 2019-02-23.

Correcciones realizadas:

En esta tarea se corrigieron los acentos, a sí como los espacios entre palabras y mejora en la redacción. Se le incluyeron las referencias a las figuras correspondientes; se modificó el estilo de letra en palabras señales. Se modificaron partes del código que tenía espacios subutilizados, así como se eliminaron acentos que generaban errores de lectura.

8

Tarea 3

5272

$\langle u, v \rangle$

1. Algoritmos para grafos

1.1. Todos los caminos más cortos

Se utiliza en grafos ponderados dirigidos conectados $G(V, E)$, donde para cada borde $\langle u, v \rangle \in E$ se le asocia un peso w para el problema de todos los pares de rutas más cortas. Tiene aplicación en los problemas del servicio urbano, como la ubicación de las instalaciones urbanas o la distribución o entrega de bienes [5].

$G = (V, E)$

Wangle range

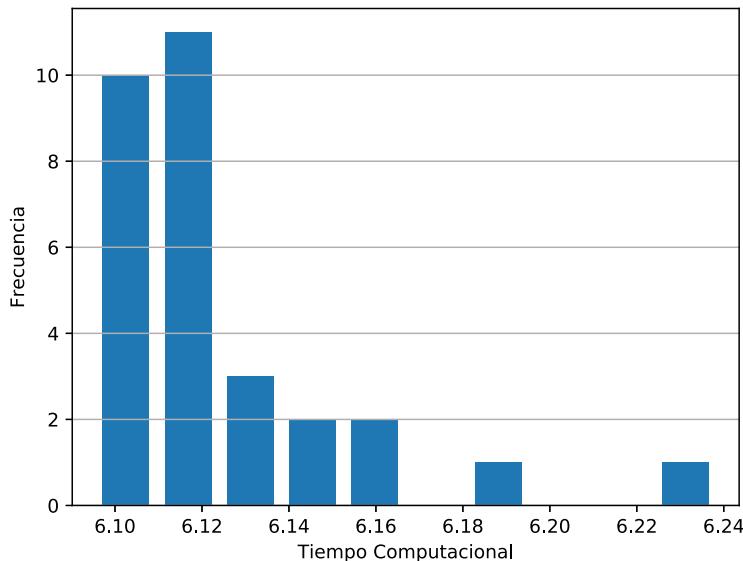


Figura 1

1.2. Centralidad intermedia

Este algoritmo calcula la ruta más corta ponderada entre cada par de nodos en un grafo conectado, en este se emplea la búsqueda por amplitud. Se utiliza para encontrar nodos que sirven de puente de una parte de un grafo a otra. Se aplica en un proceso de entrega de paquetes o red de telecomunicaciones [4].

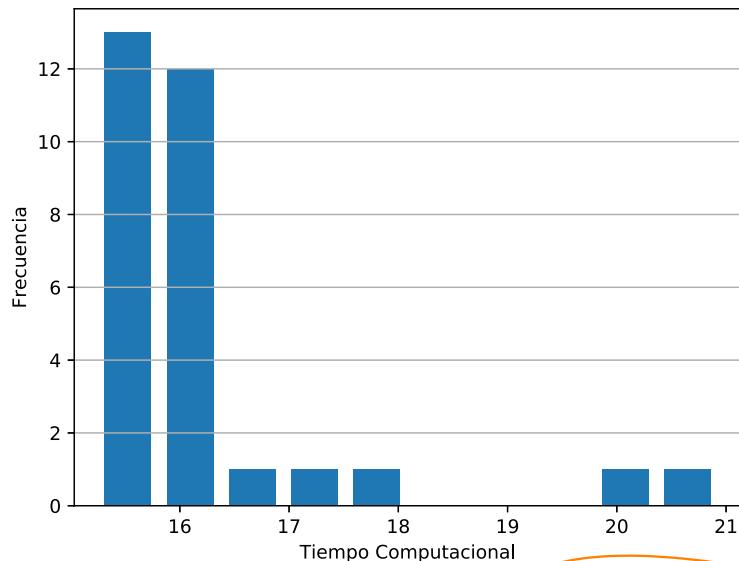


Figura 2



1.3. Búsqueda topológica

Este algoritmo crea un ordenamiento lineal de los vértices, de modo que si aparece el borde (u, v) en el gráfico, v aparece antes que u en el ordenamiento. El grafo debe de ser un gráfico acíclico dirigido [3].

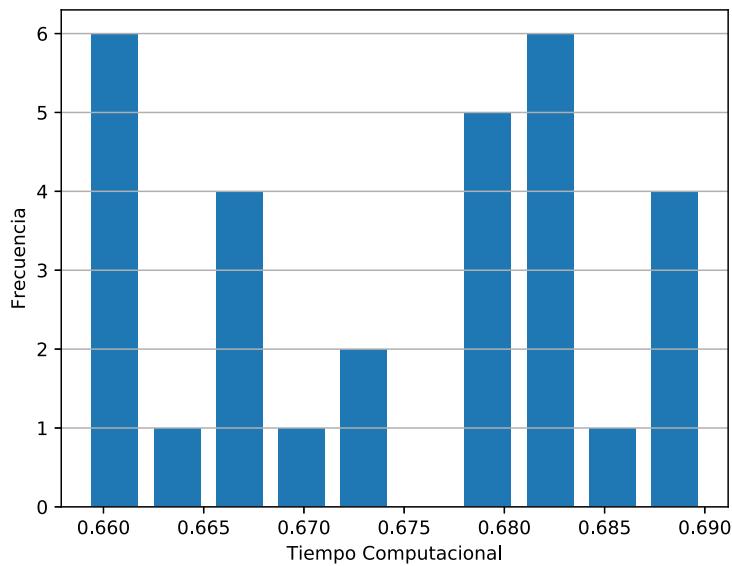


Figura 3



1.4. Camarilla máxima

Este algoritmo calcula la camarilla máxima del grafo G , es decir, el subgrafo completo del tamaño máximo. El parámetro ind es para la elección del método, si el parámetro es 0 el método es un algoritmo basado en la programación cuadrática 0 – 1. El tamaño de la salida es el número de nodos de la camarilla encontrados por el algoritmo [2].

\text{tex++}

A curved orange arrow points from the word 'ind' in the text above to the backslash and 'tex' in the text below.

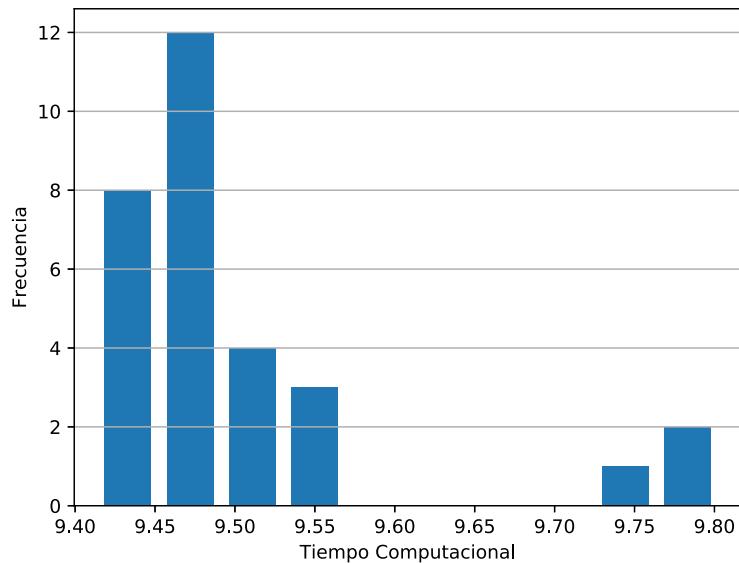


Figura 4



1.5. Árbol ~~dfs~~ ~~DFS~~

Este algoritmo de búsqueda de profundidad es una de las técnicas de desplazamiento gráfica más utilizadas; implica atravesar el grafo y construir un árbol de expansión (o bosque) arraigado [1].

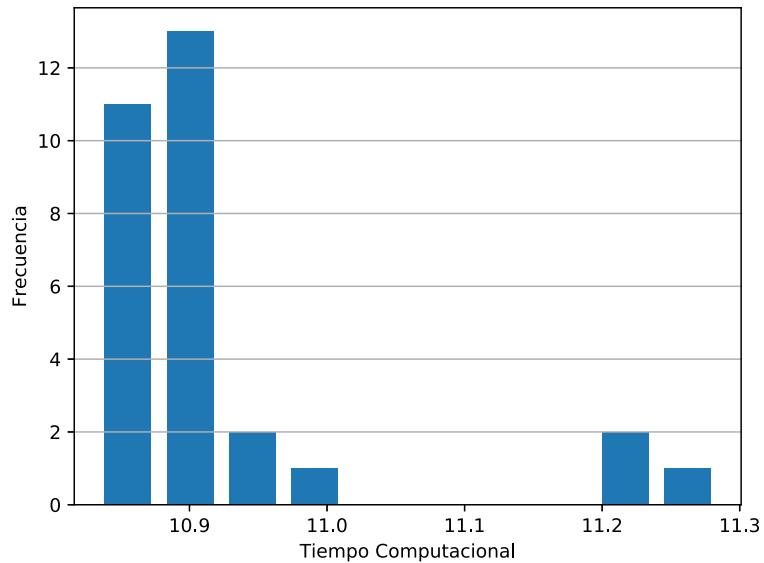


Figura 5



2. Análisis de resultados

Luego de correr los 5 algoritmos para grafos diferentes con un tiempo computacional mayor a 5 segundos con un número de réplicas igual a 70 000 y realizar los histogramas correspondientes a cada uno, se demuestra que los datos no tienen una distribución normal.

Se realizan dos gráficos de dispersión Tiempo vs. Cantidad de nodos (Figura 6) y Tiempo vs. Cantidad de arista (Figura 7) donde se observa que el algoritmo más lento es el de Centralidad intermedia y el más rápido es Búsqueda topológica.

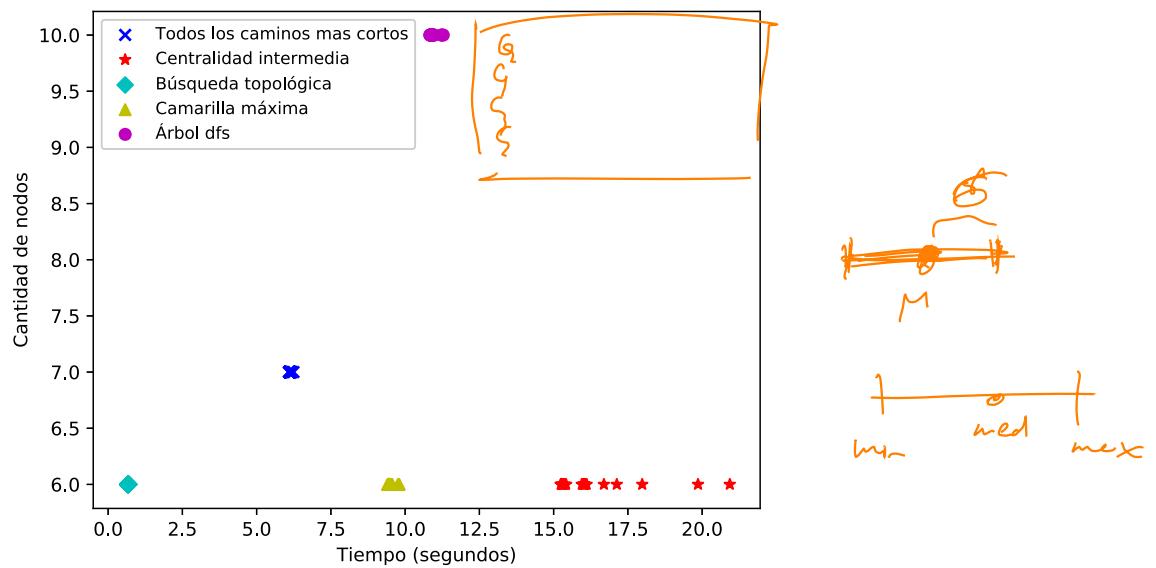


Figura 6: Tiempo vs Cantidad de Nodos

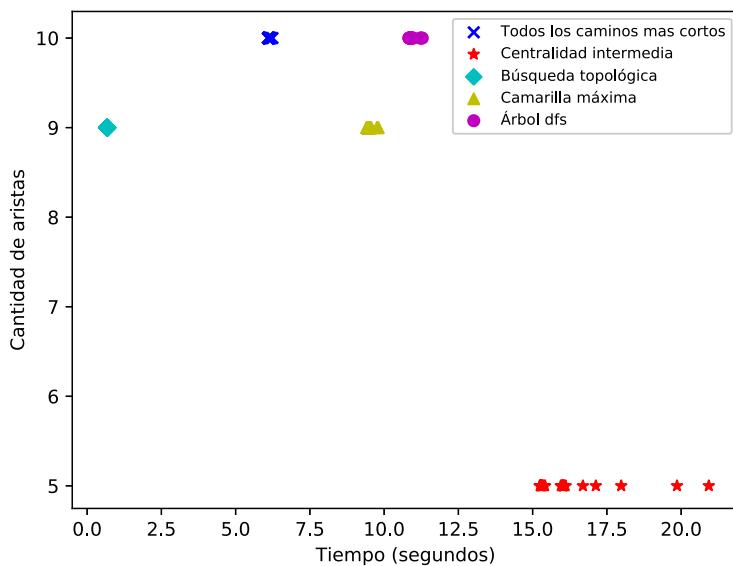


Figura 7: Tiempo vs Cantidad de Nodos

3. Fragmento de Código

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3 import time
4 import numpy as np
5 import pandas as pd
6 from scipy import stats
7 #####
8 hist, bin_edges = np.histogram(list_5, density=True)
9 first_edge, last_edge = np.min(list_5), np.max(list_5)
10 #####
11 n_equal_bins = 10
12 bin_edges = np.linspace(start=first_edge, stop=last_edge, num=n_equal_bins + 1, endpoint=True)
13 #####
14 plt.hist(list_5, bins=bin_edges, rwidth=0.75)
15 plt.xlabel('Tiempo Computacional')
16 plt.ylabel('Frecuencia')
17 plt.grid(axis='y', alpha=0.75)
18 plt.savefig("Tarea3_05.eps")
19 plt.show(1)
20 #####
21 normality_test = stats.shapiro(list_5)
22 print(normality_test)
23 #####
24 colors = [ 'b' , 'c' , 'y' , 'm' , 'r' ]
25 #####
26 lo = plt.scatter(list_1 , nodos_1 , marker='x' , color=colors[0])
27 ll = plt.scatter(list_2 , nodos_2 , marker='*' , color=colors[4])
28 l = plt.scatter(list_3 , nodos_3 , marker='D' , color=colors[1])
29 a = plt.scatter(list_4 , nodos_4 , marker='^' , color=colors[2])
30 h = plt.scatter(list_5 , nodos_5 , marker='o' , color=colors[3])
31 #####
32 plt.legend((lo, ll, l, a, h),
33             ('All-Pairs shortest paths', 'Betweenness Centrality', 'Topological Sort', 'Max
34             Clique', 'DFS Tree'),
35             scatterpoints=1,
36             loc='upper left',
37             ncol=1,
38             fontsize=9)
39 #####
40 plt.xlabel('Tiempo (segundos)')
41 plt.ylabel('Cantidad de nodos')
42 plt.savefig("Tarea3_06.eps")
43 plt.show(1)
```

Tarea3.py

Referencias

- [1] Surender Baswana and Shahbaz Khan. Incremental algorithm for maintaining a dfs tree for undirected graphs. *Algorithmica*, 79(2):466–483, 2017.
- [2] Patric RJ Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1-3):197–207, 2002.
- [3] David J Pearce and Paul HJ Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *Journal of Experimental Algorithms (JEA)*, 11:1–7, 2007.
- [4] Douglas R White and Stephen P Borgatti. Betweenness centrality measures for directed graphs. *Social networks*, 16(4):335–346, 1994.
- [5] Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM (JACM)*, 49(3):289–317, 2002.

Tarea 3

5272

1. Todos los caminos más cortos

Se utiliza en grafos ponderados dirigidos conectados $G(V, E)$, donde para cada borde $\langle u, v \rangle \in E$ se le asocia un peso w para el problema de todos los pares de rutas más cortas. Tiene aplicación en los problemas del servicio urbano, como la ubicación de las instalaciones urbanas o la distribución o entrega de bienes [5].

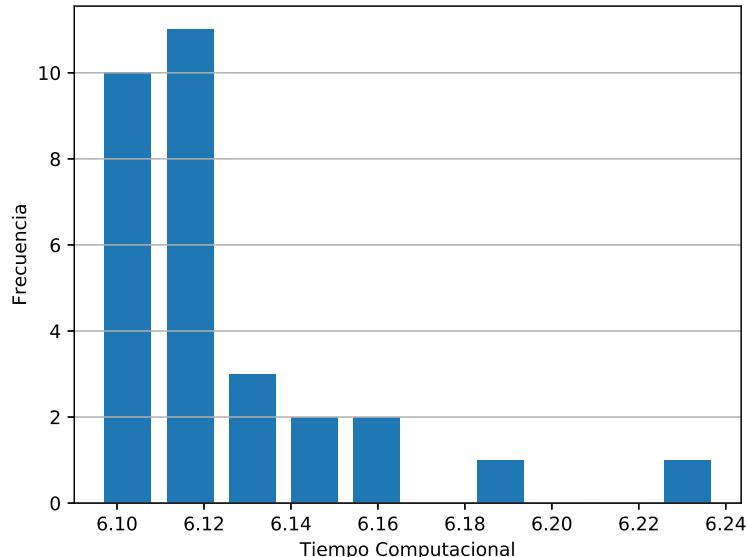


Figura 1: caminos más cortos

2. Centralidad intermedia

Este algoritmo calcula la ruta más corta ponderada entre cada par de nodos en un grafo conectado, en este se emplea la búsqueda por amplitud. Se utiliza para encontrar nodos que sirven de puente de una parte de un grafo a otra. Se aplica en un proceso de entrega de paquetes o red de telecomunicaciones [4].

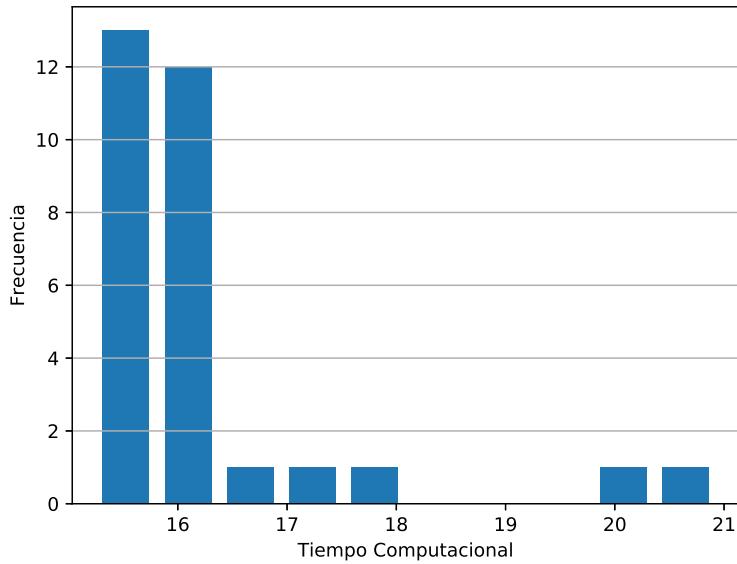


Figura 2: Centralidad intermedia

3. Búsqueda topológica

Este algoritmo crea un ordenamiento lineal de los vértices, de modo que si aparece el borde (u, v) en el gráfico, v aparece antes que u en el ordenamiento. El grafo debe de ser un gráfico acíclico dirigido [3].

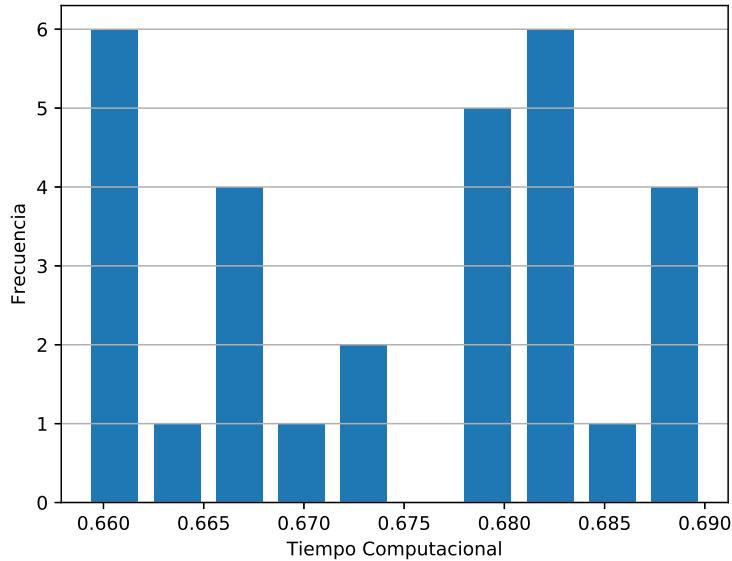


Figura 3: Búsqueda topológica

4. Camarilla máxima

Este algoritmo calcula la camarilla máxima del grafo G , es decir, el subgrafo completo del tamaño máximo. El parámetro ind es para la elección del método, si el parámetro es 0 el método es un algoritmo basado en la programación cuadrática 0 – 1. El tamaño de la salida es el número de nodos de la camarilla encontrados por el algoritmo [2].

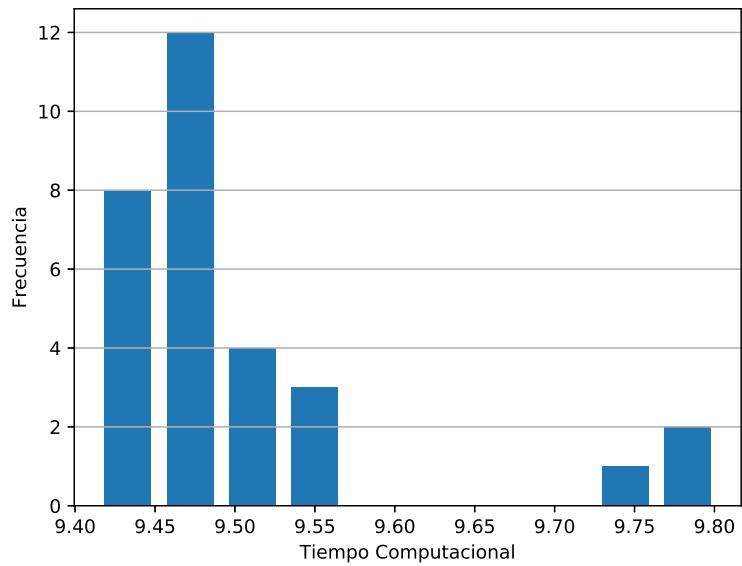


Figura 4: Camarilla máxima

5. Árbol DFS

Este algoritmo de búsqueda de profundidad es una de las técnicas de desplazamiento gráfica más utilizadas; implica atravesar el grafo y construir un árbol de expansión (o bosque) arraigado [1].

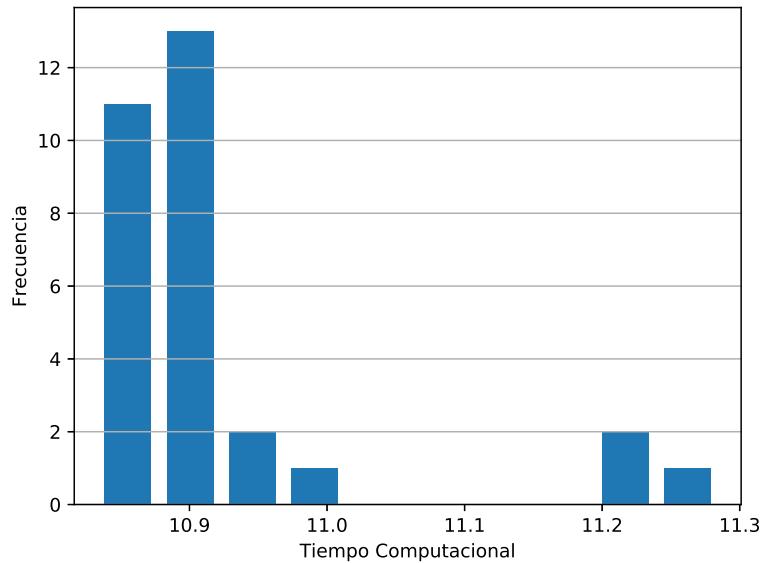


Figura 5: Árbol DFS

6. Análisis de resultados

Luego de correr los cinco algoritmos para grafos diferentes con un tiempo computacional mayor a cinco segundos con un número de replicas igual a 70 000 y realizar los histogramas correspondientes a cada uno, se demuestra que los datos no tienen una distribución normal.

Se realizan dos gráficos de dispersión Tiempo vs. Cantidad de nodos [Figura 6] y Tiempo vs. Cantidad de arista [Figura 7] donde se observa que el algoritmo más lento es el de Centralidad intermedia y el más rápido es Búsqueda topológica.

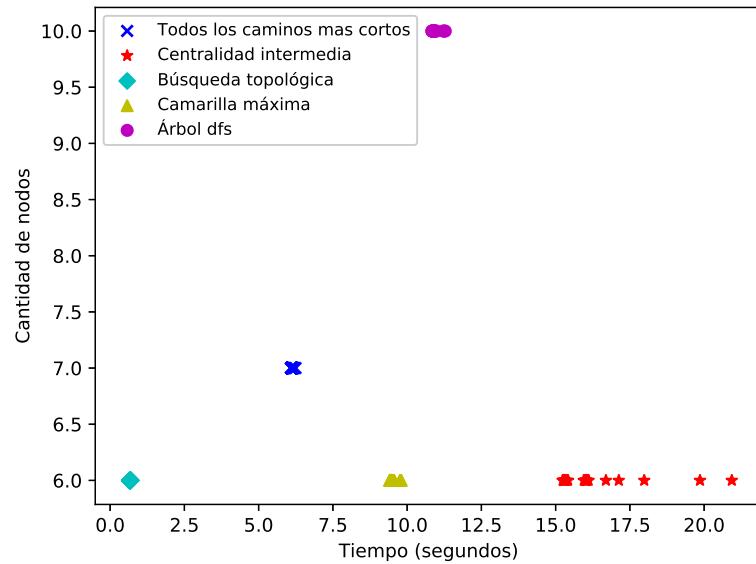


Figura 6: Tiempo vs Cantidad de Nodos

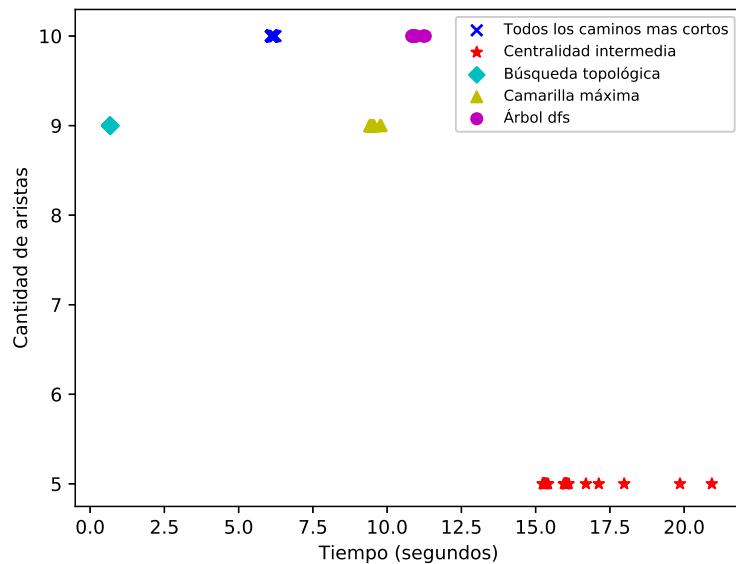


Figura 7: Tiempo vs Cantidad de Nodos

7. Fragmento de Código

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3 import time
4 import numpy as np
5 import pandas as pd
6 from scipy import stats
7 #####
8 hist, bin_edges = np.histogram(list_5, density=True)
9 first_edge, last_edge = np.min(list_5), np.max(list_5)
10 #####
11 n_equal_bins = 10
12 bin_edges = np.linspace(start=first_edge, stop=last_edge, num=n_equal_bins + 1, endpoint=True)
13 #####
14 plt.hist(list_5, bins=bin_edges, rwidth=0.75)
15 plt.xlabel('Tiempo Computacional')
16 plt.ylabel('Frecuencia')
17 plt.grid(axis='y', alpha=0.75)
18 plt.savefig("Tarea3_05.eps")
19 plt.show(1)
20 #####
21 normality_test = stats.shapiro(list_5)
22 print(normality_test)
23 #####
24 colors = [ 'b' , 'c' , 'y' , 'm' , 'r' ]
25 #####
26 lo = plt.scatter(list_1 , nodos_1 , marker='x' , color=colors[0])
27 ll = plt.scatter(list_2 , nodos_2 , marker='*' , color=colors[4])
28 l = plt.scatter(list_3 , nodos_3 , marker='D' , color=colors[1])
29 a = plt.scatter(list_4 , nodos_4 , marker='^' , color=colors[2])
30 h = plt.scatter(list_5 , nodos_5 , marker='o' , color=colors[3])
31 #####
32 plt.legend((lo, ll, l, a, h),
33             ('All-Pairs shortest paths', 'Betweenness Centrality', 'Topological Sort', 'Max
34             Clique', 'DFS Tree'),
35             scatterpoints=1,
36             loc='upper left',
37             ncol=1,
38             fontsize=9)
39 #####
40 plt.xlabel('Tiempo (segundos)')
41 plt.ylabel('Cantidad de nodos')
42 plt.savefig("Tarea3_06.eps")
43 plt.show(1)
```

Tarea3.py

Referencias

- [1] Surender Baswana and Shahbaz Khan. Incremental algorithm for maintaining a dfs tree for undirected graphs. *Algorithmica*, 79(2):466–483, 2017.
- [2] Patric RJ Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1-3):197–207, 2002.
- [3] David J Pearce and Paul HJ Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *Journal of Experimental Algorithms (JEA)*, 11:1–7, 2007.
- [4] Douglas R White and Stephen P Borgatti. Betweenness centrality measures for directed graphs. *Social networks*, 16(4):335–346, 1994.
- [5] Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM (JACM)*, 49(3):289–317, 2002.

Correcciones realizadas:

En esta tarea se corrigieron los acentos, así como los espacios entre palabras y mejora en la redacción. Se le incluyeron las referencias a las figuras correspondientes; se modificó el estilo de letra en palabras señadas. Se modificaron partes del código que tenía espacios subutilizados, así como se eliminaron acentos que generaban errores de lectura.



Tarea 4

5272

1. Introducción

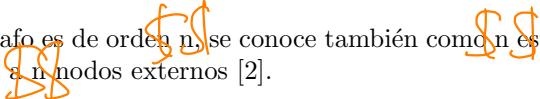
En el presente trabajo se utilizaron tres diferentes tipos de generadores de grafos en este caso los seleccionados fueron: Grafo Estrella, Grafo Paleta, Grafo Rueda; para cada uno se seleccionó una base logarítmica 2 para crear cuatro grafos de orden 4, 8, 16, 32. Los pesos de cada arco se distribuyen normalmente con media 8 y una desviación estándar de 0.3. Se utilizaron además los algoritmos de flujo máximo: Algoritmo Dinitz, Empuje de Preflujo y Ruta de Aumento más Corta.

2. Generadores de Grafos

Los generadores de grafos utilizados se encuentran a continuación:

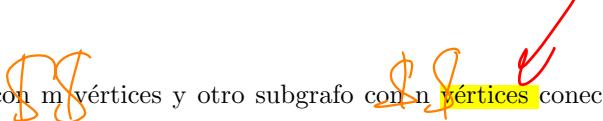
Grafo Estrella

Este grafo es de orden n , se conoce también como estrella. Devuelve un grado con $n+1$ como nodo central, conectado a n nodos externos [2].



Grafo Paleta

Este grafo consiste en un subgrafo completo con m vértices y otro subgrafo con n vértices conectados por medio de un puente [1].



Grafo Rueda

Este grafo también es llamado n ruedas, está formado por la conexión de un solo vértice universal a todos los vértices de un ciclo es decir, contiene un ciclo de orden $n - 1$ para el cual cada vértice del ciclo está conectado a otro vértice [6].

3. Algoritmos de Flujo Máximo

Los algoritmos de flujo máximo utilizados se encuentran a continuación:

Algoritmo Dinitz

Es uno de los algoritmos más rápidos y más fácil de implementar, también se le conoce como algoritmo Dinic. Este algoritmo polinomial calcula el flujo máximo, donde regresa una red residual resultante [3].

Empuje de Preflujo

Este tipo de algoritmo maneja un preflujo, es decir, no se satisface necesariamente las condiciones de conservación de flujo, por lo que es posible que un vértice reciba más flujo que el que distribuye. Los algoritmos de preflujo se pueden mejorar significativamente al incorporar condiciones para decidir qué nodos elegir, las cuales tienden a ser heurísticas [5].

Ruta de Aumento más Corta

Encuentra un flujo máximo utilizando el algoritmo de ruta de aumento más corta, regresa una red residual resultante de calcular el flujo máximo [4].

4. Resultados Computacionales

Se realiza una evaluación de los datos utilizando herramientas estadísticas como los Boxplot que se describen a continuación. El primer análisis realizado es del tiempo contra los algoritmos de flujo máximo (*Figura1*) donde se puede decir que los algoritmos Dinitz, Preflow Push y Shortest tiene una media de 0.005532, 0.009594 y 0.006333 respectivamente y una desviación de 0.005054, 0.007593 y 0.004624 para cada uno; el segundo Boxplot (*Figura2*) es tiempo contra densidad donde la media está en un rango entre 0,06 – 1,17 con tiempos entre 0,008680 – 0,003434; para el tercer Boxplot (*Figura3*) la media tiene un comportamiento para cada algoritmo de la siguiente forma: Dinitz: 0.009041, Preflow Push: 0.004764 y Shortest: 0.009435 y una desviación Dinitz: 0.008347, Preflow Push: 0.003241 y Shortest: 0.006466; por último se realizó el Boxplot de tiempo contra orden (*Figura4*) quedando de la siguiente forma: para los algoritmos de orden 4 la media y la desviación tienen un valor de 0.002647 y 0.001058 respectivamente, para los algoritmos de orden 8 un valor de 0.004634 para la media y 0.001991 para la desviación, para orden 16 valores de 0.008008 y 0.003711 respectivamente y por último para orden 32 una media de 0.015339 y una desviación de 0.008589.

Figura~\ref{...}

rem

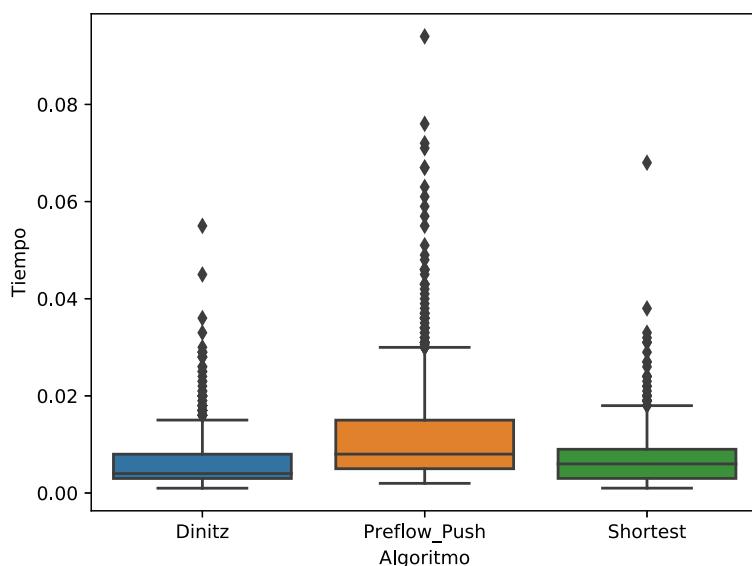


Figura 1: Boxplot Algoritmo

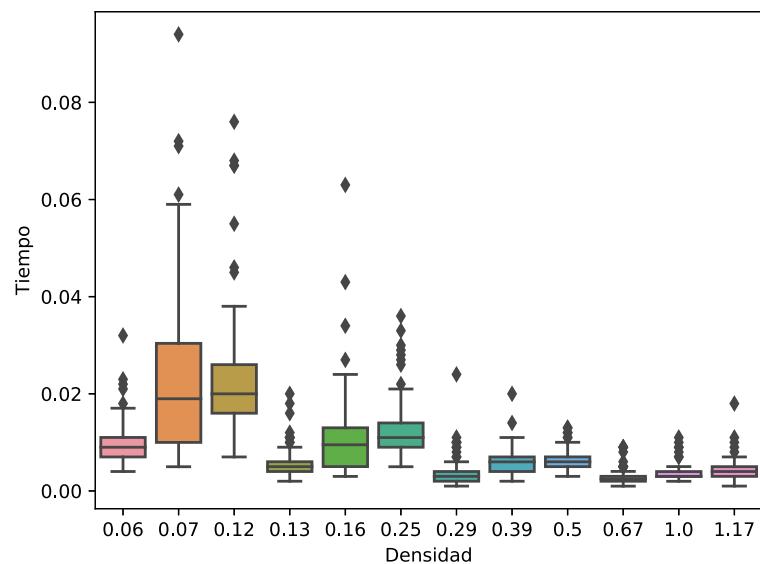


Figura 2: Boxplot Densidad

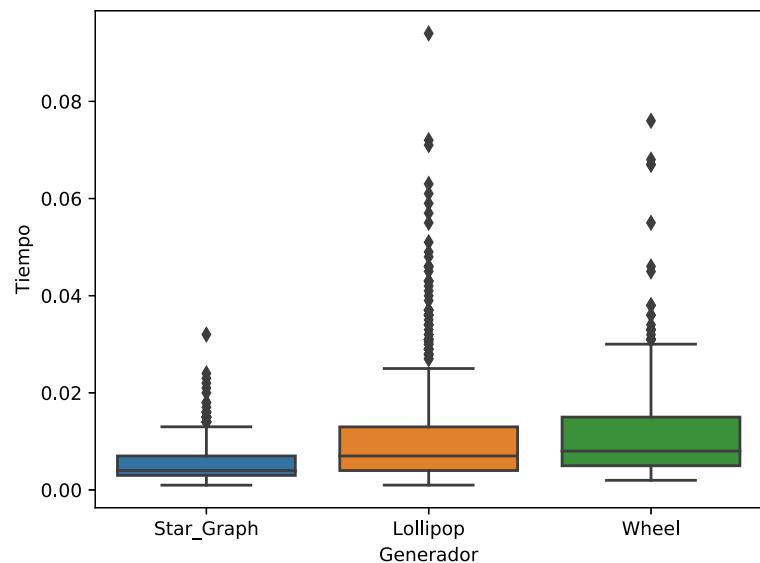


Figura 3: Boxplot Generador

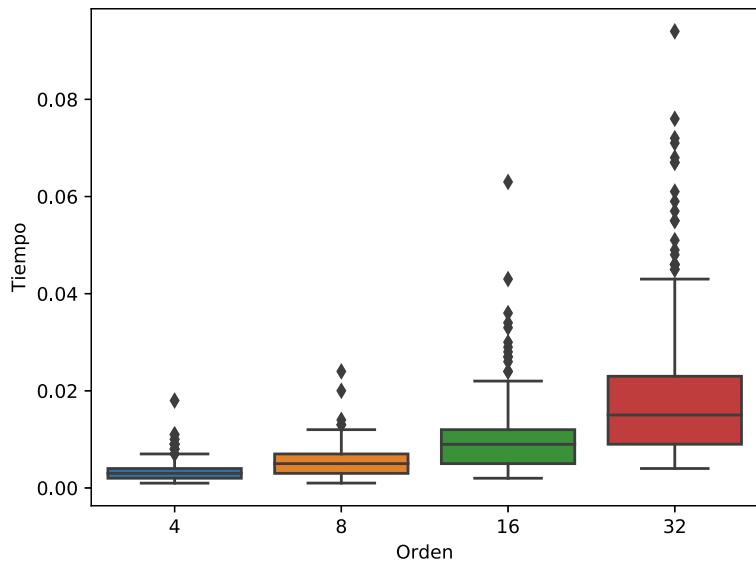


Figura 4: Boxplot Orden

Se realiza además un análisis con un ANOVA para ver si todos los factores estudiados se encuentran relacionados entre sí, lo que se puede decir que como el *valor - p* es pequeño entonces existe vínculos entre dichos factores.

ANOVA.txt

	sum_sq	df	F	PR>F
Generador	0.001308	2.0	22.024254	3.559232e-10
Algoritmo	0.009094	2.0	153.117618	4.343969e-62
Generador:Algoritmo	0.003071	4.0	25.850057	7.452684e-21
Orden	0.021814	1.0	734.561337	9.845713e-136
Generador:Orden	0.001889	2.0	31.801537	2.678743e-14
Orden:Algoritmo	0.002061	2.0	34.695840	1.681675e-15
Densidad	0.000044	1.0	1.472617	2.250941e-01
Generador:Densidad	0.000003	2.0	0.050462	9.507913e-01
Algoritmo:Densidad	0.000026	2.0	0.445265	6.407255e-01
Orden:Densidad	0.000007	1.0	0.229693	6.318402e-01
Residual	0.052890	1781.0	Nan	Nan

Se realiza una matriz de correlación (*Figura 5*) donde se puede comprobar que existe una relación entre factores como se mencionó anteriormente.

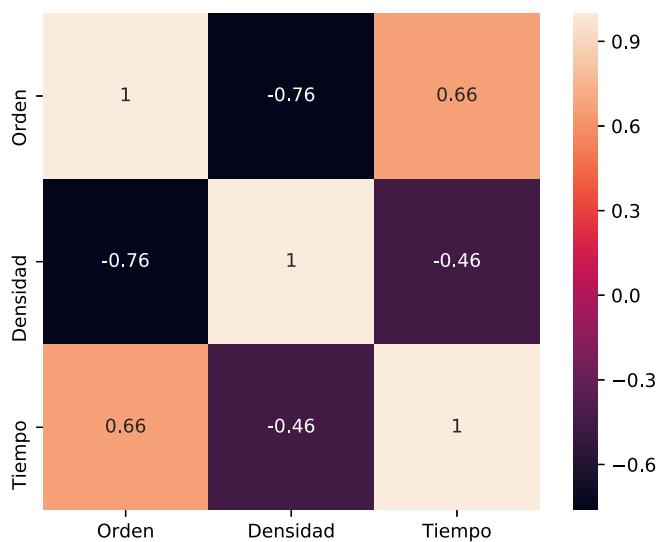


Figura 5: Matriz de Correlación

5. Fragmento de Código

```

1  for j in range(10):
2      for i in range(2, 6):
3          i = 2 ** i
4
5
6      stars_time_G = time()
7      G = nx.star_graph(i)
8      end_time_G = time()
9      total_time_G = end_time_G - stars_time_G
10     total_time_stars.append(total_time_G)
11     ndistribution_weight(8, 0.3, G)
12
13     stars_time_A = time()
14     A = nx.lollipop_graph(3, i)
15     end_time_A = time()
16     total_time_A = end_time_A - stars_time_A
17     total_time_lollipop.append(total_time_A)
18     ndistribution_weight(8, 0.3, A)
19
20     stars_time_D = time()
21     D = nx.wheel_graph(i)
22     end_time_D = time()
23     total_time_D = end_time_D - stars_time_D
24     total_time_wheel.append(total_time_D)
25     ndistribution_weight(8, 0.3, D)
26
27     for r in range(5):
28
29         for k in range(1, 6):
30             list_random_G = random.sample(range(len(G)), 2)
31             inicial_time_G = time()
32             for s in range(1, 6):
33                 value_dinitz = dinitz(G, list_random_G[0], list_random_G[1], capacity="weight")
34
35                 final_time_G = time()
36                 execution_time_G = final_time_G - inicial_time_G
37
38                 table1 = pd.DataFrame({ 'Generador': [ 'Star_Graph' ],
39                                         'Algoritmo': [ 'Dinitz' ],
40                                         'Orden': i,
41                                         'Densidad': round(G.size() / nx.complete_graph(i).size(), 2),
42                                         'Tiempo': execution_time_G + total_time_G })
43
44             table = table.append(table1)
45
46             for h in range(1, 6):
47                 list_random_G = random.sample(range(len(G)), 2)
48                 inicial_time_G = time()
49                 for s in range(1, 6):
50                     value_preflow_push = preflow_push(G, list_random_G[0], list_random_G[1], capacity="weight")
51
52                     final_time_G = time()
53                     execution_time_G = final_time_G - inicial_time_G
54
55                     m = ...
56                     table4 = pd.DataFrame({ 'Generador': [ 'Star_Graph' ],
57                                         'Algoritmo': [ 'Preflow_Push' ],
58                                         'Orden': i,
59                                         'Densidad': round(G.size() / nx.complete_graph(i).size(),
60                                         'Tiempo': execution_time_G + total_time_G })
61
62             table = table.append(table4)

```

new.py

Referencias

- [1] Romain Boulet and Bertrand Jouve. The lollipop graph is determined by its spectrum. *arXiv preprint arXiv:0802.1035*, 2008.
- [2] Khaled Day and Anand Tripathi. Unidirectional star graphs. *Information Processing Letters*, 45(3):123–129, 1993.
- [3] Shimon Even. *Graph algorithms*. Cambridge University Press, 2011.
- [4] Donald B Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977.
- [5] S Kumar and P Gupta. An incremental algorithm for the maximum flow problem. *Journal of Mathematical Modelling and Algorithms*, 2(1):1–16, 2003.
- [6] Nasser R Sabar, Masri Ayob, Graham Kendall, and Rong Qu. Roulette wheel graph colouring for solving examination timetabling problems. In *International conference on combinatorial optimization and applications*, pages 463–470. Springer, 2009.

Tarea 4

5272

1. Introducción

En el presente trabajo se utilizaron tres diferentes tipos de generadores de grafos en este caso los seleccionados fueron: grafo Estrella, grafo Paleta, grafo Rueda; para cada uno se seleccionó una base logarítmica dos para crear cuatro grafos de orden $4, 8, 16, 32$. Los pesos de cada arco se distribuyen normalmente con media ocho y una desviación estándar de 0.3 . Se utilizaron además los algoritmos de flujo máximo: algoritmo Dinitz, Empuje de Preflujo y Ruta de Aumento más Corta.

2. Generadores de Grafos

Los generadores de grafos utilizados se encuentran a continuación:

Grafo Estrella

Este grafo es de orden n , se conoce también como n estrella. Devuelve un grado con $n + 1$ como vértice central, conectado a n nodos externos [2].

Grafo Paleta

Este grafo consiste es un subgrafo completo con m vértices y otro subgrafo con n vértices conectados por medio de un puente [1].

Grafo Rueda

Este grafo también es llamado n ruedas, está formado por la conexión de un solo vértice universal a todos los vértices de un ciclo es decir, contiene un ciclo de orden $n - 1$ para el cual cada vértice del ciclo esta conectado a otro vértice [6].

3. Algoritmos de Flujo Máximo

Los algoritmos de flujo máximo utilizados se encuentran a continuación:

Algoritmo Dinitz

Es uno de los algoritmos más rápidos y fácil de implementar, también se le conoce como algoritmo Dinic. Este algoritmo polinomial calcula el flujo máximo, donde regresa una red residual resultante [3].

Empuje de Preflujo

Este tipo de algoritmo maneja un preflujo, es decir, no se satisface necesariamente las condiciones de conservación de flujo, por lo que es posible que un vértice reciba más flujo que el que distribuye. Los algoritmos de preflujo se pueden mejorar significativamente al incorporar condiciones para decidir que nodos elegir, las cuales tienden a ser heurísticas [5].

Ruta de Aumento más Corta

Encuentra un flujo máximo utilizando el algoritmo de ruta de aumento más corta, regresa una red residual resultante de calcular el flujo máximo [4].

4. Resultados Computacionales

Se realiza una evaluación de los datos utilizando herramientas estadísticas como los Boxplot que se describen a continuación. El primer análisis realizado es del tiempo contra los algoritmos de flujo máximo (*Figura 1*) donde se puede decir que los algoritmos Dinitz, Preflow Push y Shortest tiene una media de *0.005532*, *0.009594* y *0.006333* respectivamente y una desviación de *0.005054*, *0.007593* y *0.004624* para cada uno; el segundo Boxplot (*Figura 2*) es tiempo contra densidad donde la media está en un rango entre *0.06-1.17* con tiempos entre *0.008680-0.003434*; para el tercer Boxplot (*Figura 3*) la media tiene un comportamiento para cada algoritmo de la siguiente forma: Dinitz: *0.009041*, Preflow Push: *0.004764* y Shortest: *0.009435* y una desviación Dinitz: *0.008347*, Preflow Push: *0.003241* y Shortest: *0.006466*; por último se realizó el Boxplot de tiempo contra orden (*Figura 4*) quedando de la siguiente forma: para los algoritmos de orden 4 la media y la desviación tienen un valor de *0.002647* y *0.001058* respectivamente, para los algoritmos de orden 8 un valor de *0.004634* para la media y *0.001991* para la desviación, para orden 16 valores de *0.008008* y *0.003711* respectivamente y por último para orden 32 una media de *0.015339* y una desviación de *0.008589*.

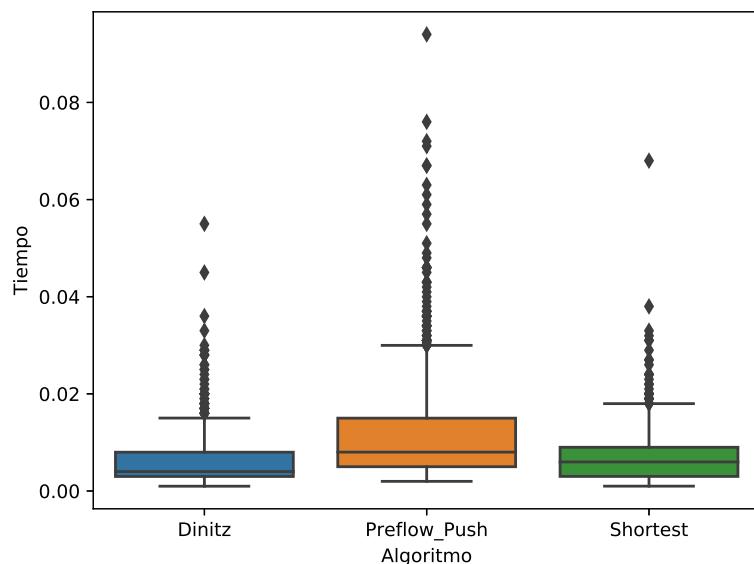


Figura 1: Boxplot Algoritmo

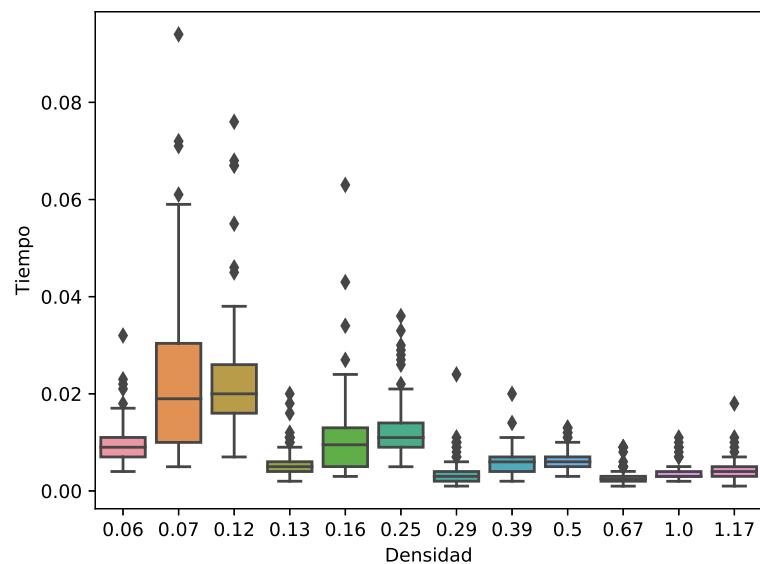


Figura 2: Boxplot Densidad

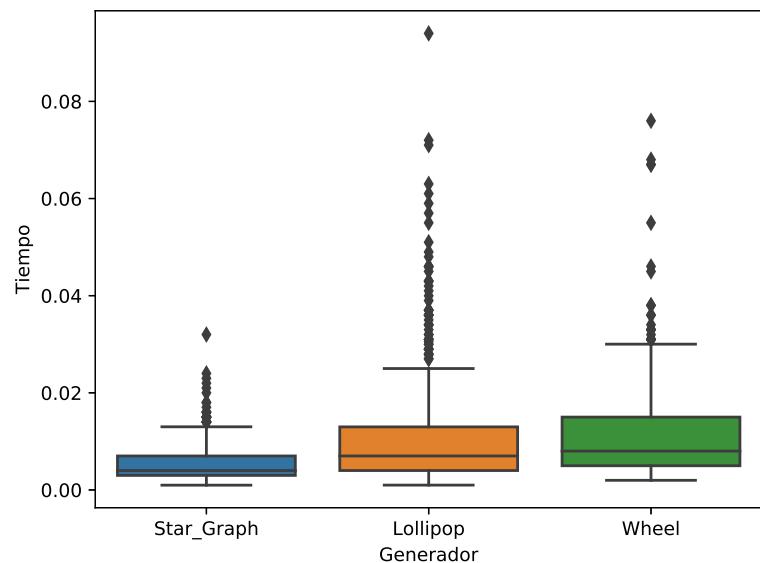


Figura 3: Boxplot Generador

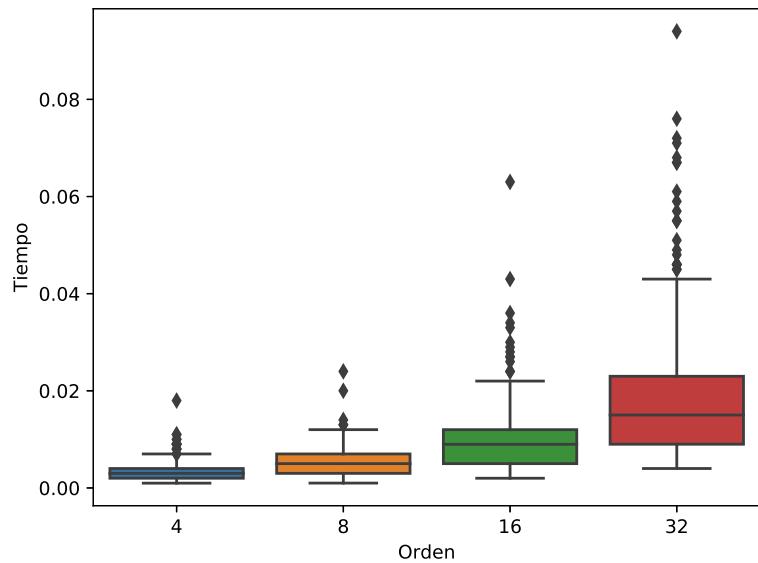


Figura 4: Boxplot Orden

Se realiza además un análisis con un ANOVA para ver si todos los factores estudiados se encuentran relacionados entre sí, lo que se puede decir que como el *valor-p* es pequeño entonces existe vínculos entre dichos factores.

ANOVA.txt

	sum_sq	df	F	PR>F
Generador	0.001308	2	22.02	3.559232e-10
Algoritmo	0.009094	2	153.11	4.343969e-62
Generador:Algoritmo	0.003071	4	25.85	7.452684e-21
Orden	0.021814	1	734.56	9.845713e-136
Generador:Orden	0.001889	2	31.80	2.678743e-14
Orden:Algoritmo	0.002061	2	34.69	1.651675e-15
Densidad	0.000044	1	1.47	2.250941e-01
Generador:Densidad	0.000003	2	0.05	9.507913e-01
Algoritmo:Densidad	0.000026	2	0.44	6.407255e-01
Orden:Densidad	0.000007	1	0.22	6.318402e-01
Residual	0.052890	1781	NaN	NaN

Se realiza una matriz de correlación (*Figura 5*) donde se puede comprobar que existe una relación entre factores como se mencionó anteriormente.

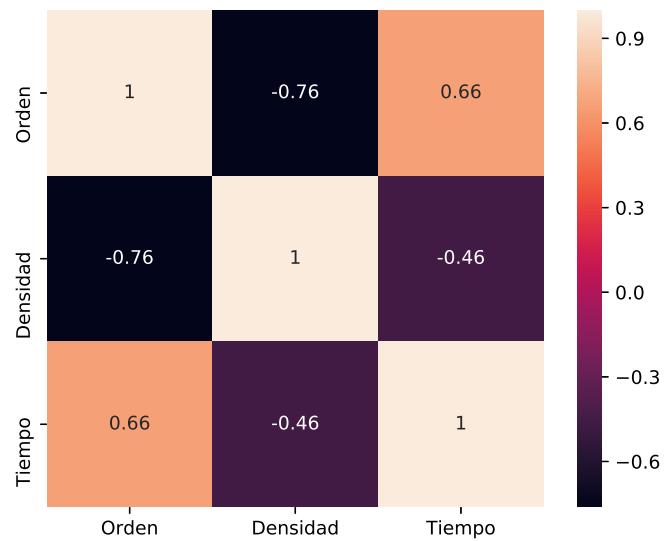


Figura 5: Matriz de Correlación

5. Fragmento de Código

```
1  for j in range(10):
2      for i in range(2, 6):
3          i = 2 ** i
4
5
6      stars_time_G = time()
7      G = nx.star_graph(i)
8      end_time_G = time()
9      total_time_G = end_time_G - stars_time_G
10     total_time_stars.append(total_time_G)
11     ndistribution_weight(8, 0.3, G)
12
13
14     stars_time_A = time()
15     A = nx.lollipop_graph(3, i)
16     end_time_A = time()
17     total_time_A = end_time_A - stars_time_A
18     total_time_lollipop.append(total_time_A)
19     ndistribution_weight(8, 0.3, A)
20
21
22     stars_time_D = time()
23     D = nx.wheel_graph(i)
24     end_time_D = time()
25     total_time_D = end_time_D - stars_time_D
26     total_time_wheel.append(total_time_D)
27     ndistribution_weight(8, 0.3, D)
28
29
30     for r in range(5):
31
32         for k in range(1, 6):
33             list_random_G = random.sample(range(len(G)), 2)
34             inicial_time_G = time()
35             for s in range(1, 6):
36                 value_dinitz = dinitz(G, list_random_G[0], list_random_G[1],
37                                         capacity="weight")
38                 final_time_G = time()
39                 execution_time_G = final_time_G - inicial_time_G
40
41             table1 = pd.DataFrame({ 'Generador': [ 'Star_Graph' ],
42                                    'Algoritmo': [ 'Dinitz' ],
43                                    'Orden': i,
44                                    'Densidad': round(G.size() / nx.complete_graph(i).size(), 2),
45                                    'Tiempo': execution_time_G + total_time_G })
46             table = table.append(table1)
47
48
49             for h in range(1, 6):
50                 list_random_G = random.sample(range(len(G)), 2)
51                 inicial_time_G = time()
52                 for s in range(1, 6):
53                     value_preflow_push = preflow_push(G, list_random_G[0], list_random_G[1],
54                                           capacity="weight")
55                     final_time_G = time()
56                     execution_time_G = final_time_G - inicial_time_G
```

new.py

Referencias

- [1] Romain Boulet and Bertrand Jouve. The lollipop graph is determined by its spectrum. *arXiv preprint arXiv:0802.1035*, 2008.
- [2] Khaled Day and Anand Tripathi. Unidirectional star graphs. *Information Processing Letters*, 45(3):123–129, 1993.
- [3] Shimon Even. *Graph algorithms*. Cambridge University Press, 2011.
- [4] Donald B Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977.
- [5] and Gupta Kumar. An incremental algorithm for the maximum flow problem. *Journal of Mathematical Modelling and Algorithms*, 2(1):1–16, 2003.
- [6] Nasser R Sabar, Masri Ayob, Graham Kendall, and Rong Qu. Roulette wheel graph colouring for solving examination timetabling problems. In *International conference on combinatorial optimization and applications*, pages 463–470. Springer, 2009.

Correcciones realizadas:

En esta tarea se corrigieron los acentos, así como los espacios entre palabras y mejora en la redacción. Se le incluyeron las referencias a las figuras correspondientes; se modificó el estilo de letra en palabras señales. Se modificaron partes del código que tenía espacios subutilizados, así como se eliminaron acentos que generaban errores de lectura.

Tarea 4

5272



1. Introducción

Para realizar el siguiente trabajo se utilizó la librería Networkx de Python y se aplicó el generador de grafo Dorogovtsev-Mendes para obtener un grafo de 14 vértices; los pesos de las aristas tienen una distribución normal con una media de 4 y una desviación estándar de 2.3.

La elección de este generador de grafo es que puede ser aplicado a una red de transporte capacitado, donde se desea enviar cierta cantidad de unidades de un producto desde un almacén hasta un cliente [1]. El algoritmo de acomodo que se utilizó para visualizar el grafo es diseño aleatorio, en este los vértices se diferencian por colores, el azul claro es la fuente mientras el azul oscuro es el sumidero y el color verde representa los restantes vértices; el ancho de las aristas representa la capacidad de estas y luego las que aparecen de rojo son por las que pasa el flujo y las negras son las que el flujo es ~~zero~~.

2. Generador de Grafo

El generador de grafo utilizado se encuentra a continuación:

Generador Dorogovtsev-Mendes

Este generador siempre produce grafos planos, el mismo comienza creando tres vértices y aristas, formando un triángulo, para posteriormente ir incluyendo un vértice a la vez; cada vez que se agrega un vértice, se elige una arista al azar y el vértice se conecta a través de dos nuevas aristas a los dos extremos [2].

3. Algoritmo de Flujo Máximo

El algoritmo de flujo máximo utilizado se encuentra a continuación:

Flujo Máximo

Sea G un grafo, s y t nodos de G y cap una función de capacidad no negativa en las aristas de G . El flujo que pasa de s a t debe satisfacer las restricciones de capacidad, es decir el flujo sobre una arista no debe exceder su capacidad. En un flujo máximo, el flujo de salida de s es máximo entre todos los flujos desde s a t [1].

4. Resultados Computacionales

Los grafos que se muestran en la *Figura 1* representan cada una de las instancias de las ~~X~~ que se utilizaron, donde se pueden observar las fuentes y los sumideros, además se puede decir que hay instancias como la 1 y la 2 donde pasa flujo por todas sus aristas. La relación de fuentes y sumidero para cada instancia se muestra a continuación:

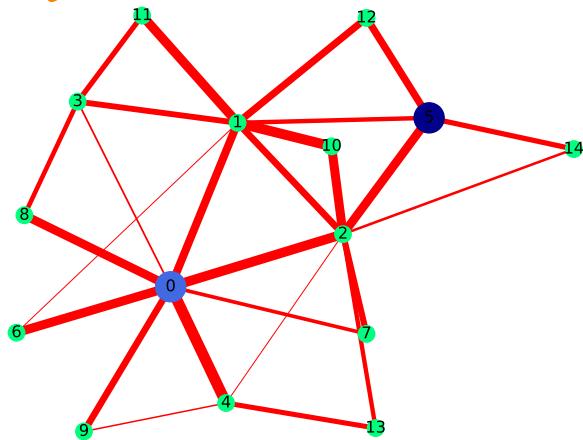
- Instancia 1 $\Rightarrow G: 10, 5$
- Instancia 2 $\Rightarrow G: 1, 5$
- Instancia 3 $\Rightarrow G: 0, 8$
- Instancia 4 $\Rightarrow G: 3, 5$
- Instancia 5 $\Rightarrow G: 5, 0$

Se utilizaron las siguientes características estructurales para cada grafo visualizado:

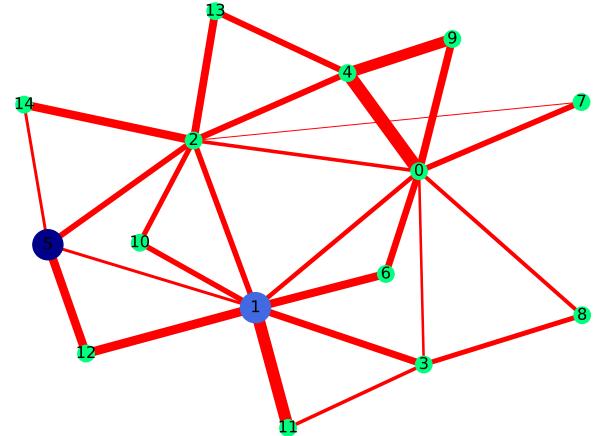
- Distribución de grado.
- Coeficiente de agrupamiento.

- Centralidad de cercanía.
- Centralidad de carga.
- Excentricidad.
- Rango de página.

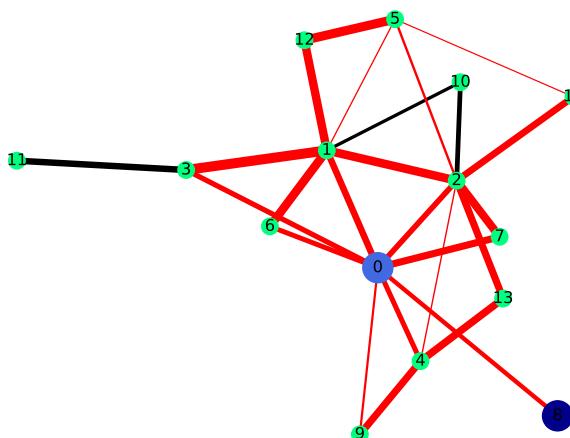
(W)



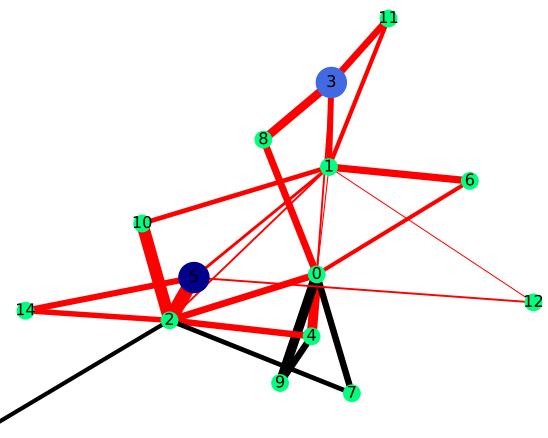
(a) Instancia 1



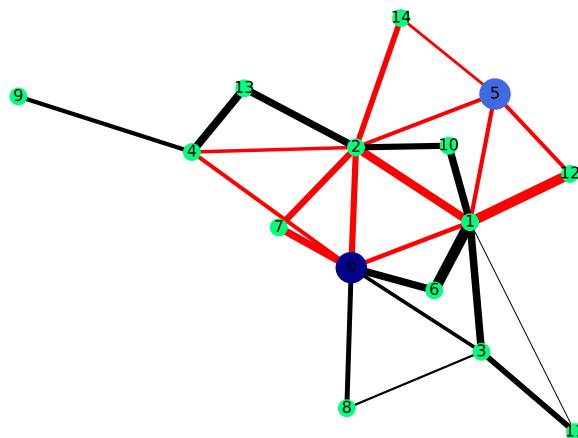
(b) Instancia 2



(c) Instancia 3



(d) Instancia 4



(e) Instancia 5

Figura 1: Instancias

Se analizó para cada instancia cuáles eran los vértices que constituyen buenas fuentes y los que son mejores sumideros *Cuadro 1*, los resultados se muestran a continuación:

	Instancias				
	1	2	3	4	5
Coeficiente de Agrupamiento	4, 2	2, 1	2, 0	0, 12	6, 2
Centralidad de cercanía	1, 0	2, 4	2, 1	2, 1	8, 1
Distribución de grado	2, 1	2, 0	14, 1	1, 3	2, 5
Centralidad de carga	2, 0	2, 1	1, 0	13, 0	3, 7
Excentricidad	5, 2	0, 1	2, 5	5, 7	2, 1
Rango de página	2, 13	3, 6	3, 1	1, 0	0, 9

Cuadro 1: Mejores fuente, sumideros

Se realiza una evaluación de como las características de los vértices que afectan a los tiempos de ejecución de los algoritmos seleccionados, en primer lugar se realizó el análisis para la instancia 1 (*Figura 2*) donde se puede decir que para las características estructurales Centralidad de cercanía y Coeficiente de agrupamiento los nodos tienen comportamientos similares, donde resalta el nodo 7 dado que tiene una duración de tiempo más grande, en el mismo los nodos 7 y 8 son los que poseen mayores funciones objetivos con una valor de 18.43 y 15.44 unidades de flujo para cada estructura respectivamente además que poseen las medias mas elevadas con un valor de 1.23 y 1.35. Para las demás estructuras los nodos tienen un comportamiento similar.

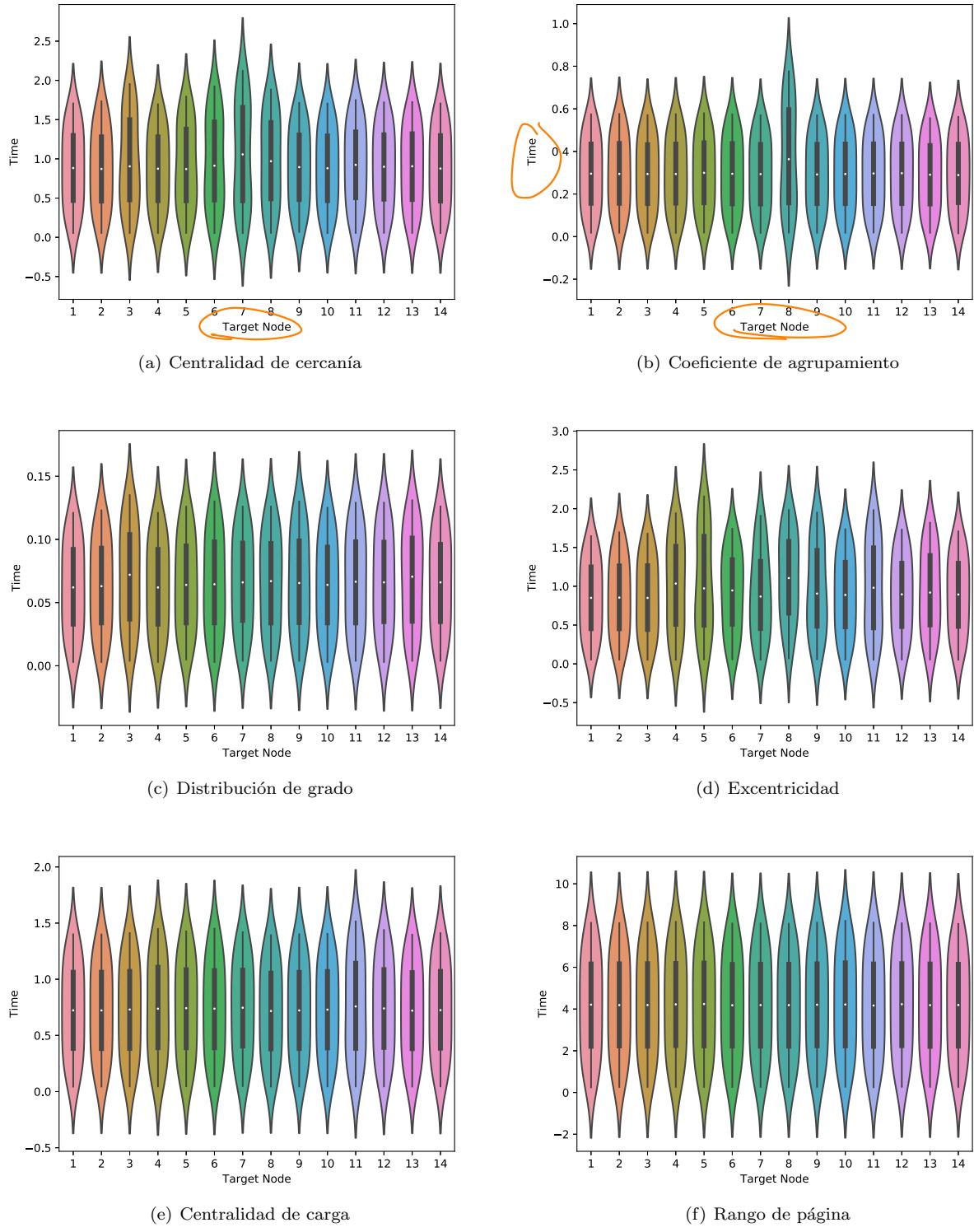


Figura 2: Gráfico de violín: Instancia 1

Para los nodos que pertenecen a la instancia 2 como se observa *Figura 2*, en la estructura de Rango de página es donde mayor se ve afectado el tiempo de ejecución, mientras que en la gráfica (b) el nodo 8 resalta sobre los demás con un mayor tiempo de ejecución el mismo presenta la mayor función objetivo con una valor de 21.28 unidades de flujo, las medias para estas estructuras son muy similares las cuales oscilan alrededor de 0.8.

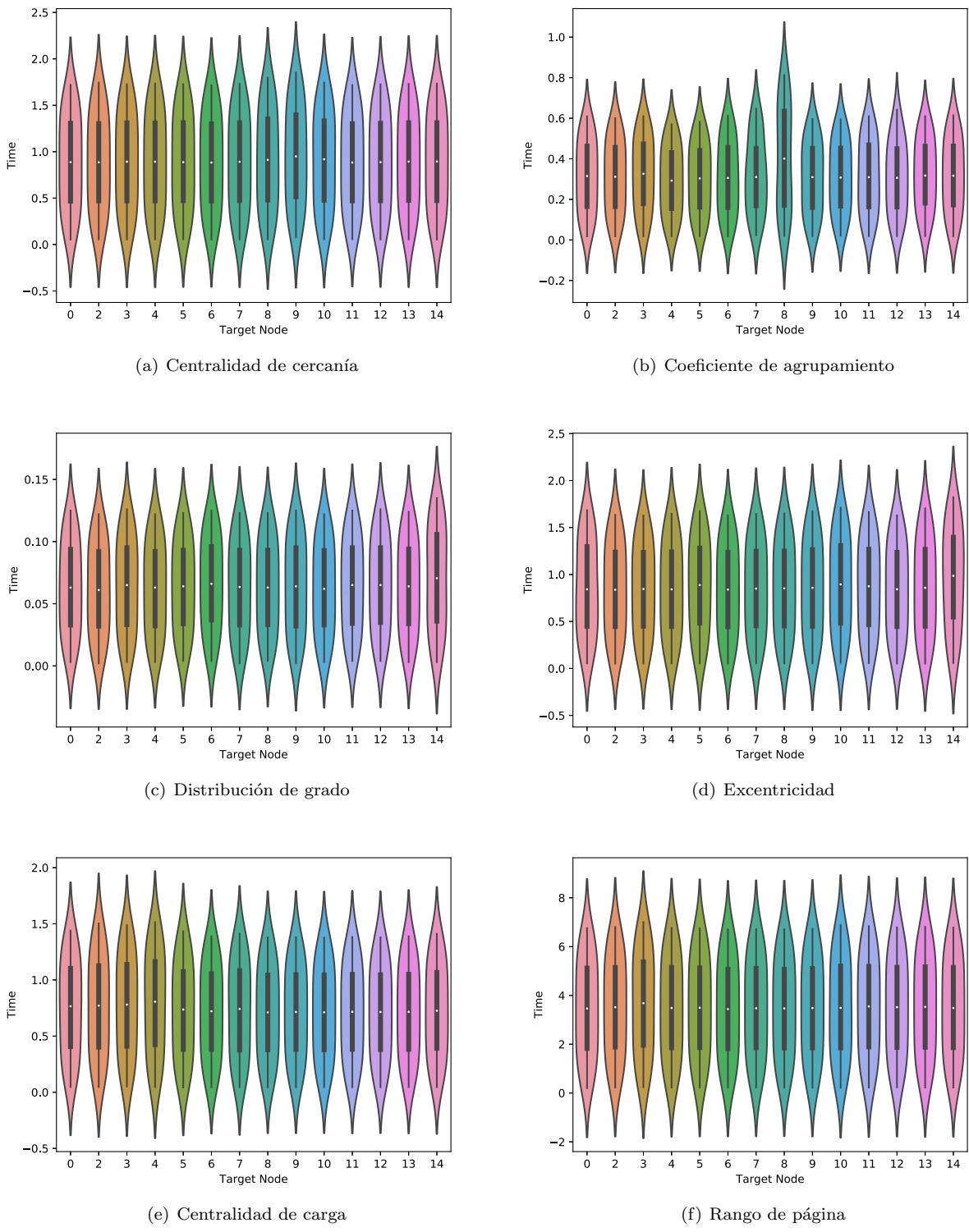
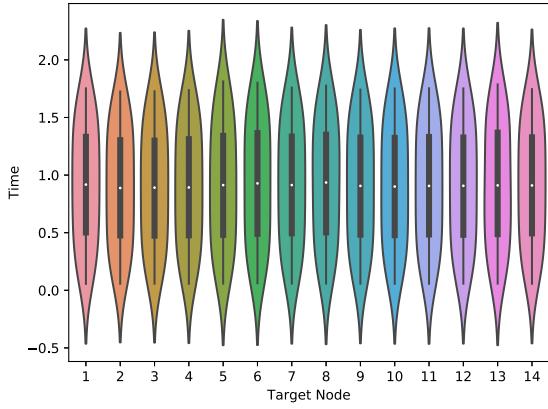
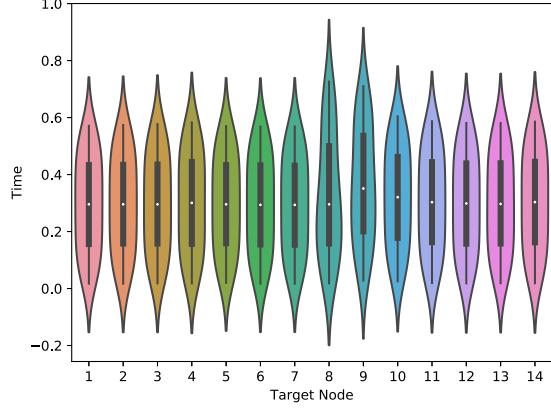


Figura 3: Gráfico de violín: Instancia 2

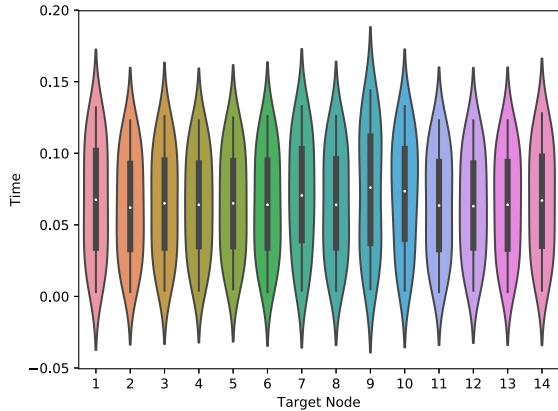
Al analizar la *Figura 3* se puede decir que al igual que la instancia anterior esta tiene un comportamiento similar, pero se puede destacar que la estructura que menos afecta el tiempo es Coeficiente de agrupamiento en esta la mayor función objetivo se encuentra en el nodo 8 con un valor de 21.28 unidades de flujo, las medias oscilan en el valor 0.3.



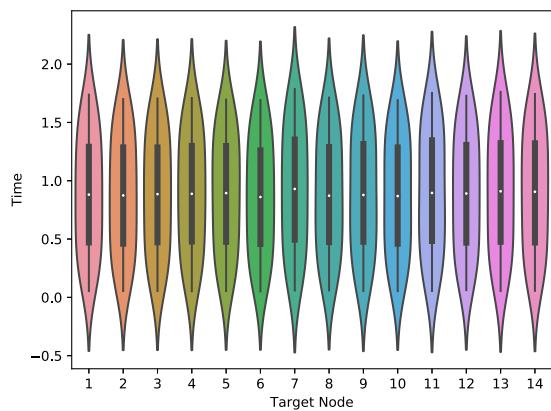
(a) Centralidad de cercanía



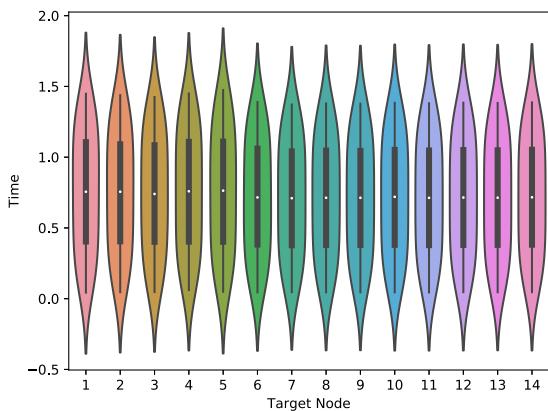
(b) Coeficiente de agrupamiento



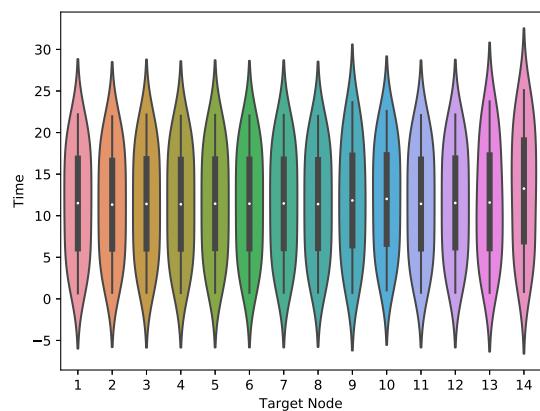
(c) Distribución de grado



(d) Excentricidad



(e) Centralidad de carga



(f) Rango de página

Figura 4: Gráfico de violín: Instancia 3

La cuarta instancia *Figura 4* el mayor valor de la función objetivo se encuentra en la característica estructural Centralidad de carga con un valor de 15.45 unidades de flujo y en este caso los mayores tiempos se encuentran en Centralidad de cercanía, las medias tienen un valor entre los 0.7 y los 0.9.

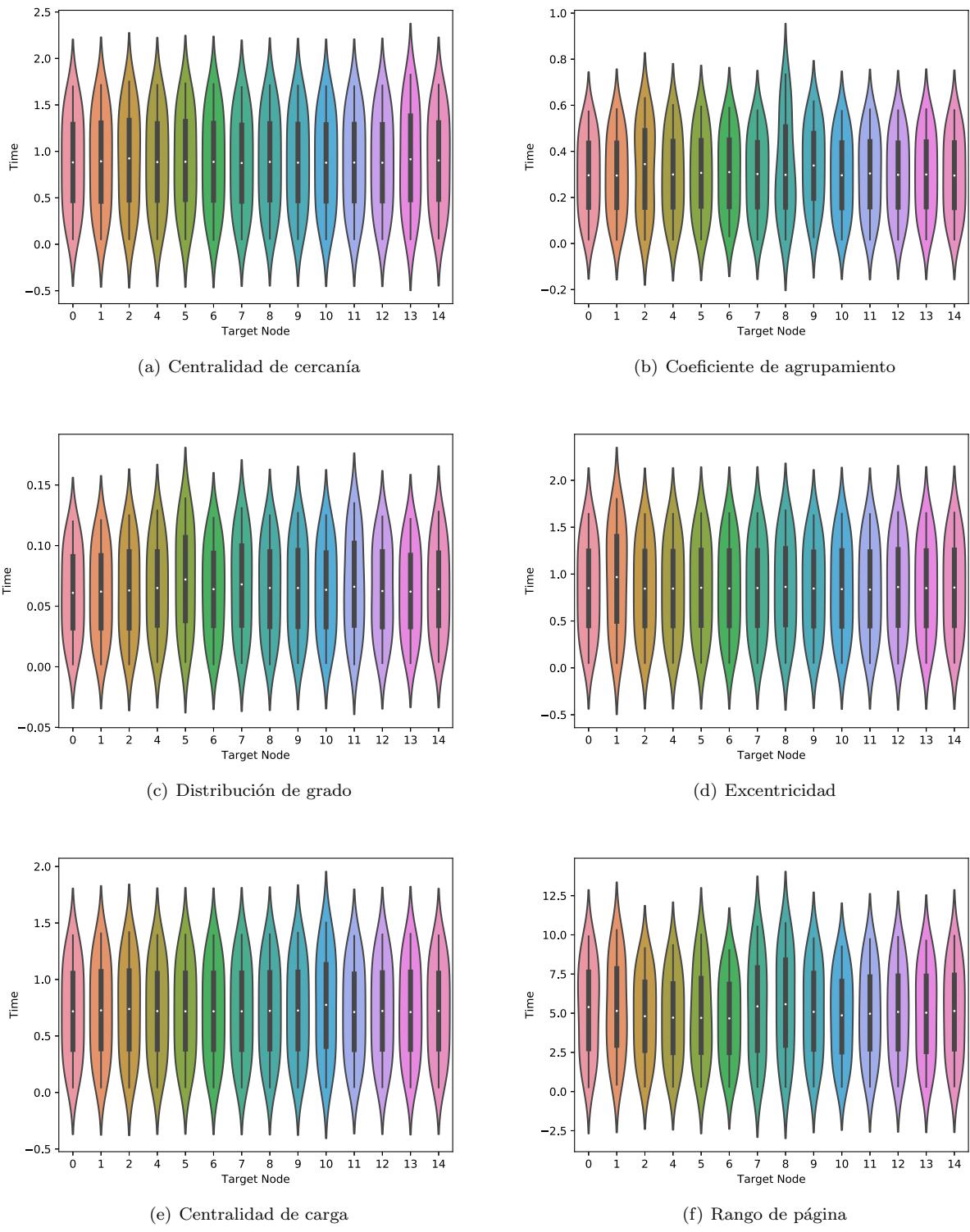


Figura 5: Gráfico de violín: Instancia 4

Por último se analizó la instancia 5 *Figura 5*, en donde los menores tiempos de ejecución se encuentran en Coeficiente de agrupamiento, el mayor valor de la función objetivo se encuentra en Rango de página con un valor de 18.17 unidades de flujo, mientras que las medias para cada una de las características vistas se encuentran alrededor del mismo valor.

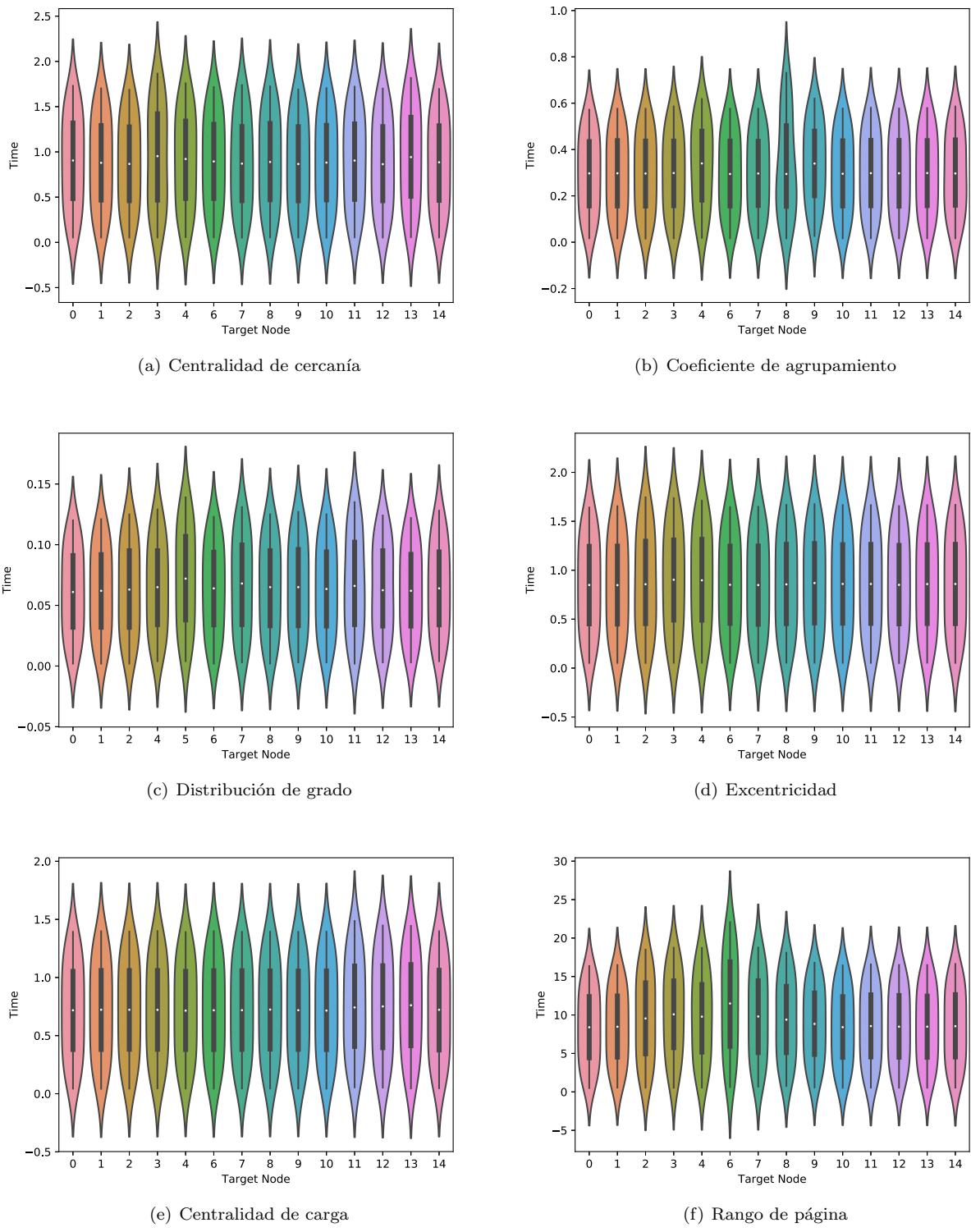


Figura 6: Gráfico de violín: Instancia 5

Se puede concluir que el algoritmo que más afecta al tiempo de ejecución es el Rango de página, mientras que el mayor valor para la función objetivo de todas las características lo presenta Coeficiente de agrupamiento con un valor de 21.28 unidades de flujo, mientras que la que más afecta es la característica estructural Excentricidad con el valor más bajo.

5. Fragmento de Código

```
1 weights = np.random.normal(4, 2.3, nx.number_of_edges(G))
2 m = 0
3 for t, s, p in G.edges(data=True):
4     p[ 'weight' ] = weights [m]
5     m += 1
6
7 st_list = []
8
9 for i in range(1):
10    for j in range(1, 2):
11        list_random_G = random.sample(range(len(G)), 2)
12        for k in range(1, 2):
13            value_dinitz = dinitz(G, list_random_G [0], list_random_G [1], capacity="weight")
14            print("Fuente", list_random_G [0])
15            print("Sumidero", list_random_G [1])
16            nodos_fuente=[]
17            nodos_fuente.append(list_random_G [0])
18            nodos_sumidero = []
19            nodos_sumidero.append(list_random_G [1])
20
21            color_map = []
22            node_T = []
23            for node in G:
24                if node == list_random_G [0]:
25                    color_map.append('royalblue')
26                    node_T.append(500)
27                else:
28                    if node == list_random_G [1]:
29                        color_map.append('darkblue')
30                        node_T.append(500)
31                    else:
32                        color_map.append('springgreen')
33                        node_T.append(150)
34
35            max_flujo , max_dic = nx.maximum_flow(G, list_random_G [0], list_random_G [1],
36                                         capacity='weight')
37            print("Diccionario", max_dic)
38            print("Flujo Maximo", max_flujo)
39            edge_colors = [ 'black' if max_dic [i] [j] == 0 and max_dic [j] [i] == 0 else 'red'
40                            for i, j in G.edges]
41
42 pos = nx.random_layout(G)
43
44 nx.draw(G, node_color=color_map, edge_color=edge_colors, node_size=node_T,
45          width=weights, with_labels=True) #Se dibuja el grafo
```

new.py

Referencias

- [1] Ravindra K Ahuja and James B Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37(5):748–759, 1989.
- [2] Abdelmalik Moujahid and Blanca Rosa Cases Gutiérrez. Analysis of spanish text-thesaurus as a complex network. 2014.

Tarea 5

5272

1. Introducción

Para realizar el siguiente trabajo se utilizó la librería Networkx de Python y se aplicó el generador de grafo Dorogovtsev Goltsev Mendes para obtener un grafo de 14 vértices; los pesos de las aristas tienen una distribución normal con una media de 4 y una desviación estándar de 2.3.

La elección de este generador de grafo es que puede ser aplicado a una red de transporte capacitado, donde se desea enviar cierta cantidad de unidades de un producto desde un almacén hasta un cliente [1]. El algoritmo de acomodo que se utilizó para visualizar el grafo es diseño aleatorio, en este los vértices se diferencian por colores, el azul claro es la fuente mientras el azul oscuro es el sumidero y el color verde representa los restantes vértices; el ancho de las aristas representa la capacidad de éstas y luego las que aparecen de rojo son por las que pasa el flujo y las negras son las que el flujo es cero.

2. Generador de Grafo

El generador de grafo utilizado se encuentra a continuación:

Generador Dorogovtsev-Mendes

Este generador siempre produce grafos planos, el mismo comienza creando tres vértices y aristas, formando un triángulo, para posteriormente ir incluyendo un vértice a la vez; cada vez que se agrega un vértice, se elige una arista al azar y el vértice se conecta a través de dos nuevas aristas a los dos extremos [2].

3. Algoritmo de Flujo Máximo

El algoritmo de flujo máximo utilizado se encuentra a continuación:

Flujo Máximo

Sea G un grafo, s y t nodos de G y cap una función de capacidad no negativa en las aristas de G . El flujo que pasa de s a t debe satisfacer las restricciones de capacidad, es decir el flujo sobre una arista no debe exceder su capacidad. En un flujo máximo, el flujo de salida de s es máximo entre todos los flujos desde s a t [1].

4. Resultados Computacionales

Los grafos que se muestran en la *Figura 1* representan cada una de las instancias de las cinco que se utilizaron, donde se pueden observar las fuentes y los sumideros, además se puede decir que hay instancias como la 1 y la 2 donde pasa flujo por todas sus aristas. La relación de fuentes y sumidero para cada instancia se muestra a continuación:

Instancia 1 $\Rightarrow G: 0, 5$

Instancia 2 $\Rightarrow G: 1, 5$

Instancia 3 $\Rightarrow G: 0, 8$

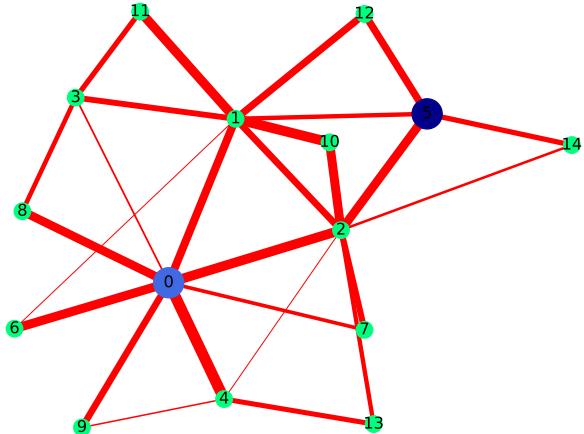
Instancia 4 $\Rightarrow G: 3, 5$

Instancia 5 $\Rightarrow G: 5, 0$

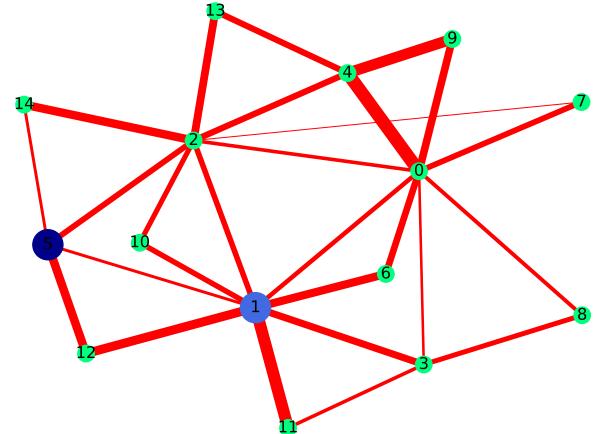
Se utilizaron las siguientes características estructurales para cada grafo visualizado:

- Distribución de grado.
- Coeficiente de agrupamiento.

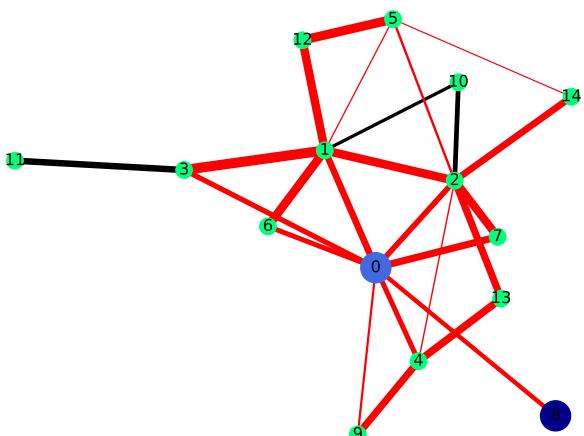
- Centralidad de cercanía.
- Centralidad de carga.
- Excentricidad.
- Rango de página.



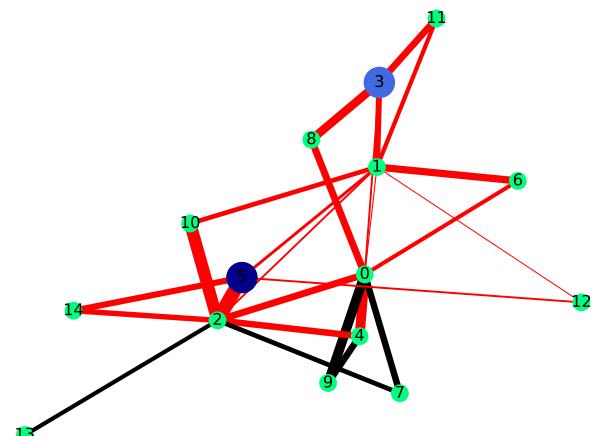
(a) Instancia 1



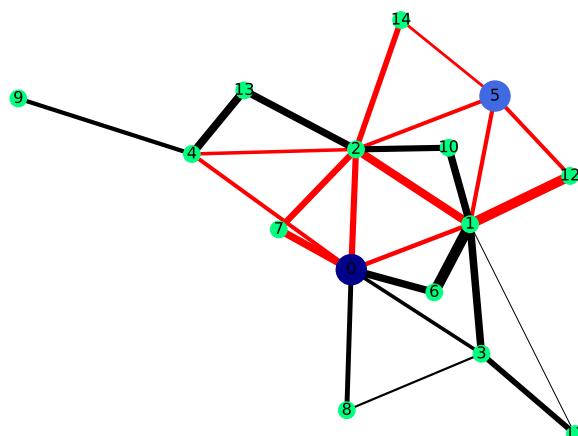
(b) Instancia 2



(c) Instancia 3



(d) Instancia 4



(e) Instancia 5

Figura 1: Instancias

Se analizó para cada instancia cuáles eran los vértices que constituyen buenas fuentes y los que son mejores sumideros *Cuadro 1*, los resultados se muestran a continuación:

	Instancias				
	1	2	3	4	5
Coeficiente de agrupamiento	4, 2	2, 1	2, 0	0, 12	6, 2
Centralidad de cercanía	1, 0	2, 4	2, 1	2, 1	8, 1
Distribución de grado	2, 1	2, 0	14, 1	1,3	2,5
Centralidad de carga	2, 0	2, 1	1, 0	13, 0	3, 7
Excentricidad	5, 2	0, 1	2, 5	5, 7	2, 1
Rango de página	2, 13	3, 6	3, 1	1, 0	0,9

Cuadro 1: Mejores fuente, sumideros

Se realiza una evaluación de como las características de los vértices que afectan a los tiempos de ejecución de los algoritmos seleccionados, en primer lugar se realizó el análisis para la instancia 1 *Figura 2* donde se puede decir que para las características estructurales Centralidad de cercanía y Coeficiente de agrupamiento los nodos tienen comportamientos similares, donde resalta el nodo 7 dado que tiene una duración de tiempo más grande, en el mismo los nodos 7 y 8 son los que poseen mayores funciones objetivos con una valor de 18.43 y 15.44 unidades de flujo para cada estructura respectivamente además que poseen las medias mas elevadas con un valor de 1.23 y 1.35. Para las demás estructuras los nodos tienen un comportamiento similar.

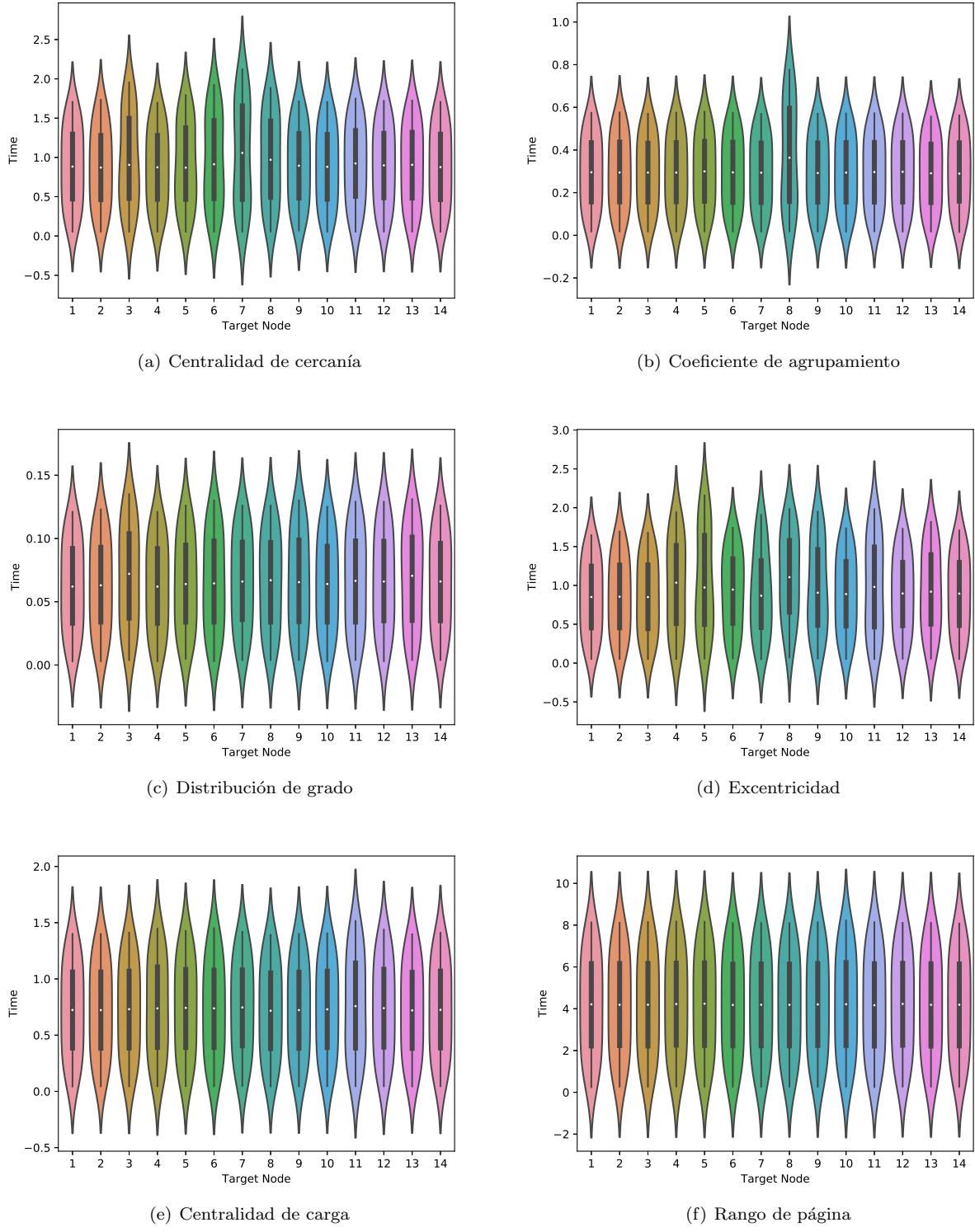
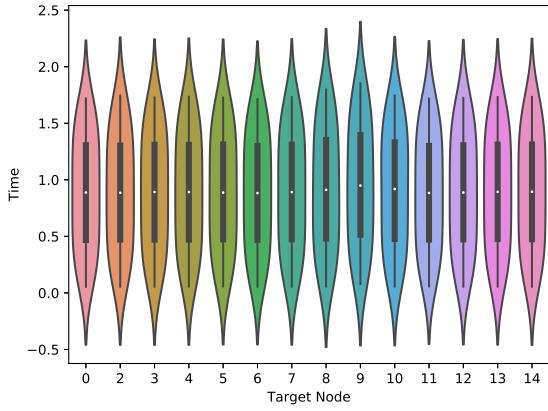
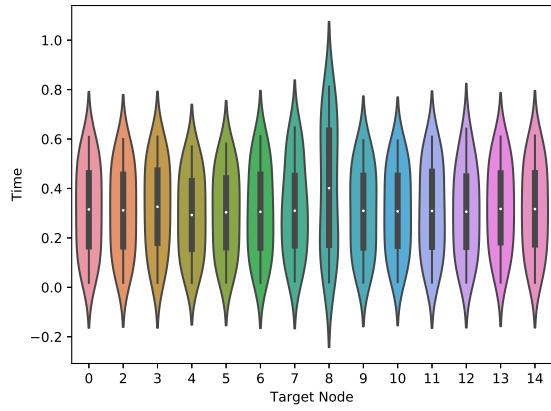


Figura 2: Gráfico de violín: Instancia 1

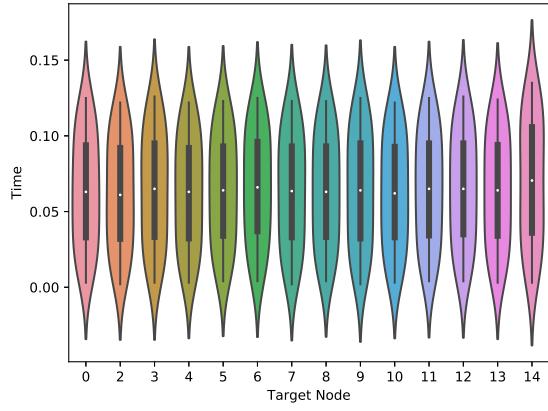
Para los nodos que pertenecen a la instancia 2 como se observa *Figura 2*, en la estructura de Rango de página es donde mayor se ve afectado el tiempo de ejecución, mientras que en la gráfica b el nodo 8 resalta sobre los demás con un mayor tiempo de ejecución el mismo presenta la mayor función objetivo con una valor de 21.28 unidades de flujo, las medias para estas estructuras son muy similares las cuales oscilan alrededor de 0.8.



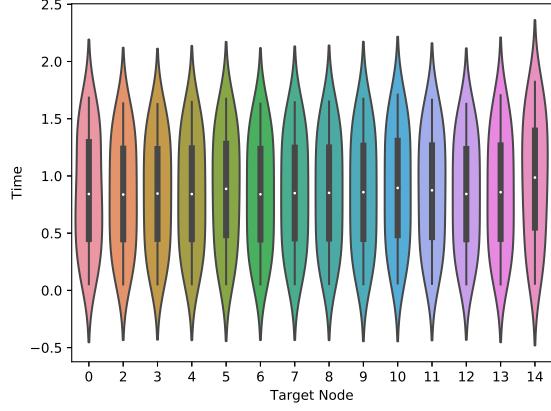
(a) Centralidad de cercanía



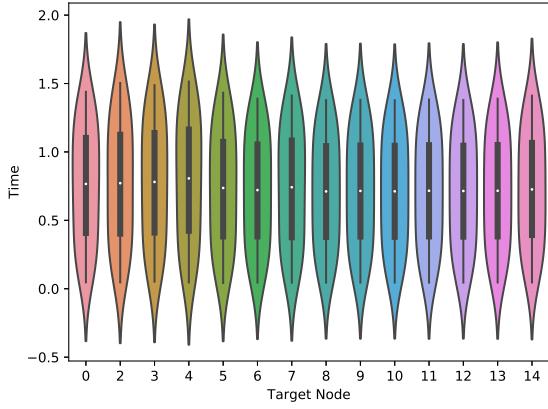
(b) Coeficiente de agrupamiento



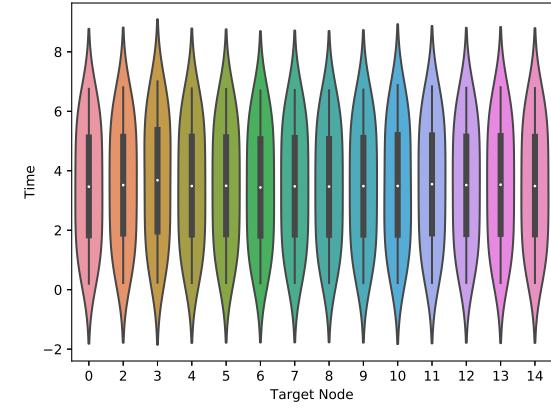
(c) Distribución de grado



(d) Excentricidad



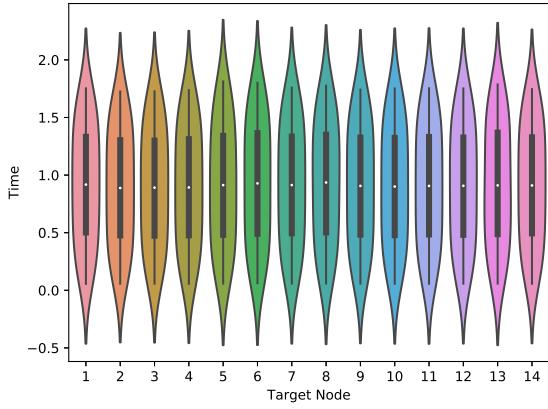
(e) Centralidad de carga



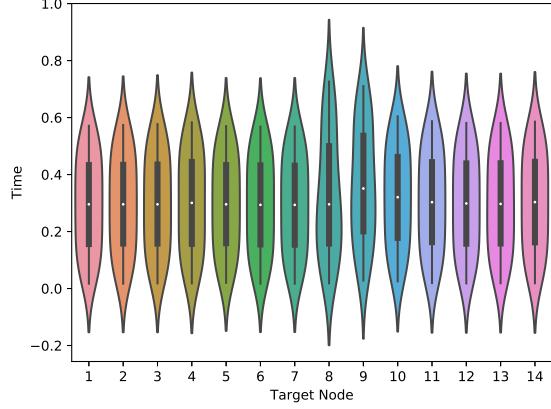
(f) Rango de página

Figura 3: Gráfico de violín: Instancia 2

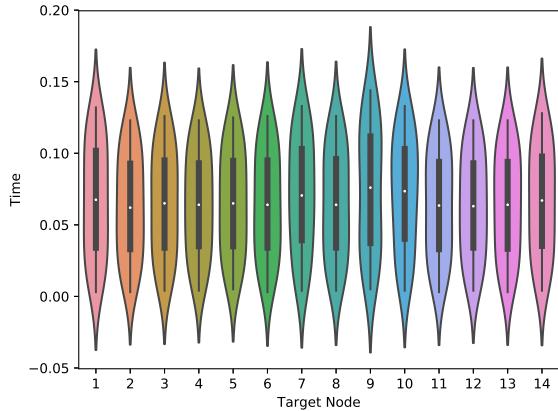
Al analizar la *Figura 3* se puede decir que al igual que la instancia anterior esta tiene un comportamiento similar, pero se puede destacar que la estructura que menos afecta el tiempo es Coeficiente de agrupamiento en esta la mayor función objetivo se encuentra en el nodo 8 con un valor de 21.28 unidades de flujo, las medias oscilan en el valor 0.3.



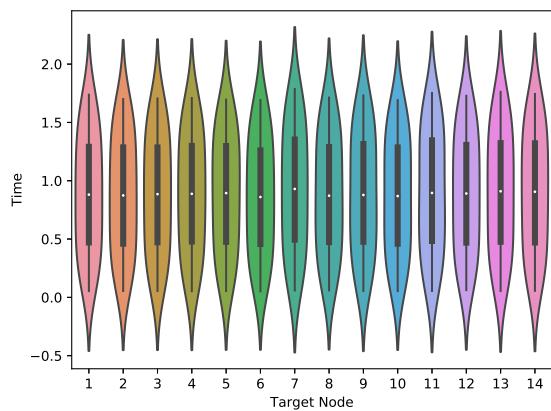
(a) Centralidad de cercanía



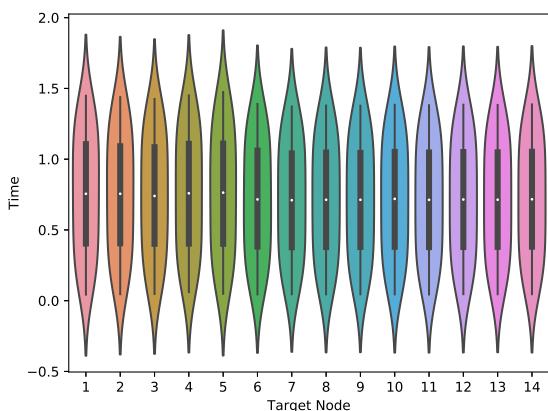
(b) Coeficiente de agrupamiento



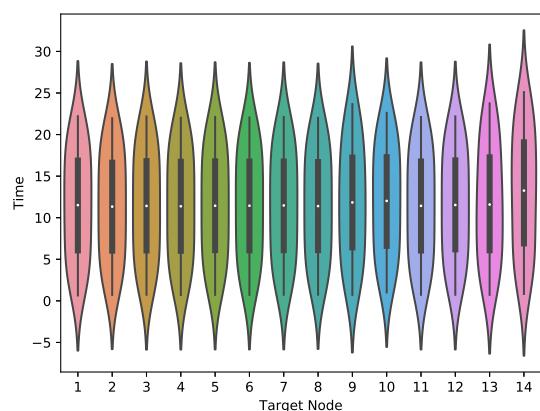
(c) Distribución de grado



(d) Excentricidad



(e) Centralidad de carga



(f) Rango de página

Figura 4: Gráfico de violín: Instancia 3

La cuarta instancia *Figura 4* el mayor valor de la función objetivo se encuentra en la característica estructural Centralidad de carga con un valor de 15.45 unidades de flujo y en este caso los mayores tiempos se encuentran en Centralidad de cercanía, las medias tienen un valor entre los 0.7 y los 0.9.

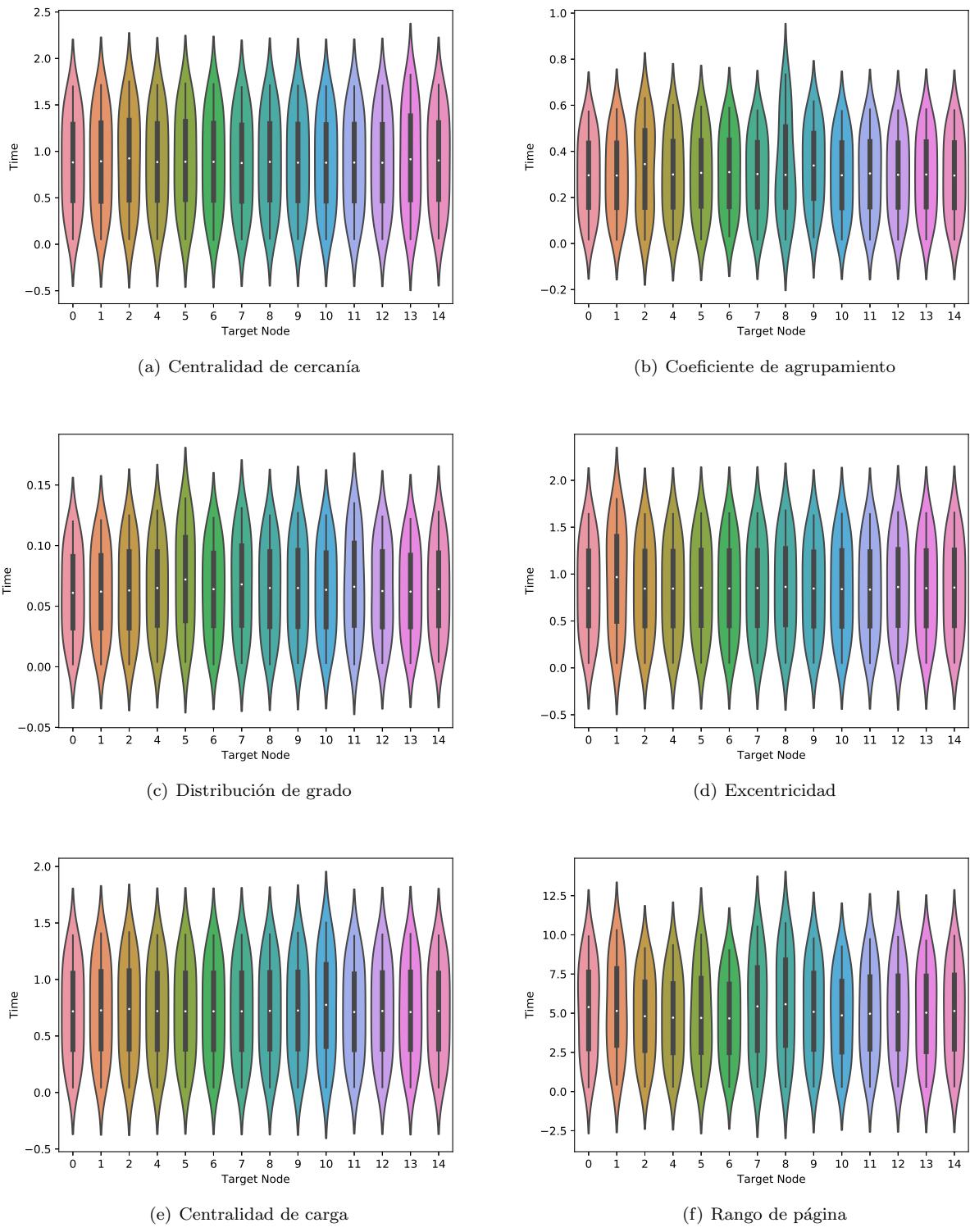


Figura 5: Gráfico de violín: Instancia 4

Por último se analizó la instancia 5 *Figura 5*, en donde los menores tiempos de ejecución se encuentran en Coeficiente de agrupamiento, el mayor valor de la función objetivo se encuentra en Rango de página con un valor de 18.17 unidades de flujo, mientras que las medias para cada una de las características vistas se encuentran alrededor del mismo valor.

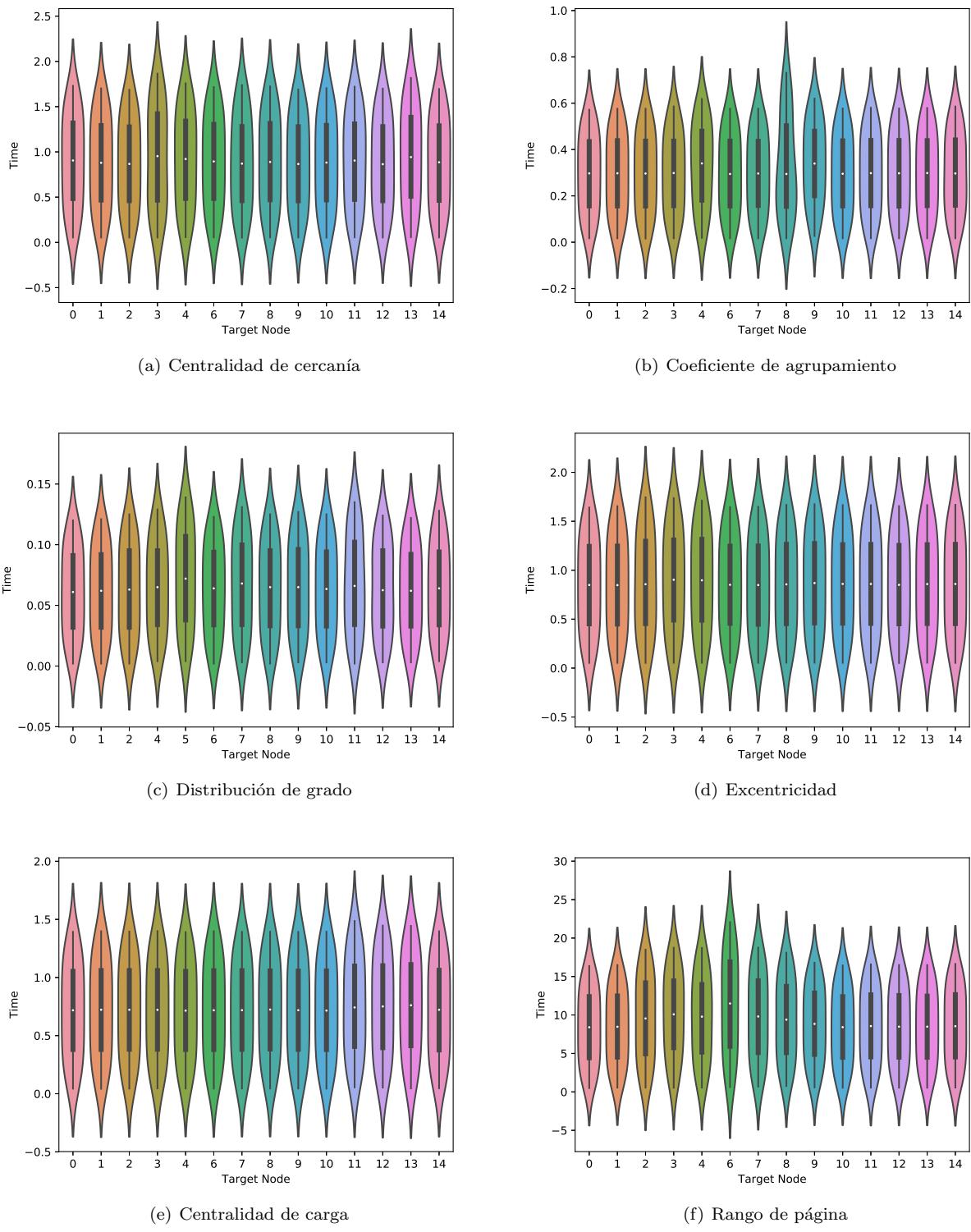


Figura 6: Gráfico de violín: Instancia 5

Se puede concluir que el algoritmo que más afecta al tiempo de ejecución es el Rango de página, mientras que el mayor valor para la función objetivo de todas las características lo presenta Coeficiente de agrupamiento con un valor de 21.28 unidades de flujo, mientras que la que más afecta es la característica estructural Excentricidad con el valor más bajo.

5. Fragmento de Código

```
1 weights = np.random.normal(4, 2.3, nx.number_of_edges(G))
2 m = 0
3 for t, s, p in G.edges(data=True):
4     p[ 'weight' ] = weights[m]
5     m += 1
6
7 st_list = []
8
9 for i in range(1):
10     for j in range(1, 2):
11         list_random_G = random.sample(range(len(G)), 2)
12         for k in range(1, 2):
13             value_dinitz = dinitz(G, list_random_G [0], list_random_G [1], capacity="weight")
14             print("Fuente", list_random_G [0])
15             print("Sumidero", list_random_G [1])
16             nodos_fuente=[]
17             nodos_fuente.append(list_random_G [0])
18             nodos_sumidero = []
19             nodos_sumidero.append(list_random_G [1])
20
21             color_map = []
22             node_T = []
23             for node in G:
24                 if node == list_random_G [0]:
25                     color_map.append('royalblue')
26                     node_T.append(500)
27                 else:
28                     if node == list_random_G [1]:
29                         color_map.append('darkblue')
30                         node_T.append(500)
31                     else:
32                         color_map.append('springgreen')
33                         node_T.append(150)
34
35             max_flujo, max_dic = nx.maximum_flow(G, list_random_G [0], list_random_G [1],
36                                         capacity='weight')
37             print("Diccionario", max_dic)
38             print("Flujo Maximo", max_flujo)
39             edge_colors = [ 'black' if max_dic[i][j] == 0 and max_dic[j][i] == 0 else 'red'
40                             for i, j in G.edges]
41
42 pos = nx.random_layout(G)
43
44 nx.draw(G, node_color=color_map, edge_color=edge_colors, node_size=node_T,
45          width=weights, with_labels=True) #Se dibuja el grafo
```

new.py

Referencias

- [1] Ravindra K Ahuja and James B Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37(5):748–759, 1989.
- [2] Abdelmalik Moujahid and Blanca Rosa Cases Gutiérrez. Analysis of spanish text-thesaurus as a complex network. 2014.

Tarea 6

5272

1. Introducción

En este trabajo se estudiará el sistema de distribución de bienes que presenta una empresa para hacer llegar sus productos a los clientes, para esto cuentan con una flota heterogénea con la que se contará para conformar las diferentes rutas para poder satisfacer la mayor cantidad de productos a los clientes según la demanda, este tema a sido estudiado por diferentes autores donde plantean diferentes soluciones para el mismo [1, 2]. Para llegar al cliente se pueden elegir varios caminos y, dependiendo de este, el algoritmo puede o no alcanzar su solución óptima [3]; lo anterior se puede representar a través de un grafo dirigido o no dirigido, donde uno de los vértices es considerado como el origen y otro como el destino, de tal forma que cualquier material u objeto puede fluir desde el origen hasta el destino, el material que circula por el grafo se le llama flujo. Entre el origen y el destino existe una cantidad determinada de vértices interconectados entre sí a través de arcos, cada uno de estos arcos tiene una capacidad máxima que puede transportar entre los nodos que conecta, la cual puede variar de un arco a otro [4].

Para este poder dar solución a este tipo de problema, se ha utilizado la librería NetworkX de Python para generar un grafo no dirigido, los pesos de las aristas se generaron aleatoriamente; para esto se ha utilizado una CPU AMD A8 a 2.00 GHz de 8 GB.

2. Solución del problema

El caso ha estudiado se representa en un grafo donde el vértice 0 representa el depósito de donde comienza a fluir el material hacia el destino (vértice 1), los datos tienen una media de 29,7306 y una desviación de $3,55e-15$ el grafo se muestra a continuación (*Figura 1*) :

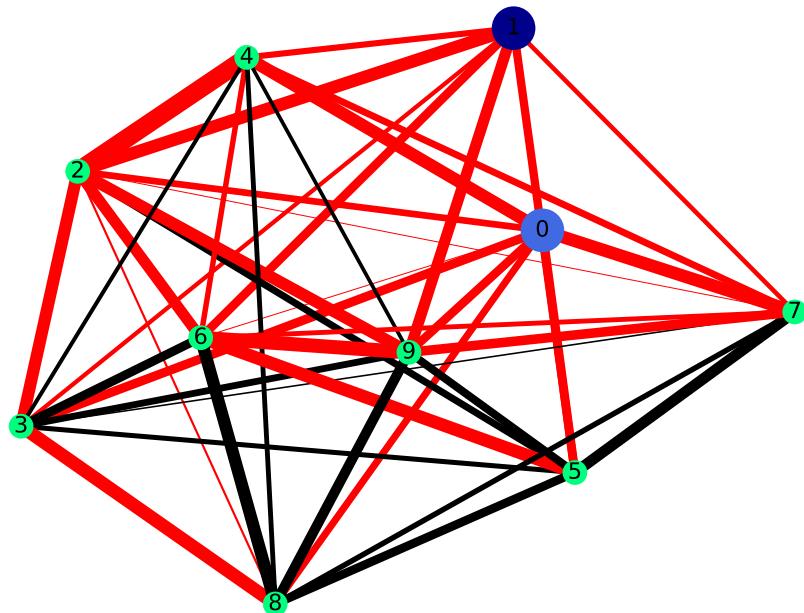


Figura 1: Red de distribución

Con los datos anteriores se construye un histograma (*Figura 2*):

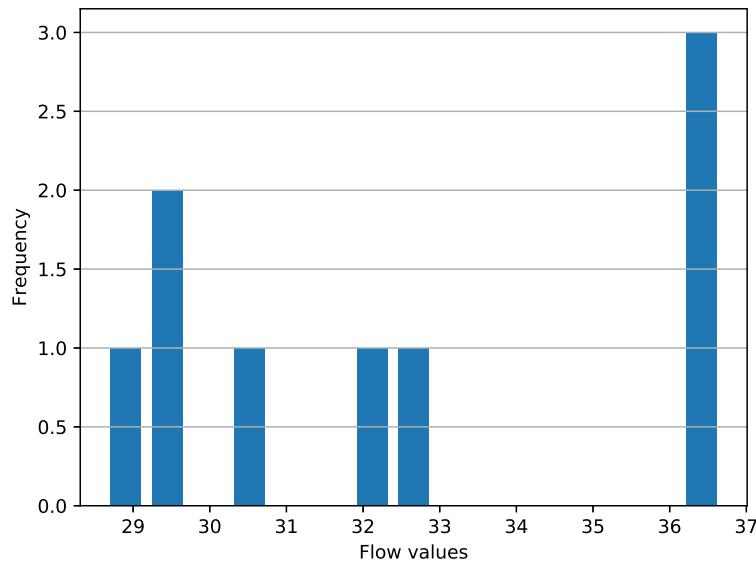


Figura 2: Histograma de flujo

Se realiza la prueba Shapiro-Wilk para probar que los datos se distribuyen normalmente con $p\text{-valor}0,0026 \times 10^{-5}$.

Los tiempos de llegada del flujo a cada vértice presentan una media de 0.049 y un desviación igual 0.029 . Se realizó también la prueba de Shapiro-Wilk para comprobar que los tiempos se distribuyen normal obteniendo un $p\text{-valor}0,507^{-5}$

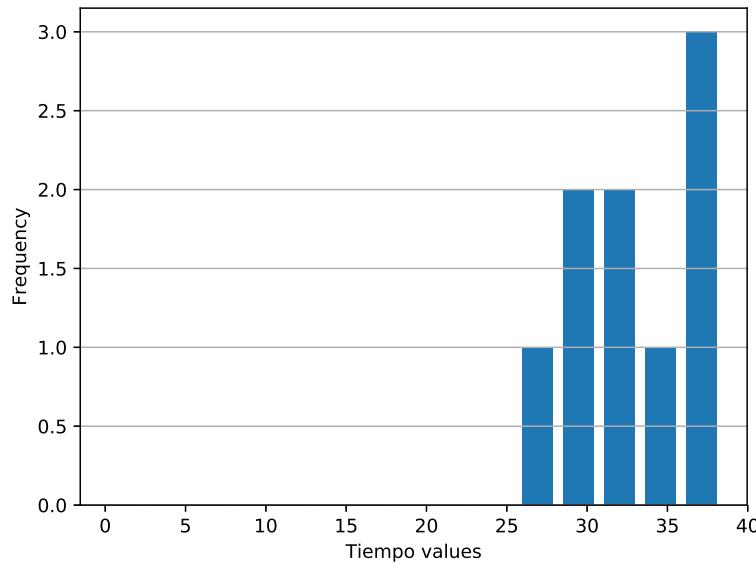


Figura 3: Histograma de flujo

Por último se realiza un análisis de correlación entre la demanda y el tiempo donde se concluyó que uno no depende de otra como se muestra en la (Figura 4).

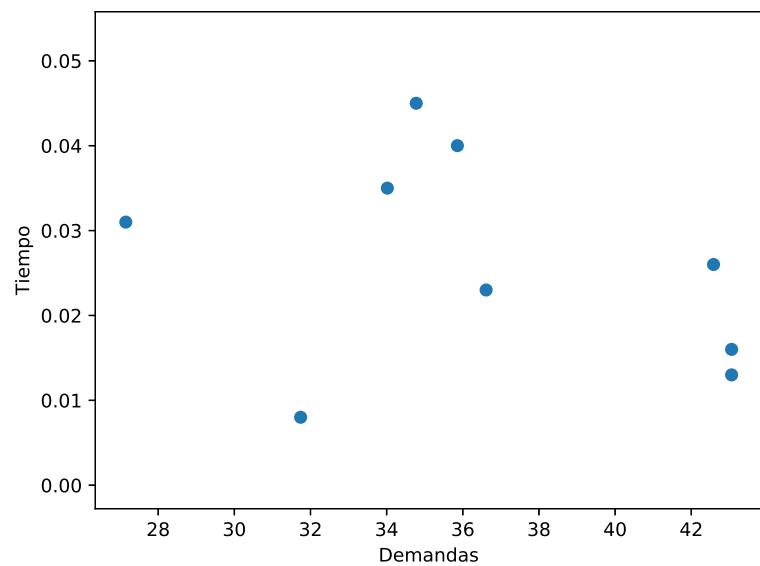


Figura 4: Diagrama de dispersión

3. Fragmento de código

```
1 import networkx as nx
2 from networkx.algorithms.flow import dinitz
3 import matplotlib.pyplot as plt
4 import numpy as np
5 import pandas as pd
6 import random
7 from scipy import stats
8 import seaborn as sns
9
10 G = nx.complete_graph(10)
11
12 weights = np.random.normal(4, 2.3, nx.number_of_edges(G))
13 m = 0
14 for t, s, p in G.edges(data=True):
15     p['weight'] = weights[m]
16     m += 1
17
18 st_list = []
19
20 for j in range(1, 2):
21     list_random_G = random.sample(range(len(G)), 2)
22     for k in range(1, 2):
23         value_dinitz = dinitz(G, 0, 1, capacity='weight')
24
25         color_map = []
26         node_T = []
27         for node in G:
28             if node == 0:
29                 color_map.append('royalblue')
30                 node_T.append(500)
31             else:
32                 if node == 1:
33                     color_map.append('darkblue')
34                     node_T.append(500)
35                 else:
36                     color_map.append('springgreen')
37                     node_T.append(150)
38
39 max_flujo, max_dic = nx.maximum_flow(G, 0, 1, capacity='weight')
40
41 #df= pd.DataFrame({'Cluster':[1], 'Flow_Value':max_flujo})
42 #all_data=all_data.append(df)
43
44 print("Diccionario", max_dic)
45 print("Flujo Maximo", max_flujo)
46 edge_colors = ['black' if max_dic[i][j] == 0 and max_dic[j][i] == 0 else 'red' for i, j in G.edges]
47
48 pos = nx.random_layout(G)
49
50 nx.draw(G, node_color=color_map, edge_color=edge_colors, node_size=node_T, width=weights,
51          with_labels=True) #Se dibuja el grafo
52 plt.savefig("Tarea5_01.eps")
53 plt.show() #Se dibuja en pantalla
```

Tarea6.py

Referencias

- [1] Nicos Christofides and Samuel Eilon. An algorithm for the vehicle-dispatching problem. *Journal of the Operational Research Society*, 20(3):309–318, 1969.
- [2] Julio Mario Daza, Jairo R Montoya, and Francesco Narducci. resolución del problema de enrutamiento de vehículos con limitaciones de capacidad utilizando un procedimiento metaheurístico de dos fases (solving the capacitated vehicle routing problem using a twophase metaheuristic procedure). *Revista EIA*, 6(12):23–38, 2013.
- [3] Armin Lüer, Magdalena Benavente, Jaime Bustos, and Bárbara Venegas. El problema de rutas de vehículos: Extensiones y métodos de resolución, estado del arte. In *EIG*, 2009.
- [4] FRANCISCO JAVIER DANITZ Miller. Métodos de asignación de peajes de los sistemas de transmisión eléctrica según el uso de la red. *Santiago de Chile*, 2001.